IMPLEMENTATION, INTEGRATION, AND APPLICATION OF EMBEDDED DOMAIN-SPECIFIC LANGUAGES

by

William Gallard Hatch

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

TODO - submit date

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

William Gallard Hatch

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Matthew Flatt
	Eric Eide
	John Regehr
	Vivek Srikumar
	Tijs van der Storm

ABSTRACT

Domain-specific languages (DSLs) provide features such as notations, semantics, abstractions, and more that help programmers solve domain problems in more direct, correct, or flexible ways than using a general-purpose language. Embedded DSLs expand on the benefits of DSLs by providing an easier path to implementation, as well as better interoperability with a host language and other DSLs embedded into the same host. However, embedded DSLs present additional challenges for implementation, integration, and application.

This dissertation addresses various challenges of embedded DSLs. We address parsing embedded domain-specific notations with a novel parsing algorithm. Our parsing algorithm admits arbitrary, ambiguous, left-recursive, procedural parsers, advancing the state of the art for expressive parsing. We address host language and DSL integration with the example of an embedded shell language. This shell language demonstrates design for tight integration, allowing smooth growth from interactions to programs. We address DSL application with the example of a DSL for creating random program generators. This DSL allows fuzzers for programming language implementations to be created quickly and with low effort.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

TODO - write personal acknowledgements.

This material is based upon work supported by the National Science Foundation under grant number 1526324, grant number 1823244, and grant number 1527638.

CHAPTER 1

INTRODUCTION

A domain-specific language (DSL) provides programmers with a framework for solving problems limited to a particular problem domain rather than being a general-purpose tool. A DSL may provide domain-specific notations, semantics, data structures, or abstractions, or any combination of these features. DSLs may be created for various purposes, such as to improve programming convenience, program correctness, program legibility, programmer productivity, or accessibility to domain experts who are not programming experts.

While DSLs may be written as stand-alone languages, DSLs often need to communicate with general-purpose languages (GPLs) or rely on general-purpose features as part of the language such as control flow or data structures. Embedded DSLs (EDSLs) solve this problem by being embedded directly into a GPL. This embedding allows EDSLs to share components with the host GPL, easing DSL development, communication between the DSL and the host, and even fine-grained mixing of the DSL and the host within a single file, module, or expression. Because embedding eases these concerns, it also allows multiple DSLs to be mixed together with much greater ease.

While EDSLs provide many benefits, they also pose some challenges. Parsing is more complicated when dealing with the concrete syntax of a composite language with multiple notations. Identifying useful domain targets, designing DSLs, and implementing them is a challenge requiring expertise in problem domains as well as in programming language design and implementation. Even when a good domain and language design are known, it can be challenging to design a good embedding of a DSL within a host language.

This dissertation advances knowledge in language-oriented programming in the implementation, integration, and application of domain-specific languages. We explore the implementation of DSLs through a novel parsing algorithm that improves the expressiveness and flexibility in the concrete syntax of DSLs. We explore the integration of DSLs through Rash, a command language that allows shell commands and Racket functions to be mixed at the expression level. We explore the

application of DSLs through Xsmith, a DSL for defining automatic program generators for testing language implementations.

In our first project we design and implement a parsing system that allows improved expressivity and compositionality of domain-specific and general-purpose programming syntax. While parsing is a problem as old as programming, it still poses significant challenges for defining and mixing domain-specific notations. Our algorithm is the first parsing algorithm to support arbitrary parsing procedures, including left-recursive procedures, as parsing productions; to support data-dependent and other context sensitive constructions; to support the full class of context-free grammars, including ambiguous grammars; to support dynamic extensibility; and to support grammar definition and composition with any mix of ad hoc procedures, parser combinators, and declarative formats like Backus-Naur Form (BNF). Other algorithms lack support for many of these features, making the definition and composition of many notations difficult, impractical, or impossible. Our algorithm includes a novel method of handling the left-recursion problem in parsing by using first-class delimited continuations. We implement this algorithm as a Racket library called Chido Parse.

Our second project, Rash, is a command language embedded in the general-purpose language Racket. Command languages like the Bourne Shell provide users the ability to richly interact with computing environments, and allow users to capture those interactions and convert them into reusable programs. However, there are tensions between interactive conveniences and features needed for reliable, maintainable programs. Typically, command languages allow convenient interactions and growth to scripts that are near-literal transcriptions of interactions, but fail to allow a gradual and full growth from these simple scripts to larger programs. Rash provides a pipeline semantics that allows extensible, fine-grained mixing of operating system subprocesses and Racket functions. Additionally, Rash provides a concrete syntax that allows the recursive mixing of line-oriented shell-style code and Racket code. Rash allows a smooth path for interactions to grow not only to simple scripts, but into full programs that can utilize all Racket features.

The goal of our third project is to develop a domain-specific language for programming language implementation fuzzing. Fuzzers are helpful in finding bugs that developers, test suites, and type systems have missed. However, many fuzzers can only find limited sets of shallow bugs, such as crashing bugs in early stages of parsing or input analysis. Fuzzers that produce grammatically and semantically valid inputs can find deep bugs that pass many layers of validation code, and differential testing provides an oracle for a much broader class of bugs than simply detecting crash

bugs. But fuzzers that produce tests to be useful in differential testing are difficult and expensive to build. We propose a domain-specific language called Xsmith for building highly effective fuzzers for programming languages. Xsmith allows users to specify their language's grammar, type system, and other rules to create a fuzzer that produces valid, deterministic programs. These fuzzers are cheap to write with few lines of code, but are capable of finding deep bugs.

These projects advance the state of the art in the implementation, integration, and application of embedded domain-specific languages. By admitting arbitrary, ambiguous, left-recursive, procedural parsers, Chido Parse advances the state of the art for expressive parsing for embedded and mixed notations. Rash demonstrates the design for tight integration of an embedded shell, allowing smooth growth from interactions to programs. Xsmith applies EDSL techniques to allow fuzzers to be implemented with low cost.

CHAPTER 2

EXPRESSIVE PARSING WITH DELIMITED CONTINUATIONS

When creating embedded DSLs, any custom notation is constrained by parsing technology. Different parsing technologies support different language classes, and provide different expressive affordances to programmers. Different parsing systems also have very different attributes and affordances when considering language composition. If a custom notation and the host notation are to be mixed together, a composite parser must be created. While a single composite parser may be written, it requires more effort than simply composing two parsers where possible. Also, writing composite parsers from scratch rather than composing existing parsers does not scale to composing several languages (all together or in pairs).

Recursive descent parsers can handle context-sensitive grammars, they can be written in a direct, procedural style, and they can be extended easily with new parsing procedures—even dynamically. However, recursive descent parsers do not support all context-free grammars, at least not in a direct and extensible form, because they do not support ambiguous or left-recursive grammars.

Generalized parsing algorithms like GLL (Scott and Johnstone 2010), meanwhile, support all context-free grammars, including ambiguous and left-recursive grammars, they allow for flexible grammar composition, and they support declarative interfaces like Backus Naur Form (BNF) (McCracken and Reilly 2003). Then again, GLL is limited to context-free grammars, while many constructs used in programming languages are not context-free. While procedural recursive descent and GLL offer complimentary strengths, they have not previously been composable due to the left-recursion problem.

Our novel parsing algorithm in Chido Parse combines the benefits of recursive descent parsing and GLL parsing. To handle left recursion and ambiguity with procedural parsing, the algorithm captures and schedules delimited continuations to resolve dependency cycles and to explore alternative parses. Chido Parse supports the entire class of context-free grammars and beyond, while

allowing programmers to write and compose parsers using any mix of parser combinators, declarative formats, and ad hoc procedures. The algorithm thus supports the entire class of context-free grammars and beyond, while allowing programmers to write and compose parsers using any mix of parser combinators, declarative formats, and ad hoc procedures.

2.1 Introduction

Procedural recursive descent parsing is a simple strategy that has many benefits. First, programmers use the parsing system by simply writing procedures in their programming language of choice. Second, the parsing system can provide easy hooks for extension. For example, Racket's S-expression parser uses a *readtable*, which is a table mapping short character prefixes to parsing procedures, and users can create new parsers by extending the current readtable with custom procedures. Finally, procedural parsing systems can support dynamic extension. For example, Racket's default S-expression parser includes a `#reader` production, which causes the parser to dynamically load a new module that is named by the next token; that module supplies a parsing procedure to continue reading from the current input stream. Using this mechanism, programmers can extend the grammar of their language on the fly, without needing to create separate modules for every combination of parsers.

Although it has several benefits, classical procedural parsing also has significant limitations. Procedural parsers are easy to compose in series, but they are difficult to compose as alternates; some framework is required to determine which procedure is appropriate to use at a given point in the input stream. This weakness is reflected in the traditional readtable API, which dispatches to a parsing procedure based on a one- or two-character lookahead. A two-character lookahead is not even good enough for Racket's base syntax, which includes both `#reader` and `#rx` forms; parsers for those forms must be implemented within a single procedure in Racket's parser. Also, while procedural recursive descent parsers can parse languages outside the formal category of context-free grammars, they cannot parse all context-free grammars conveniently. Left-recursive grammars, including grammars with the natural specification of infix operators, would cause infinite recursion if they were expressed directly in recursive descent.

In contrast to recursive descent parsers, generalized parsing systems such as GLL (Scott and Johnstone 2010) and GLR (Tomita 1985) support the full range of context-free grammars, including ambiguous and left-recursive grammars. Additionally, because context-free grammars are closed

under union, concatenation, and repetition, generalized systems based on context-free grammars support rich composition capabilities. But while these systems support the entire set of context-free grammars, they do not support context-sensitive grammars, which are needed by features of many popular programming languages. For example, the off-side rule used by Haskell and Python and the @-expression notation used in Racket's documentation system are both context-sensitive. While there are some extensions to GLL and GLR to support some context-sensitive languages, they also do not support embedding existing or custom parsing procedures into a grammar, like the dynamic grammar extension of Racket's `#reader`.

Our new parsing system, Chido Parse, is organized as an extended GLL to combine the benefits of GLL with the benefits of procedural parsing. It allows procedures as parser productions, and it uses delimited continuations (Danvy and Filinski 1990; Felleisen 1988; Sitaram 1993) to detect and manage left recursion in procedural sub-parsers. To handle ambiguity, parsers can return multiple parse derivations, and a tentative result from a parsing alternative can be discarded if it does not succeed in later sequential combinations. By admitting arbitrary, ambiguous, left-recursive, procedural parsers, Chido Parse achives expressiveness beyond that of existing parsing systems.

Our main contributions are the following:

- A new method for allowing left recursion, including hidden left recursion, in procedural parsing by using delimited continuations.

- A generalization of the recursive descent and GLL algorithms to allow grammar productions to be defined by arbitrary procedures.

- A concrete implementation of this system as a library called Chido Parse.

- A demonstration that this new algorithm and library improve on the state of the art for the creation of composable and extensible grammars for programming languages, discussed in Section 2.6.1.

2.2 Motivating Examples

As a running example for this section, we will discuss an infix arithmetic expression grammar. We will begin with just basic components, and eventually reach a language with the grammar shown in figure 2.1, capable of parsing expressions such as `3*7^6+5`.

A recursive descent parser is a function from an input stream, which is called a *port* in Racket terminology, to a parse derivation, typically an S-expression enriched with source-location information in Racket. To fit into the Chido Parse framework, a parsing function must be wrapped with `proc-parser`. Procedural parsers may dispatch to other parsers using `parse*-direct`, and extract parse results from returned derivations using `d-result`. Parse derivation objects are assembled with the `d` constructor, which accepts a semantic result as well as optional keyword arguments for subderivations or derivation end locations using Racket's `#:keyword` syntax.

```
(define plus
  ;; The functions `proc-parser`, `parse*-direct`, `d`, and functions
  ;; prefixed with `d-` are part of the Chido Parse API.
  (proc-parser
   (λ (in)
     ;; Parse an expression, an operator, and an expression in sequence.
     (define l (parse*-direct in expression))
     (define op (parse*-direct in "+"))
     (define r (parse*-direct in expression))
     ;; The `d` function constructs a derivation, and optionally
     ;; includes sub-derivations with the `#:ds` keyword.
     (d (list 'plus (d-result l) (d-result r)) #:ds (list l op r)))))
```

Note that throughout the chapter, we will insert snippets of Racket code to demonstrate parsing ideas discussed. While the snippets of code use somewhat abbreviated names, the code is otherwise real working code. Because working code often includes details unrelated to the discussion at hand, and to help familiarize the reader with Racket code and the Chido Parse API, we include code comments to help the reader understand the core ideas of each example. Instead of writing parsing functions completely by hand, they can be constructed with combinators. Chido Parse provides combinators such as `sequence`, `repetition`, `alt-parser`, `not-parser` (which succeeds when the parser given to it fails), and `peek-parser` (which parses ahead but returns a derivation that consumes no input).

```
;; The `alt-parser`, `kleene-plus`, and `char-range-parser` functions
;; are provided by Chido Parse.
(define expression (alt-parser plus number))
(define number (kleene-plus (char-range-parser "09")))
```

The above example is left recursive, because `(parse*-direct in expression)` leads back to the `plus` parser via `(alt-parser plus number)`. In a naïve implementation of procedural parsing, this left recursion would cause an infinite loop. By capturing first-class delimited continuations, Chido Parse detects the left-recursive dependency cycle, de-schedules the procedural plus

parser, and schedules the number parser to resolve the cycle. When there is a cycle that can not be broken, Chido Parse detects it and fails the cyclic parser, rather than looping indefinitely.

Even with left recursion handled automatically, the example expression parser has an issue: it implements an ambigious grammar, because the plus operator can be treated as either left- or right-associative. If we add other operators, we will add further ambiguity due to a lack of operator precedence.

Chido Parse allows ambiguous grammars, and its `parse*` function returns a lazy stream of parse results, so the example parser so far is not necessarily wrong. Usually a single result is desirable, however, rather than a parse forest. Worse, ambiguous operator precedence and associativity makes the number of parse results exponential in the number of operator uses! To have an efficient parser that returns a single canonical result, Chido Parse lets a programmer filter ambiguity at its source with disambiguation filter (Klint and Visser 1994) (Brand et al. 2002) combinators such as `derivation-filter`:

```
(define filtered-plus
  ;; The `derivation-filter` function is a combinator from Chido Parse.
  (derivation-filter plus
    ;; Filter to make a right-associative variant of `plus`.
    ;; This checks that the parser used to build the
    ;; leftmost subderivation was not the `plus` parser.
    (λ (d) (not (equal? (d-parser (first (d-subderivations d)))
                        plus)))))
```

For many grammars, it is nice to simply write in a declarative notation like an extended Backus-Naur form. Below is an example use of a BNF-in-S-expressions embedded DSL, where optional `#:keyword` syntax allows declarative specification of precidence and associativity.

```
;; The `define-bnf` macro is provided by Chido Parse.
(define-bnf expression-grammar
  ;; This is an alternative definition of the `expression` parser
  ;; in a BNF-like form encoded in S-expressions.
  [expression [number]
              [expression "+" expression
                          #:associativity 'left]
              ;; Each clause here represents an alternate sequence,
              ;; with optional keyword arguments for disambiguation.
              [expression "*" expression
                          #:associativity 'left
                          #:precedence-greater-than "+"]
              [expression "^" expression
                          #:associativity 'right
                          #:precedence-greater-than "*"]])
```

Several non-procedural parsing systems, such as GLL and GLR, admit left-recursive, disambiguated BNF-style declarations like this. However, they generally can not embed or create parsers for

context-sensitive languages. Some extended systems, such as the extended Earley parser Yakker (Jim et al. 2010) and the extended GLL parser Iguana (Afroozeh and Izmaylova 2015), can include *data-dependent* parsing (Jim et al. 2010). Data-dependent parsing allows sequence parsers to capture intermediate results and use arbitrary computation with those results to filter the overall result of the sequence. With data-dependent parsing, Yakker and Iguana can parse some context-sensitive languages, such as XML. But data-dependent generalized parsing systems such as the above still can not embed arbitrary parsing procedures as grammar productions.

The above `bnf-parser` form is simply a composition of procedural and alternative parsers in Chido Parse, wrapped in a macro to give it a declarative BNF feel. We can extend our `bnf-parser` with arbitrary parsing procedures.

```
(define fancy-expression-grammar
  ;; The `extend-bnf` macro is provided by Chido Parse.
  (extend-bnf
   expression-grammar
   ;; Here we extend the BNF grammar from the previous code snippet
   ;; with three more alternates.
   [expression [expression "/" expression
                           #:associativity 'left
                           #:precedence-greater-than "+"]
               [procedural-complex-number-parser]
               [dynamic-parse-loader]]))

(define procedural-complex-number-parser ...)
(define dynamic-parse-loader ...)
```

Of particular note is the above `dynamic-parse-loader` extension, which can dynamically load parser modules while parsing. The `dynamic-parse-loader` allows `fancy-expression-grammar` to be further extended dynamically within a module that uses `fancy-expression-grammar`. An example of using a dynamically loaded parser is given in Section 2.5.4.

```
(define dynamic-parse-loader
  ;; This is a parser that loads extensions dynamically based
  ;; on the source code being parsed.
  (proc-parser
   (λ (in)
     (define prefix (parse*-direct in "◇"))
     ;; Read a module name as a symbol.
     (define symbol (parse*-direct in symbol-parser))
     (define dynamic-parser
       ;; Dynamically load a parser from the module specified.
       (dynamic-require (syntax->datum (d-result symbol))
                        'dynamic-parser))
     (define open-brace (parse*-direct in "{"))
     (define dynamic-parse (parse*-direct in dynamic-parser))
     (define close-brace (parse*-direct in "}"))
     (d (d-result dynamic-parse)
```

```
#:ds (list prefix symbol open-brace dynamic-parse close-brace)))))
```

2.3 Core Parsing Algorithm

We will describe our algorithm by describing a procedural recursive descent algorithm, then generalizing it.

In our presentation of recursive descent, assume all recursive parsing goes through a `parse` function, which takes a parser object and an input stream, which we will call a `port` using Racket terminology, and returns a parse derivation object. A derivation object is a structure that contains a start position, end position, and a semantic result for the parse. The most basic parser is a string, representing a literal string to be parsed. The next parser object could be a function that takes a port and returns a parse derivation.

```
(λ (in)
  (define start (port-position in))
  (define c (read-char in))
  (d start (+ 1 start) c))
```

Note that our derivation constructor `d` used above and other parsing functions presented here are simplified versions of those used in other sections of this chapter. Additionally, note that strings as parsers are redundant, since parsers for literal strings may be defined as procedures. However, literal string parsers both simplify our presentation and are convenient in practice.

A final parser object is an alternative parser, which stores a list of parsers that could be applied to the port. How should the alternative parser choose which parser to execute? To solve this problem, we modify our notion of procedural parsers. Rather than procedural parsers being merely a function from port to derivation, they will be a structure containing such a function as well as a prefix. The alternative parser can then choose a function based on longest prefix match.

```
(struct alt-parser (parser-list))
(struct proc-parser (prefix-string procedure))
;; The `d` struct will define a `d` constructor that takes
;; three arguments, one for each field.
;; The `d` constructor with optional arguments used
;; in the previous section is an extended wrapper for
;; this constructor.
(struct d (start end result))
```

This interface defines a simple procedural recursive descent parsing system where procedures may recur by calling `parse` with either an alternative or a procedural parser. However, this system displays some classic problems with recursive descent. First, if a procedure has the empty string as

its prefix, it may "left recur" by calling `parse` at its starting position with itself or another parser that is mutually left recursive with it. In a naïve implementation, this dependency loop causes infinite recursion or, more practically, a stack overflow error. There are various proposed methods for solving this left-recursion problem (Frost and Hafiz 2006). However, they all either greatly increase computational complexity or require the parser to be rewritten in a way that obscures the parsing logic. We propose a novel method for handling left recursion by using delimited continuations to de-schedule left-recursive parsers.

2.3.1 Delimited Continuations

Before we describe our modified algorithm, let us review the core operations of delimited continuations: `call-with-continuation-prompt`, `abort-current-continuation`, `call-with-composable-continuation`, and application of captured continuations (illustrated in figure 2.2).

The `call-with-continuation-prompt` (`call/prompt`) function takes a thunk to run, a prompt tag, and an abort handler function. It then installs a prompt (with the supplied handler) in the current continuation and executes the supplied thunk.

The `abort-current-continuation` (`abort/cc`) function takes a prompt tag and a value to pass to the prompt's abort handler. It then aborts the current continuation up to the prompt, replacing that section of the continuation with an expression that applies the prompt's handler to the given value.

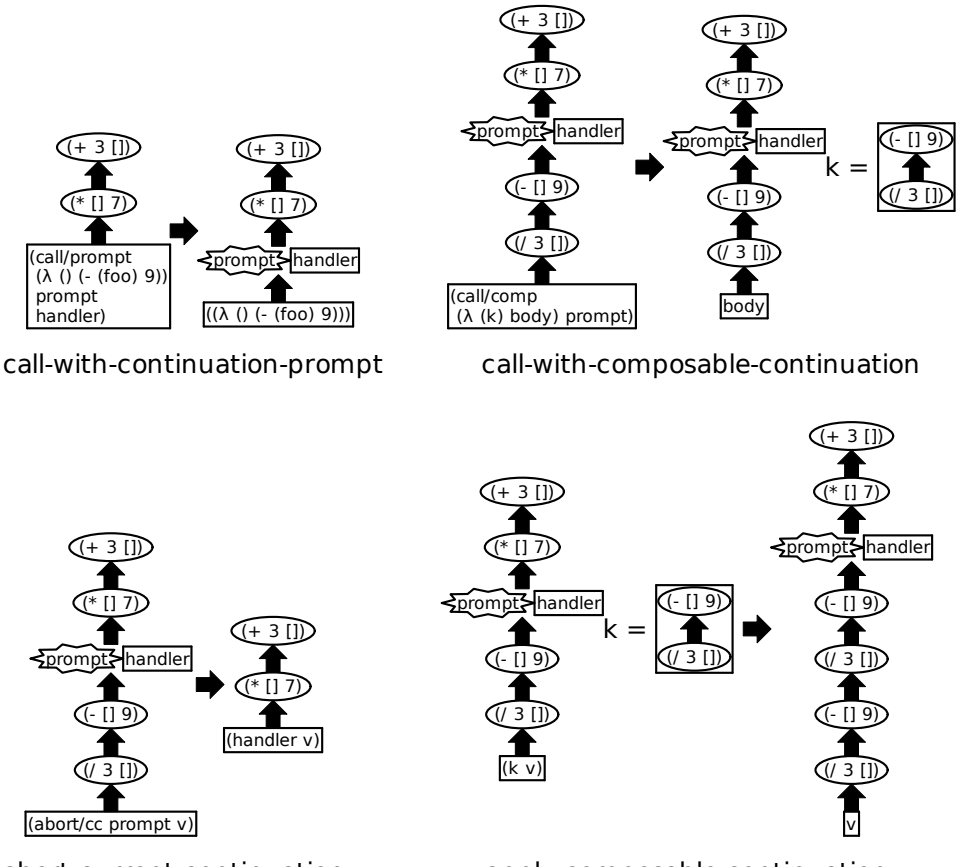
The `call-with-composable-continuation` (`call/comp`) function takes a function of one argument, in this case `k`, and a prompt tag. It captures the current continuation up to the prompt with the matching tag, and executes the given function, passing the continuation in as `k`. When applying a captured composable continuation `k`, the captured continuation frames do not replace the current continuation stack, but rather are added to the end of the stack. In other words, the captured continuation `k` behaves like a normal function.

2.3.2 Parsing With Delimited Continuations

To solve the left-recursion problem with delimited continuations, the `parse` function must install a delimited continuation prompt around a call to a parsing procedure. When a recursive call is made, the `parse` function can de-schedule the parent parser by capturing and aborting its continuation delimited by the previous prompt. Using this method, we can view each call to `parse` as a *job* that can be descheduled to resolve cyclic dependencies. When jobs are descheduled, the

```
⟨exp⟩   ::= ⟨exp⟩   +   ⟨exp⟩
          |  ⟨exp⟩   *   ⟨exp⟩
          |  ⟨exp⟩   /   ⟨exp⟩
          |  ⟨exp⟩   ^   ⟨exp⟩
          |  ⟨nat⟩
⟨nat⟩   ::= ⟨digit⟩+
⟨digit⟩ ::= 0-9
```

Figure 2.1: Arithmetic Grammar



In these diagrams, ovals represent continuation frames and [] represents the "hole" in a continuation where a result will go. Continuation stacks grow down.

Figure 2.2: Delimited Continuation Operations

continuations are stored in job records along with the job's dependency. When a dependency cycle can not be resolved, rather than looping indefinitely, the scheduler can simply return a parse failure for the job that caused the cycle. While we will continue to explain the algorithm in prose here, those desiring further clarity may wish to consult the appendix, which includes a model implementation as a literate program.

Scheduling parse jobs begins with an original call into a `parse` function, and scheduling is performed greedily to satisfy that root parse call. Each time a recursive call to one of Chido Parse's parsing functions is made, the current job is de-scheduled and a dependency is created linking the de-scheduled job to the job for the recursive call.

Each parsing job contains dependency information, so we can traverse a graph of dependencies. In particular, we can trace a path from the root job to any other parsing job. When a job *J* is de-scheduled with a dependency on a job *K* that is already on the dependency path from the root job to *J*, it indicates a left-recursive cycle.

When a left-recursive cycle is found between jobs *J* and *K*, it can potentially be resolved by scheduling a different choice from any alternate parsers on the path between *J* and *K* (inclusive of *J* and *K*). If all choices from alternates between *J* and *K* are stuck in cycles (or if there is no alternate between *J* and *K*), then the cycle is broken by returning a parse failure result to job *J*. This situation arises, for example, in an infix arithmetic parser when each infix operator alternate is waiting for a parse of the left expression at a position when no non-infix expression can be parsed, or when a series of successively larger infix expression parses finally matches the entire expression and can not find any new ambiguous parse trees to be on the left-hand side.

The next problems with our system stem from ambiguity. Consider the following expression grammar:

```
(define expression (alt-parser (list number plus)))
(define number (proc-parser "" (λ (in) #|parse a number...|#)))
(define plus
  (proc-parser "" (λ (in)
                    (define start (port-position in))
                    ;; Parse expression, op, expression in sequence.
                    (define l (parse in expression))
                    (define op (parse in "+"))
                    (define r (parse in expression))
                    (d start (d-end r)
                       (list 'plus (d-result l) (d-result r))))))
```

The intent of this parser is for the `expression` parser to return multiple results, first for a number,

then potentially for an addition expression where the first number is its left argument. Additionally, the grammar leaves the associativity of the plus operator ambiguous, meaning that there could be several valid parse derivations for a series of plus operations. However, our current `parse` API returns just one derivation per call! To fix this, we can instead define `parse*`, which is like `parse` but returns a lazy *stream* of results, where getting the next value in the stream forces computation back into the parsing system. This allows alternative parsers to return results from each of its parsers, and even multiple results from each parser. Additionally, for ease of programming, we allow procedural parsers to return a stream tree of results, which we (lazily) flatten into a single stream to be returned from `parse*`. It should be noted that forcing computation from a stream of parse results may trigger a left-recursive computation, so it is important that the stream implementation wrap its computation with a continuation prompt as a parsing job.

We can now define a new version of plus that handles ambiguity:

```
(define plus
  (proc-parser
   ""
   (λ (in)
     (define start (port-position in))
     (for/stream ([l (parse* in expression start)])
       ;; For each ambiguous derivation from the previous parser,
       ;; use the next parser at the end point of the derivation.
       (for/stream ([op (parse* in "+" (d-end l))])
         (for/stream ([r (parse* in expression (d-end op))])
           (parse-derivation start (d-end r)
                            (list 'plus (d-result l) (d-result r)))))))))
```

Note that the implementation is much more complicated due to ambiguity. Rather than working with a single result from each `parse*`, we must account for the possibility of multiple results using the `for/stream` form, which lazily loops over a stream, with the body returning once per stream element. Since multiple derivations returned by `parse*` may have different lengths, we need to add a third argument for the start location to the `parse*` function. Additionally, we need a rewindable version of our input port object.

To return our implementation of plus to its former simplicity, we introduce `parse*-direct`, which leverages delimited continuations to implicitly loop over a stream. The `parse*-direct` function also has an associated continuation prompt that is installed each time a job is run, after the job prompt. When `parse*-direct` is called, it captures its continuation up to its prompt, then loops over `parse*`'s result stream by calling the continuation, wrapped in another `parse*-direct` prompt, inside `for/stream`.

```
(define (parse*-direct in parser position)
  ;; `call/comp` captures the continuation (without replacing it).
  (call/comp (λ (k) (abort/cc parse*-direct-prompt
                              ;; `abort/cc` replaces the continuation with this thunk.
                              (λ () (for/stream ([deriv (parse* in parser position)])
                                      (k deriv)))))
             parse*-direct-prompt))
```

By extending `parse*-direct` further to set the port location within the `for/stream` loop, we can additionally make the start position argument optional. This allows us to write ambiguous, left-recursive, procedural parsers in a direct style:

```
(define plus
  (proc-parser ""
               (λ (in)
                 (define start (port-position in))
                 ;; Parse in series, like before, but implicitly
                 ;; looping for each ambiguous result.
                 (define l (parse*-direct in expression))
                 (define op (parse*-direct in "+"))
                 (define r (parse*-direct in expression))
                 (d start (d-end r)
                    (list 'plus (d-result l) (d-result r))))))
```

To aid in understanding, figure 2.3 provides an overview of the Chido Parse API.

Ultimately, delimited continuations solve two problems: the left-recursion problem and the problem of complicated, loop-filled code to handle ambiguity. Users of a parsing system that uses this algorithm must know that their parsing code may run multiple times in an unspecified order, and thus program functionally to avoid race conditions. Additionally, users should filter parse ambiguity as close as possible to its source if they want to avoid exponential complexity, and must understand the basics of the stream API. However, users do not need to understand the delimited continuation API or the low-level details of the parsing algorithm to write parsers against it.

With the foundation of `proc-parser` and `alt-parser`, higher-level layers may be built and composed. Parser combinators and disambiguation filters are straightforward to implement as functions that construct `proc-parser`s, and declarative DSL interfaces like extended BNF can also be constructed to desugar to calls to combinators and filters.

2.4 Applications

We have applied the Chido Parse library to a several practical problems in the Racket ecosystem. While every feature of Chido Parse is not leveraged by every example, supporting the suite of examples is not possible without support for arbitrary left-recursive procedural parsers, dynamic

extensibility, context sensitivity, and composability.

2.4.1 Readtables and S-Expressions

Much of the motivation of this parsing project was to improve the status quo for creating parsers for DSLs in Racket. Many Racket languages use a readtable, which is a dispatch table for parsers that are written separately and then composed together. However, readtable-based composition is limited and brittle.

A readtable has a table mapping of single-character prefixes to parsing procedures. When a readtable is used to parse, the first character of the input is matched against the table. If a mapping is found, the mapped procedure is applied. If no mapping is found, a built-in symbol parser is applied. Prefix characters in the table have not only a parsing procedure, but also a flag to indicate whether symbol parsing should stop at that character. The extra flag for symbol parsing frees users from needing to craft an appropriate identifer parser when extending a readtable.

Setting the `current-readtable` parameter installs a modified readtable, and by convention each parser in a readtable should use `current-readtable` for recursive parsing. This convention allows each parsing procedure to access a customized readtable to apply or further extend for its own recursive calls without needing to know anything about the other extensions in the readtable. Readtable extension also lends itself to Racket's "meta-language" protocol that allows users to compose parser extensions. For example, the `at-exp` meta-language extends the `current-readtable` with the Scribble documentation notation (prefixed with the `@` character). So, a user may write a module in `#lang at-exp racket` to use the Racket language extended with Scribble notation. In happy cases where extension prefixes do not conflict, multiple such meta-languages may be composed together.

The single-character prefix of readtables, however, is very limiting to Racket DSL developers. For example, a readtable extension can not define a dollar-prefix parenthesis grouping `$( ... )` that behaves different from normal parenthesis grouping `( ... )` without blocking other extensions that begin with the dollar sign or making the dollar sign unusable as a character in symbols. The limitation is so great that Racket's readtable includes a one-off hack, making the hash (#) character be the prefix of a second `dispatch` table. While the extra `dispatch` mode allows limited use of two-character prefixes, there is a clear desire for longer prefixes. In fact, Racket's default prefix for regular expression literals is `#rx`, and Racket's default prefix for dynamic parser module

loading is `#reader`. Because they both begin with the prefix `#r`, the `#rx` and `#reader` parsers must be built together as a single procedure that dispatches on the extended prefixes, erroring when none of the known prefixes match. Adding another alternate that begins with the prefix `#r` cannot be done compositionally.

Thus, while readtables are useful for building parsers for certain languages, they disallow local ambiguity, left-recursion, infix operators, and postfix operators. Languages that require these features must be built using different parsing tools and can not share extensions or use the same meta-language protocol as readtable languages.

Using the Chido Parse library, we have implemented a readtable-like parser constructor, called *chido-readtables*. Like Racket's readtables, chido-readtables include a default symbol parser that can be automatically affected by extensions. Chido-readtables additionally use a similar convention of referring to a dynamic `current-chido-readtable` parameter so that extensions can refer to a fully customized table without information about the customizations. However, chido-readtables allow arbitrary length prefixes, parsers with no static prefix, left-recursive parsers, and mixfix operators with declarative associativity and precedence.

The following code creates three increasingly customized readtables. To create `rt1`, the current readtable is accessed and extended with the "at-reader," a parser for the context-sensitive Scribble (Flatt et al. 2009) notation. This extension automatically makes the built-in symbol reader stop when encountering `@` characters, disallowing them from symbols. This first extension matches the capabilities of Racket's core readtable interface.

```
(define rt1
  ;; extend to support Scribble @-reader
  (extend-chido-readtable (make-at-reader-parser #:prefix "@")
                          (current-chido-readtable)))

;; read `$(a b c)` as `(#%dollar-paren a b c)`
(define rt2 (chido-readtable-add-list-parser "$(" ")" rt1 #:wrapper '#%dollar-paren))

;; read `a $+$ b $+$ c` as `($+$ ($+$ a b) c)`
(define rt3 (chido-readtable-add-mixfix-operator "_$+$_" rt2 #:associativity 'left))
```

However, the extensions applied to build `rt2` and `rt3` are possible only with the more powerful Chido Parse readtable interface. The `rt2` readtable is extended with a new list parser that begins with the multi-character string `"$("`. While this extension automatically disallows symbols from containing the string `"$("`, it does not disallow dollar signs in symbols generally, or prevent other extensions prefixed with the dollar sign. In fact, `rt3` extends `rt2` with the infix operator `$+$`. Using

`rt3`, users may embed Scribble documentation in their code, and code escapes inside the Scribble documentation can still include `$()` lists and infix `$+$` operations. Chido-readtable extensions may always be added compositionally, although some compositions may ultimately create parsers for ambiguous languages.

The chido-readtable abstraction is useful especially for extended s-expression languages, and because Racket programmers are already comfortable using the similar readtable abstraction. Because they are simply Chido Parse parsers, chido-readtables are easily composable with other Chido Parse parsers, giving Racket language designers extra flexibility to design and compose different language components each with the interface that makes the most sense for that component.

2.4.2 Declarative Parsers

Grammars are often written in a declarative fashion using notations such as Backus-Naur Form (BNF). These notations are popular because they are machine readable but also easy for humans to read and reason about. However, it is often the case that a grammar written for humans will include a simplified, ambiguous BNF grammar along with side-conditions that disambiguate the grammar. Many parsing systems only allow basic BNF grammars, or even a limited subset of them due to a lack of support for all context-free grammars. Some parsing systems such as SDF3 (Brand et al. 2002) support BNF with extensions that allow machine-readable specification of disambiguating side conditions like operator precidence. A few systems such as Iguana (Afroozeh and Izmaylova 2015) support a BNF format extended to handle extra binding and computation clauses to implement context-sensitive features such as off-side indentation rules in Haskell and Python, or tag matching in XML.

Chido Parse allows users to write arbitrary procedures and consult arbitrary data structures to perform parsing, to compose parsers, and to build parsing abstractions. In particular, this flexibility allows users to build parser constructors that read BNF or other formats. We have included with Chido Parse an extended BNF DSL that supports regular right-hand sides, capturing intermediate results for data-dependent parsing, referencing procedural parsers as BNF productions, and declarative specification of operator precedence and associativity disambiguation. Our initial BNF is implemented as an s-expression-based DSL as demonstrated in Section 2.3. However, we have also used that DSL in s-expression notation to construct a parser to enable a more traditional BNF notation.

Here is an excerpt of an XML parser written using our declarative BNF-style DSL and notation. Extra annotations, including `%`, `@`, and `/`, specify details about how the semantic parse result is constructed, such as whether to include the name of the alternate in the result, where to splice list elements, and whether to omit elements. Because XML requires opening and closing tags to match, it is not a context-free language. While this XML parser is mostly declarative, it includes an escape to use a disambiguation combinator to perform data-dependent filtering of parse results. Specifically, it captures the result of the starting tag `STag` into a variable, then references that variable to ensure that the ending tag `ETag` matches.

```
#lang chido-parse/bnf-syntactic
...
% element : @EmptyElemTag
          | open = @STag
            ;; The Start Tag is captured as the variable `open`.
            @content
            ;; Parse an End Tag, but filter to only tags that
            ;; match the Start Tag.
            /$(derivation-filter
               ETag
               (λ (close) (equal? (derivation->tag-name open)
                                  (derivation->tag-name close))))
...
```

2.4.3 Composition

In today's polyglot programming landscape, many programmers want to (and do) mix multiple languages in a single file. This is particularly visible in web programming, where there is a profusion of HTML templating languages, JavaScript and CSS embedded in HTML, and bits of HTML and SQL embedded in strings of front-end and back-end languages.

However, language embedding and composition are often rife with problems. Embedding and interpolating bits of language syntax in strings for later interpretation commonly suffers from security problems, in addition to irritating string escaping. Constructing a parser that can statically parse the mixed languages often involves writing a complex, monolithic parser that encapsulates the grammars of all mixed languages.

With Chido Parse we can write language parsers in a modular fashion and easily compose disparate parsers. Resulting parsers allow a mixed-language file to be parsed in one pass and are not vulnerable to injection attacks. For example, the following example combines our XML parser with our readtable-based S-expression parser into a parser for ParenHP, a web preprocessor language inspired by PHP (PHP: Hypertext Preprocessor 2020):

```
(define parenhp-s-exp
  ;; hash-tag-parser uses parenhp-parser to parse tags
  (extend-chido-readtable hash-tag-parser basic-readtable))
(define parenhp-parser
  (extend-bnf xml-parser
              [element ["<¿" parenhp-s-exp "¿>" #:result/stx (λ (l m r) m)]]))
```

The composite parser returns racket syntax objects where the XML portions represent expressions that generate racket *x-exprs*, an s-expression encoding of XML data used by Racket's web server modules. Using this parser and some macro definitions, a user may produce a quick dynamic web page that displays the client's IP address and query parameters with the following code:

```
#lang chido-parse/demo/parenhp
<html>
  <h1> Hello to <¿(request-client-ip (current-request))¿>  </h1>
  <p>Query parameters:
    <¿(append #<ul/>
              (for/list ([b (request-bindings (current-request))])
                #<li><¿(car b)¿> is <¿(cdr b)¿></li>))¿>
  </p>
</html>
```

2.4.4 Parser Profusion and Dynamic Extension

While we can create a composite parser in the manner of ParenHP by using a parsing module that composes a set of parsers we care about, there are potentially very many interesting language combinations. For example, a user might want to embed SQL queries, shell pipelines, logic programming relations, user-defined data literals, and more, each with its own syntax. It would be impractical to create such a module for each member of the (ever growing) superset of DSL notation parsers. It is more tenable to allow users to create ad hoc composite languages with dynamic extension loading and composable meta-languages.

One approach to notation mixing in macro-extensible languages like Racket is to embed DSL code in strings and parse them during macro expansion. This approach is optionally used by some Racket-based DSLs such as Rash (Hatch and Flatt 2018).

```
(require rash)
(for/list ([x (list "earth" "mars")])
  ;; Use the `echo` unix program, pipe its output to
  ;; the Racket `string-append` function.
  (rash "echo hello |>> string-append _ x"))
```

This delayed-string approach does not suffer from common issues with run-time code string interpolation, because the strings are parsed as part of macro expansion during compilation, and the approach can be used with any parsing technology. However, this approach is not well composable

with common "quasiquoting" macros, such as Racket's `syntax` form used to instantiate syntax templates. In the following example, the `syntax-parse` form matches the syntax object for the string `"world"` as the pattern variable `x`. The `syntax` form on the right-hand side eagerly substitutes all instances of `x` in its template with the syntax for `"world"`. But because the `x` on the right hand side is inside a string that has not yet been parsed, the substitution will be missed, resulting in an unbound variable error—or, worse, an unintentional capture of a different variable named `x`.

```
(require rash syntax/parse)
(syntax-parse (syntax "world")
  [x (syntax (rash "echo hello |>> string-append _ x"))])
```

By dynamically loading parser extensions in a procedural parser like Chido Parse, embedded notations can be parsed together in one parsing phase before macro expansion, sidestepping issues with delayed string parsing. In the below example, we assume a module parser that includes a parser like the `dynamic-parse-loader` defined in Section 2.3. The module parser does not include any built-in support for parsing Rash code, but it can be dynamically included using the `dynamic-parse-loader`. In this manner, the module parser can provide dynamic, extended language support without delaying any parsing to expansion time.

```
(require rash syntax/parse)
(syntax-parse (syntax "world")
  [x (syntax ◇rash-parser{
       echo hello |>> string-append _ x
      })])
```

While the features of the applications above can be implemented in isolation with other parsing systems, Chido Parse enables these and other applications to be implemented and composed all together in a single system.

2.5 Evaluation

We evaluate Chido Parse by considering aspects of its utility for parsing, including expressiveness, static analyzability, failure reporting, algorithmic complexity, and performance.

2.5.1 Expressiveness

The main goal of designing Chido Parse was to create a maximally expressive and flexible parsing system. In particular, we had the goal of supporting several specific features required to support all the applications listed in Section 2.5:

- Support for constructing and composing parsers using any mix of ad hoc procedural parsers, declarative formats like BNF, and parser combinators.

- Support for dynamic parser extensibility.

- Support for ambiguous and left-recursive grammars.

- Support for the entire class of context-free grammars.

- Support for context-sensitive grammars, including but not limited to data-dependent grammars (Jim et al. 2010).

As shown in figure 2.4, Chido Parse is the first system to accomplish all of these goals.

While figure 2.4 demonstrates Chido Parse's expressiveness, the rest of the evaluation section focuses on aspects of parsing where Chido Parse has room for improvement.

2.5.2 Static Analysis

For various applications it is important to statically analyze a parser. Chido Parse allows the embedding of arbitrary parsing procedures, and even dynamic extensibility, and is therefore not well suited to these static analyses. Other, more limited parsing systems can afford much simpler and more effective static analyses for optimization or making guarantees about a grammar.

2.5.3 Ambiguous Failure Messages

Generalized parsing systems that allow ambiguity, including Chido Parse, can follow several paths to multiple successful parse trees. However, when a parse is unsuccessful, multiple paths lead to multiple failure locations and messages. These ambiguous failure messages make it much harder to design clear and precise failure reporting. This is an active research topic, and we leave an optimal solution to future work.

2.5.4 Computational Complexity

Some generalized parsing algorithms can achieve a cubic worst-case complexity by using a compact parse representation like SPPF (Tomita 1985) in which interior ambiguity is represented by shared nodes. Because our procedural parsers can inspect intermediate results to make branching decisions, we can not rely on a method for compacting ambiguous internal nodes to reduce complex-

ity, allowing ambiguous parsers to exhibit exponential worst-case complexity. To demonstrate this complexity, we performed an experiment parsing the following ambiguous expression grammar:

⟨*exp*⟩ ::= ⟨*exp*⟩ + ⟨*exp*⟩
| x

Because the grammar does not specify an associativity for the + operator, the number of parse trees for an expression is exponential in the number of + instances. We benchmarked parsing on expressions with different numbers of + instances, as shown in Figure 2.5 and Figure 2.6. Figure 2.5 shows that the time to parse the first result of the ambiguous stream seems nearly linear, but Figure 2.6 shows that the time to process the entire stream of results is exponential. Additionally, since procedural parsers may dynamically and arbitrarily construct and apply new parsers and do any computation, our system may exhibit super-exponential complexity or may not terminate.

However, in the common case where disambiguation filters are used and exponential computation is not performed in parsing procedures, our system in practice enjoys near-linear complexity. To demonstrate, we generated random s-expressions, printed them with Racket's pretty printer, and parsed them with our s-expression parser. Figure 2.7 shows parsing time by number of characters in the textual s-expression representation. Figure 2.8 shows the same parse times compared to the number of elements in the parsed tree representation. Both figures show time to exhaust the parse stream, which in each case only contains one element.

All benchmarks were run single-threaded on an AMD FX-8350 processor, with 24GB of RAM, using Nixos version 21.05 and Racket version 8.3cs.

2.5.5 Performance

Chido Parse is very expressive, but achieves expressiveness with some performace cost. While computational complexity is near-linear for language parsers we have implemented, our current implementation in Racket is not sufficiently performant to provide drop-in replacements for common parsing tasks. Our current implementation of a flexible, extensible s-expression parser runs about 100 times slower than Racket's default s-expression parser using Racket 8.0. This speed may be acceptable for certain domain-specific languages where file sizes are small and flexibility in concrete syntax is of the utmost importance, but not for the majority of Racket languages and programs. However, we do not believe there is an inherent obstacle preventing significant perfor-

mance optimizations. With approximately one order of magnitude improvement, it would cease to be the slowest component in the compilation of common Racket modules and would become a viable, practical tool for many DSL and extensible language projects.

2.6 Related Work

We compare Chido Parse to related parsing systems. In particular, we discuss left recursion, context-free grammars, parser combinators, parser extensibility, and delimited continuations.

2.6.1 Left Recursion for Procedural Parsers

Various methods have been proposed to support left recursion in top-down procedural parsers, including several listed by Frost and Hafiz (2006). Most of these methods either obscure the resulting parser, fail to support some grammars, or result in exponential complexity.

Warth et al. (2008) describe a method for Packrat parsers to handle left recursion. This method starts by seeding the Packrat memo tables with an initial failure result, causing left-recursive parsers to fail. After an alternative parser succeeds, the modified Packrat parser re-runs the left-recursive parser with the new results, called "growing the seed." Additionally, the modified Packrat parser keeps track of the stack of parsers to identify indirect left recursion and rerun parsers appropriately. This modified version of the Packrat algorithm does not preserve the linear-time guarantee of the original Packrat algorithm, but it increases the algorithm's supported set of grammars to a superset of PEG (Ford 2004) grammars. However, the extension does not add support for ambiguous grammars.

Frost et al. (2007) describe a method to support left recursion by passing an extra parameter when recurring to count the number of recursions. When the number of recursive calls to a given procedure exceeds the remaining input length, recursion is curtailed. While this method is applicable to any style of top-down parsing, it requires knowledge of the input length and can increase algorithmic complexity by a large factor. In its original presentation in a memoized combinator system without arbitrary parsing procedures, the system achieves an overall worst-case complexity of $O(n^4)$. However, if arbitrary procedures are allowed, a naive implementation of this method of left-recursion handling increases complexity by a factor of the grammar size raised to the power of the input length, since each left-recursive alternate could run at each recursion depth.

Another method to achieve left recursion with procedural parsers is by explicit transformation

to continuation-passing style (CPS) and memoization (Johnson 1995). The Meerkat (Izmaylova et al. 2016) parsing system provides a parser combinator system that implements the memoized CPS strategy together with the GLL parsing algorithm. Users could add custom parsing procedures to Meerkat by extending its undocumented AbstractParser class. However, custom parsers in Meerkat and other systems using this method for left recursion must be manually converted to continuation-passing style, which is very cumbersome and obfuscates the resulting code.

Our method for handling left recursion can be used without prior knowledge of the input length and only recurs into procedures that can make progress. While our overall parsing system exhibits super-exponential worst-case performance, it is not due to the method of handling left-recursion via delimited continuations. Our method achieves similar complexity to memoized explicit continuation-passing style, but because our method captures continuations automatically, custom parsing procedures may be written in a direct style without manual CPS conversion.

2.6.2 Context-Free Grammars and Beyond

There are various generalized algorithms that support the entirety of the class of context-free grammars, including GLL (Scott and Johnstone 2010), GLR (Tomita 1985), and Earley (Earley 1970) parsing. These algorithms are limited to context-free grammars, but support all context-free grammars, including grammars with ambiguity and left-recursion. These algorithms admit grammars in a form amenable to analysis and optimization. Unlike proper subsets of context-free grammars such as LR and LL, the full set of context-free grammars is closed under union, concatenation, and repetition, allowing the grammars supported by these generalized systems to be composed with ease.

There are extensions to some of these algorithms that extend their power. The Syntax Definition Formalism SDF3 uses an extended SGLR parser that allows some context-sensitive languages, including indentation-sensitive languages that use the off-side rule, to be parsed with declarative rules (Erdweg et al. 2012) (Amorim et al. 2018). Yakker (Jim et al. 2010) is an extension of the Earley algorithm that allows intermediate results to be captured and used in computation of side conditions for terminal and nonterminal productions. Additionally, Yakker supports *blackbox* procedural parsers. Iguana is an extended GLL that similarly supports data-dependent parsing. While computation in side-conditions allow Yakker and Iguana parsers to support context-sensitive grammars, they do not allow arbitrary procedural parsing of the input stream. Yakker's blackbox

parsers allow limited procedural parsing, but blackbox parsers may not recur back into the Yakker parsing system.

Chido Parse supports context-sensitive grammars by allowing arbitrary procedural parsing, including predicate side-conditions. Procedural parsing also enables parse-time grammar extension by using dynamic module loading. However, full static analysis of procedural parsers is undecidable, so a procedural parsing system may not enjoy the same static optimizations that can be employed in more limited systems.

2.6.3 Parser Combinators

Parser combinator systems, such as Parsec (Leijen and Meijer 2001), allow grammars to be built by composing parser objects. Primitive parser objects in combinator systems may limited to a particular set of parser or grammar objects, such as with Binsbergen et al. (2018), or the primitive parser objects may be arbitrary parsing procedures (Wadler 1985). When arbitrary procedures are allowed, these systems generally either execute infinite recursion when given a left-recursive grammar or have exponential computational complexity in both the worst case and common case for left-recursive grammars.

A noteworthy point in the design space is described by Danielsson (2010). In this system, parsing procedures may be composed and proved to terminate, and some forms of left-recursion are allowed. However, the system is limited to procedures and parsers that can be proven to terminate within the dependently-typed logic of its implementation language.

Our system implements combinators that accept arbitrary parsing procedures while automatically handling left-recursion by capturing delimited continuations. While our system has worst-case super-exponential complexity, it enjoys near-linear complexity in common practical cases.

2.6.4 Language Workbenches and Extensible Grammars

Language workbenches and related systems such as SDF (Visser 1997), Metaborg (Bravenboer and Visser 2004), Spoofax (Kats and Visser 2010), Rascal (Klint et al. 2009), and Racket (Flatt and PLT 2010) rely on extensible parsing systems to provide re-usability and composability of language components. Language workbenches are limited in the languages they can effectively support by the power and expressiveness of the parsing systems they use. Empowering language workbenches to support a broader set of languages while also supporting more flexible ways of specifying parsers for those languages is the primary goal of Chido Parse.

Many advances in powerful and expressive parsing systems have been designed for use in language workbenches, such as SGLR (Visser 1999), Iguana (Afroozeh and Izmaylova 2015), and Meerkat (Izmaylova et al. 2016). These and similar parsing systems allow a direct encoding of ambiguous patterns such as infix operators with annotations to encode disambiguation filters. Generalized systems like GLR admit the entire class of context-free grammars, allowing grammars to be composed. SGLR further improves composability by removing a dependence on separate, non-composable lexical analysis stages. While the original formulations of GLL, GLR and SGLR are restricted to context-free grammars, Meerkat and Iguana improve parsing power by extending beyond the set of context-free grammars by introducing data-dependent parsing.

However, these systems do not support other context-sensitive grammars beyond context-free or data-dependent grammars. Also, these systems do not allow writing parsers in a mixture of BNF, combinator, and procedural styles, and generally do not support dynamic parser extension. In addition to supporting context-free and data-dependent grammars, Chido Parse supports other context-sensitive grammars, allows any mixture of ad hoc procedural parsers, combinators, and BNF specifications to be composed, and allows dynamic parser extensibility.

2.6.5 Delimited Continuations in Parsing

Kiselyov (2009) shows that delimited continuations can be used to make a parser incremental and restartable. Chido Parse does not use delimited continuations to provide incrementality, but rather to solve the left-recursion problem.

2.7 Conclusion

We have demonstrated various limitations of the expressiveness and convenience of extant parsing technology. We have presented a novel parsing algorithm that combines the benefits of GLL and ad hoc procedural parsing to overcome those limitations, as well as a novel method for handling left-recursion in procedural parsers that enables this combination. Our system allows users to write and compose parsers, including for ambiguous, left-recursive, and context-sensitive grammars, and allows parsers to be written and composed using declarative abstractions like BNF, using parser combinators, or in a direct, procedural style. We have given examples that demonstrate how this algorithm solves problems in creating extensible and composable languages.

2.7.1 References

Michael D. Adams, Celeste Hollenbeck, and Matthew Might. On the Complexity and Performance of Parsing with Derivatives. In *Proc. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016. http://dx.doi.org/10.1145/2908080.2908128

Ali Afroozeh and Anastasia Izmaylova. One parser to rule them all. In *Proc. ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015. https://doi.org/10.1145/2814228.2814242

Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-Printing Layout-Sensitive Languages. In *Proc. 11th ACM SIGPLAN International Conference on Software Language Engineering*, pp. 3–15, 2018.

L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. GLL Parsing with Flexible Combinators. In *Proc. Conference on Software Language Engineering (SLE)*, 2018.

M.G.J. van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. *Journal of Functional Programming - JFP*, pp. 143–158, 2002.

Martin Bravenboer and Eelco Visser. Concrete syntax for objects. In *Proc. 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004. https://doi.org/10.1145/1035292.1029007

Nils Anders Danielsson. Total parser combinators. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2010. https://doi.org/10.1145/1863543.1863585

Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. ACM Conference on Lisp and Functional Programming*, pp. 151–160, 1990.

Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM* 13(2), pp. 94–102, 1970.

Sebastian Erdweg, Tillmann Rendel, and Klaus Ostermann. Layout-sensitive generalized parsing. In *Proc. Conference on Software Language Engineering (SLE), volume 7745 of LNCS*, pp. 244–263, 2012.

Matthias Felleisen. The theory and practice of first-class prompts. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 180–190, 1988.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: closing the book on ad hoc documentation tools. In *Proc. SIGPLAN International Conference on Functional Programming*, 2009. https://doi.org/10.1145/1596550.1596569

Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010. https://racket-lang.org/tr1/

Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proc. 2004 ACM Symposium on Principles of Programming Languages (POPL)*, 2004.

Richard Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser Combinators for Ambiguous Left-Recursive Grammars. In *Proc. 10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pp. 167–181, 2008.

Richard Frost, Rahmatullah Hafiz, and Paul C. Callaghan. Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars. In *Proc. 10th Conference on Parsing Technologies*, pp. 109–120, 2007.

Richard A. Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *ACM SIGPLAN Notices* 41(5), pp. 46–54, 2006. https://doi.org/10.1145/1149982.1149988

William Gallard Hatch and Matthew Flatt. Rash: From Reckless Interactions to Reliable Programs. In *Proc. 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2018. https://doi.org/10.1145/3278122.3278129

Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, General Parser Combinators. In *Proc. Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, 2016.

Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and Algorithms for Data-dependent Grammars. In *Proc. Principles of Programming Languages (POPL)*, 2010. https://doi.org/10.1145/1706299.1706347

Mark Johnson. Memoization of Top Down Parsing. *Computational Linguistics*, 1995.

Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

Oleg Kiselyov. Differentiating Parsers. 2009. http://okmij.org/ftp/continuations/differentiating-parsers.html

Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2009. https://doi.org/10.1109/SCAM.2009.28

Paul Klint and Eelco Visser. Using Filters for the Disambiguation of Context-free Grammars. In *Proc. ASMICS Workshop on Parsing Theory*, pp. 1–20, 1994.

Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Departement of Computer Science, Universiteit Utrecht, UU-CS-2001-27, 2001. https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/

Daniel D. McCracken and Edwin D. Reilly. Backus-Naur Form (BNF). In *Encyclopedia of Computer Science*, pp. 129–131 John Wiley and Sons Ltd., 2003.

Matthew Might, David Darais, and Daniel Spiewak. Parsing with Derivatives: A Functional Pearl. In *Proc. 16th ACM SIGPLAN International Conference on Functional Programming*, 2011. https://doi.org/10.1145/2034574.2034801

PHP: Hypertext Preprocessor. 2020. https://www.php.net/

Elizabeth Scott and Adrian Johnstone. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253(7), pp. 177–189, 2010. https://doi.org/10.1016/j.entcs.2010.08.041

Dorai Sitaram. Handling Control. In *Proc. Programming Language Design and Implementation*, 1993.

Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

Eelco Visser. Syntax definition for language prototyping. PhD dissertation, University of Amsterdam, 1997.

Eelco Visser. Scannerless generalized-LR parsing. Programming Research Group, University of Amsterdam, P9709, 1999.

Phillip Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp. 113–128, 1985.

Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat Parsers Can Support Left Recursion. In *Proc. 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 2008. https://doi.org/10.1145/1328408.1328424

Daniel H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control* 10(2), pp. 189–208, 1967.

- `parse*` :

  Contract: `input-port? parser? [optional (or/c int? parse-derivation?)] -> (stream-of parse-derivation?)`

  Note: The normal empty stream at the end of a derivation stream may be replaced by a parse-failure object

- `parse` :

  Contract: `input-port? parser? [optional (or/c int? parse-derivation?)] -> (or/c parse-derivation? parse-failure?)`

  Note: If `parse*` would have returned a stream of more than one result, a failure is returned.

- `parse*-direct` :

  Contract: `input-port? parser? [optional (or/c int? parse-derivation?)] -> (or/c parse-derivation? parse-failure?)`

  Note: While a single result is returned to the continuation where `parse*-direct` is used, the continuation may be invoked multiple times. Effectively, use of `parse*-direct` turns the outer parser procedure into a loop instead of requiring the user to write a loop.

In the contract notation used above, `or/c` indicates that any of the included options are valid. The notation `[optional contract]` used in procedure contracts indicates an optional argument.

Figure 2.3: Overview of Chido Parse API

	Left recursion	Direct procedural left-recursion	Arbitrary procedural parsers	Data-dependent context sensitivity	Ambiguous grammars	Full CFG support	Composable	Extensible dynamically	Parser combinator interface	BNF interface	Purely declarative †1	Worst-case computational complexity
Chido Parse	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		Exponential †2
GLL	✓				✓	✓	✓		✓	✓	✓	Cubic
GLR	✓				✓	✓	✓			✓	✓	Cubic
Earley	✓				✓	✓	✓			✓	✓	Cubic
CYK (Younger 1967)	✓				✓	✓	✓				✓	Cubic
Parsing With Derivatives †6	✓				✓	✓	✓		✓	✓	✓	Cubic
Yakker	✓		†3	✓	✓	✓	✓			✓		Cubic if limited to CFG †2
Iguana	✓			✓	✓	✓	✓			✓		Cubic
Meerkat	✓		†4	✓	✓	✓	✓	†4	✓	✓		Cubic †2
Packrat	†5						✓		✓		✓	Linear
Parsec			✓	✓			✓	✓	✓			†2
Frost et al. 2008	✓				✓	✓	✓		✓	✓	✓	n^4
Procedural Recursive Descent			✓	✓			✓	✓	✓			†2
LR	✓									✓	✓	Linear

Rows for algorithms imply the basic version of the algorithm, while rows for implementations generally imply an extended version of one of the algorithms.

†1 By "Purely declarative", we mean that a system lacks a Turing complete component, allowing more opportunities for static analysis.

†2 Parsing systems that admit arbitrary procedures may exhibit arbitrarily large complexity or non-termination based on the procedures used.

†3 Yakker (Jim et al. 2010) allows "blackbox" parsing procedures to be used as nonterminals, but they can't recur back into the Yakker system.

†4 The Meerkat (Izmaylova et al. 2016) implementation supports custom procedural parsers in an undocumented way, but not in the public interface.

†5 The original formulation of PEG (Ford 2004) and Packrat (Ford 2002) parsers does not support left-recursion, but (Warth et al. 2008) describes a memoization technique for Packrat parsers to support left-recursion. This modification increases the computational complexity of the parser, however.

†6 (Might et al. 2011) (Adams et al. 2016)

Figure 2.4: Comparison of Parsing Algorithms and Frameworks

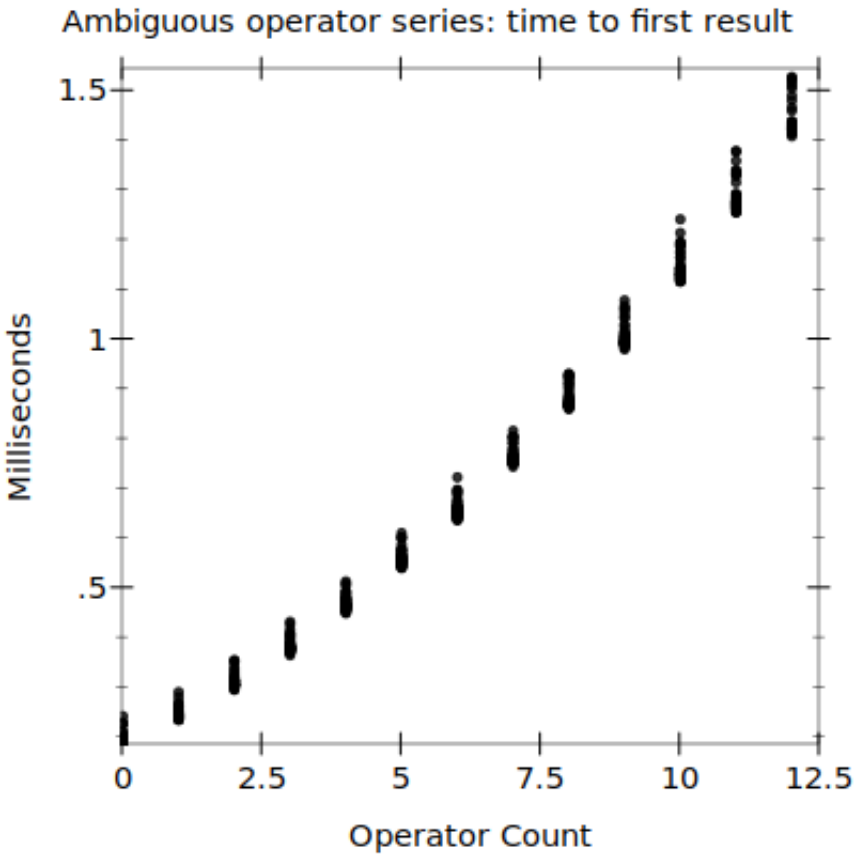


Figure 2.5: Ambiguous operator parsing, time to first result

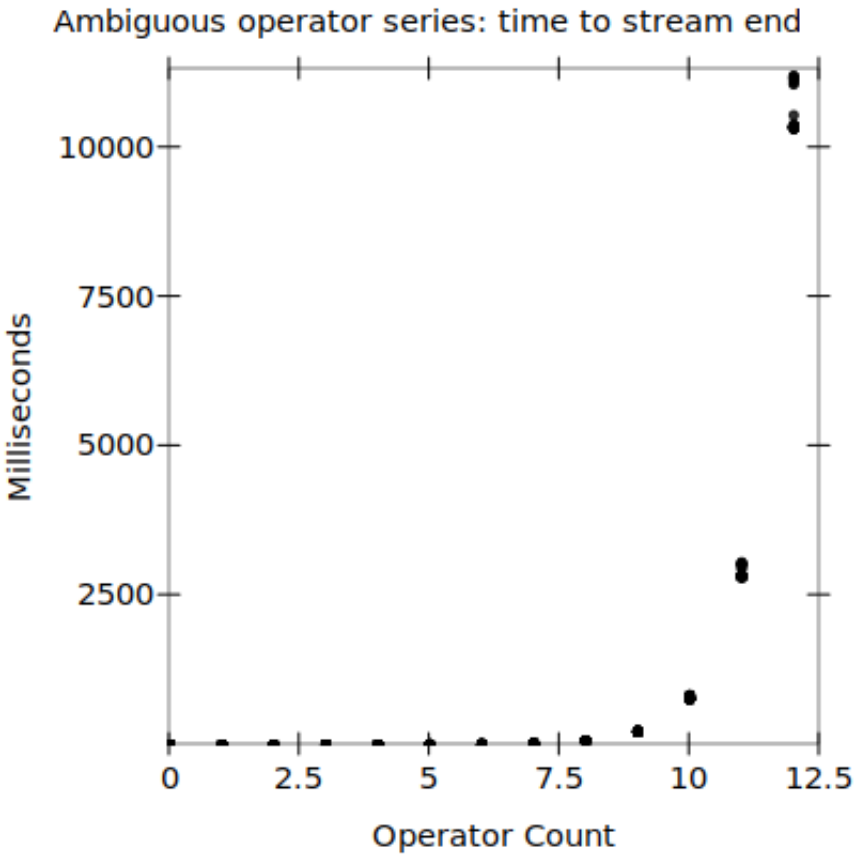


Figure 2.6: Ambiguous operator parsing, time to stream end

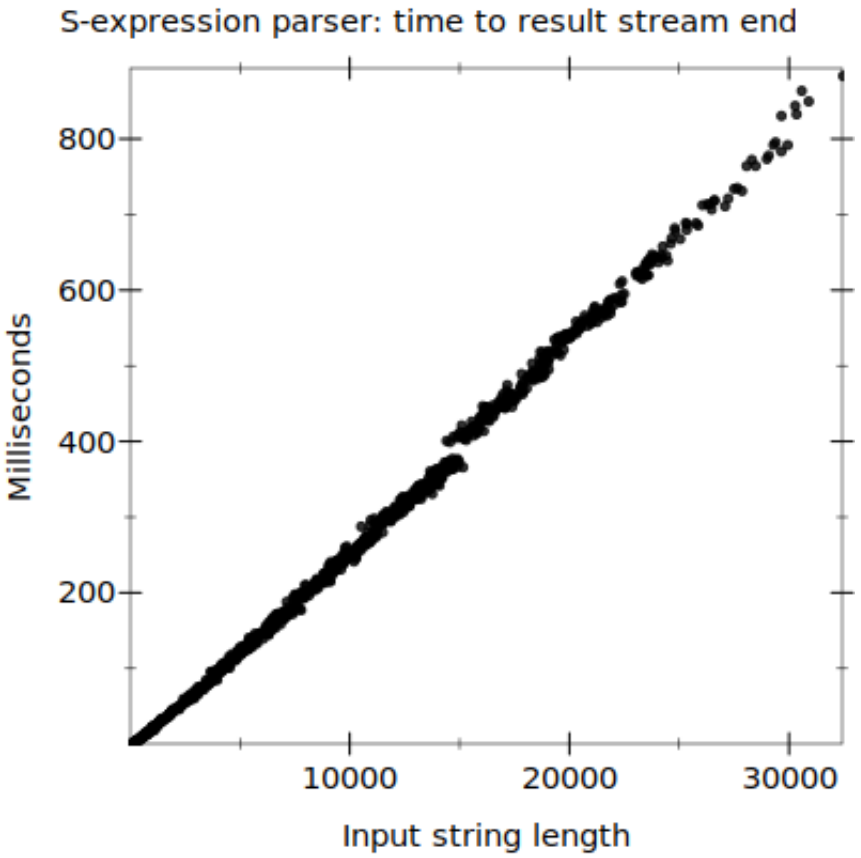


Figure 2.7: S-expression parsing by string length

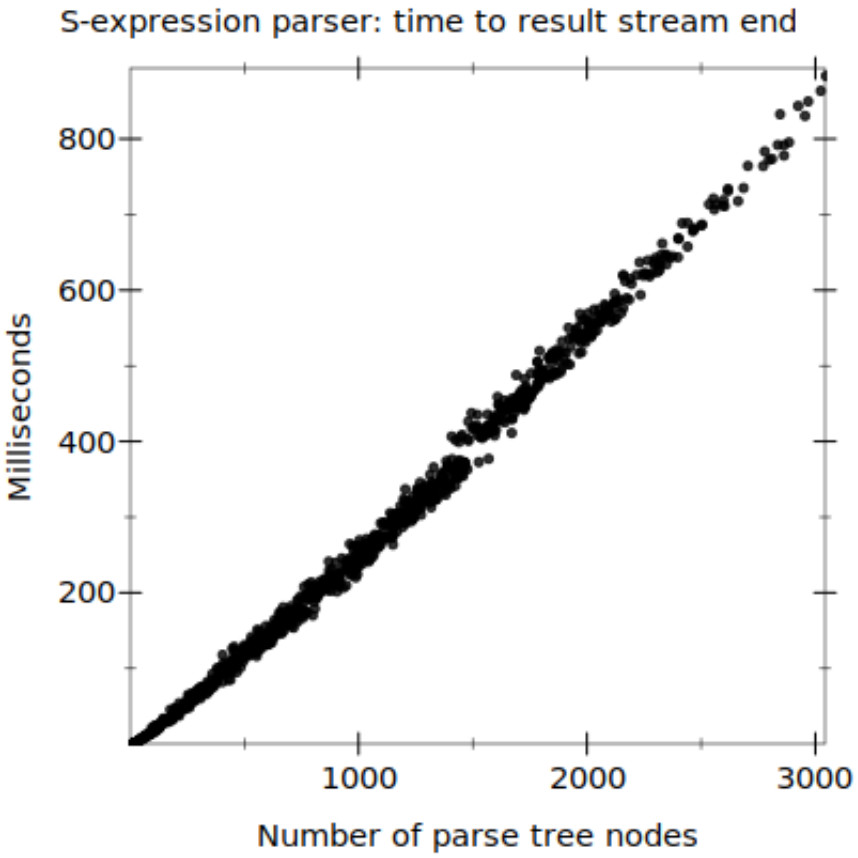


Figure 2.8: S-expression parsing by parsed tree size

CHAPTER 3

RASH: FROM RECKLESS INTERACTIONS TO RELIABLE PROGRAMS

Embedded DSLs offer the possibility of both easier implementation, due to re-use of host language features, and of greater expressiveness, due to integration with the host language and other DSLs embedded into the same host. However, for an embedded DSL to maximize its expressive power compared to a stand-alone DSL, it requires careful design to accommodate tight integration between the DSL and its host.

This chapter discusses Rash, a command shell language embedded in Racket. It provides an example of design for tight integration of EDSLs. Specifically, it is designed by decomposing the notational and semantic requirements, and designing both the notation and semantics of the DSL to integrate tightly, at the expression level, with the host language. This design allows fine-grained, recursive nesting of the shell DSL and the host. Additionally, Rash provides an example of exposing and extending the host language features that allow an embedded language to be built by providing access to Racket's macro system and specific new families of macros within the embedded language. This enables new DSLs to be embedded even within the DSL, such as an example `make`-like language embedded in Rash.

Rash captures the features of command languages like the Bourne Shell that make it useful for interactive and exploratory programming, and for automation via copying interactions into scripts. However, unlike common shell languages, Rash scales beyond small scripts because Rash programs can have an arbitrary mix of shell-like code and general-purpose Racket code. Thus Rash provides a gradual scale between shell-style interactions and general-purpose programming.

This chapter is a reprint of Hatch and Flatt (2018).

3.1 Impulsive Introduction

Programmers often write prototypes, quick solutions, or exploratory programs, then later edit or rewrite them to move them along a spectrum of program maturity and scale. Moving code along

this scale is often viewed as a transition from "scripts" to more mature "programs," and current research aims to improve that transition, especially through gradual typing (Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006). In this paper, we address a point in the spectrum that precedes even the "script" level of maturity: command sequences in an interactive shell.

Different features and aspects of programming languages are well suited to different stages of program maturity. For example, static types are clearly useful for ensuring and maintaining software correctness, but types are often seen as burdensome or obstructive when writing scripts, so many scripting languages eschew types. Programmers want brevity and even less formality in interactive settings, so `read-eval-print` loops (REPL) often have relaxed rules on object mutability and introspection, and Unix shells offer especially terse notations.

Tailoring a language to a specific point in the spectrum has obvious advantages, but serving different points in the spectrum through wholly distinct languages creates new problems. Developers may be forced to choose between maintaining a program in a language that is no longer suited to the program's evolution, or rewriting the program in a new language. We should instead make languages adapt and interoperate along the maturity spectrum, providing a smooth path from one end to the other. Gradual typing systems like TypeScript (Microsoft 2018), Reticulated Python (Vitousek et al. 2014), and Typed Racket (Tobin-Hochstadt et al. 2008) are the most prominent efforts toward this alternative, but they start at the "scripts" point in the spectrum.

To support graduality between shell-style interactions and general-purpose programs, a language must be general-purpose while also supporting domain-specific features of a shell, such as process and file manipulation. Additionally, there is tension between optimizations for programmatic and interactive settings, such as optimizing notation for variables or literal data, and optimizing for legibility or terseness. The key challenge of graduality between interactions and programs is to support a seamless mixture of general-purpose and command notation.

Rash is a command language embedded in the general-purpose Racket language.[1] Rash supports lightweight syntax and programming patterns similar to popular command languages, like Bash and PowerShell, but Rash and normal Racket code can be embedded within each other at the expression level. Rash also interoperates with other Racket-based languages, including Typed Racket. With these features, Rash affords easy and gradual movement along the spectrum from interactions to

[1] http://rash-lang.org

scripts to programs, and it allows programmers to combine modules that inhabit different stages of the maturity spectrum.

Rash is organized into two subsystems which embody the contributions of this paper. The first subsystem is Linea, which provides a line-oriented syntax suitable for terse interactions. Linea maintains interoperability with other Racket dialects by leveraging Lisp's traditional separation of *read* and *macro expansion* phases. Linea's reader accepts a mixture of line-oriented and S-expression notation and produces output suitable for macro expansion. Linea also introduces *line macros*, a customization in the macro-expansion phase allowing programmers to add user-defined keywords and block semantics. The second subsystem is a domain specific language (DSL) for pipelining, which generalizes Unix-style pipelines to support arbitrary objects in addition to byte streams, similar to PowerShell's (Snover 2002) pipelines. Both subsystems are libraries that can be used on their own to provide part of the convenience of a shell language within conventional Racket programs, but their benefits synergize to form the Rash language.

3.2 Impetuous Overview

A programmer reading Rash code should understand four main points:

- Rash is primarily line-based. Lines within a program or block are evaluated in series from top to bottom.
- The meanings of lines are determined by *line macros*.
- The default line macro is `run-pipeline`, which implements the pipelining DSL. Pipelines may be composed of subprocesses that communicate using byte streams or Racket functions that communicate using arbitrary Racket objects.
- Despite being primarily line based, users can embed S-expressions and blocks of lines within each other by using parentheses `()` and braces `{}`, respectively.

To clarify these points, as well as to demonstrate the benefits of embedding a command language inside a general-purpose language, we discuss an example REPL session. Suppose that Alyssa P. Hacker wants to look at some of her spending habits over the last year. Each month, Alyssa's bank gives her a CSV file detailing her debit card purchases. Alyssa stores this file as ∼`/records/`*year*`/`*month*`/purchases.csv` in her computer, with *year* and *month* substituted appropriately. She starts a Rash session and types:

```
cd records
ls 2017/*/* | grep purchases
```

Many of Rash's cues for the syntax of commands and the inclusion of command pipelines come directly from existing shell languages such as Bourne shell (Bourne 1978). The above commands work as a Bourne shell user would expect: First the `cd` line changes the current working directory, then the `ls` and `grep` pipeline is executed. Output from the `ls` subprocess specified on the left of the `|` operator becomes the input to the `grep` subprocess specified on its right, and the pipeline prints a listing of Alyssa's `purchases.csv` files.

Alyssa made various purchases at Computer Store throughout the year, and would like to see a summary of them. She types:

```
in-dir 2017/* {
  echo summary of (current-directory)
  grep -i "computer store" purchases.csv
}
```

to see a quick report for Computer Store purchases every month. This example shows several Rash features:

- Parentheses in a command line escape to Racket. In the expression `echo summary of (current-directory)`, one of the arguments to the `echo` subprocess is computed by the Racket function `current-directory`.

- The `in-dir` identifier is a line macro. Line macros are keywords that, when placed at the start of the line, determine the meaning of that line. The `in-dir` line macro performs glob expansion on its first argument and executes its second argument once for each directory matched. It also parameterizes each execution of the second argument so that `(current-directory)` returns the matched directory.

- Braces read a block of code in line mode. Braces can be used in line mode as this example shows, or they can be used inside parenthesised S-expressions to escape back to line mode. Braces implicitly act like Racket's `begin` form, which evaluates its sub-forms, in this case, lines, sequentially.

- While logical lines in Rash are usually the same as physical lines, there are ways to combine multiple physical lines into one logical line. Newline characters are treated as normal whitespace if they are escaped by putting a backslash in front of them, if they are inside a multiline

comment, or if they are inside parentheses. If newlines are inside a curly brace block, they delimit the lines of an embedded line-mode context, all of which is part of one logical line in the outer context. The example above takes advantage of the behavior of braces to give the `in-dir` line macro a multi-line body.

- The `echo` and `grep` lines in the loop body aren't obviously using a line macro like `in-dir`, but every line that doesn't explicitly begin with a line macro has a default inserted. While users can configure different default line macros for different lexical regions of code, the `run-pipeline` line macro is used as the default throughout this paper (with the exception of Section 3.4.2). The `echo` and `grep` lines shown above are technically pipelines, although they are degenerate pipelines of only one command each.

Alyssa then wants to know how much she spent in total in December. She runs:

```
cat 2017/12/purchases.csv |> csv-file->dicts \
  |> map (λ (t) (hash-ref t "amount")) \
  |> map (λ (n) (string->number n)) |> apply +
```

Rash's pipeline DSL supports two types of pipeline segments: subprocess segments, which communicate using byte streams, and function segments, which communicate using Racket objects. The `|>` operator used above sends the result of the previous pipeline segment to a Racket function. The `|>` operator builds its function by using all the forms to its right until the next pipeline operator, using the _ identifier as the name of the argument it gets from the previous pipeline segment. If the _ identifier is not explicitly present in one of the argument forms, the `|>` operator appends it to the end of the list. So `|> a b` will use the function `(λ (x) (a b x))` while `|> a _ b` will use the function `(λ (x) (a x b))`.

When a Racket function segment follows a subprocess segment, such as `csv-file->dicts` following `cat`, the subprocess output stream is passed to the function as a Racket port object, which is Racket's encapsulation of a byte stream. If a subprocess segment follows a function segment, the return value of the function is printed to the stdin of the subprocess. To prevent blocking, all adjacent subprocess pipeline segments, as well as the first function segment following them, if any, are run in parallel, while function segments are executed from left to right sequentially.

The above pipeline uses the function `csv-file->dicts` to parse the CSV contents and return a list of one hash table per purchase, using the fields of the CSV header line as keys. The pipeline then uses more functions to extract and sum the dollar amounts of the purchases.

It is somewhat implausible that someone would frequently type examples like the previous one in an interactive shell. Instead of the above example, Alyssa can also get the same result by running this simplified version:

```
|> csv-file->dicts "2017/12/purchases.csv" \
  =map= hash-ref _ "amount" \
  =map= string->number |> apply +
```

Pipeline operators are user-definable macros, so users can create new operators that simplify the notation for their commonly used patterns. For example, users may define operators for mapping over or filtering lists or other data structures, operators for chaining object method or monad function calls, or operators that determine their behavior based on context, such as an operator that behaves like `|>` when its first argument is bound as a Racket function and otherwise behaving like `|`. In the example above, Alyssa uses a custom `=map=` pipeline operator which simplifies and flattens the common pattern of mapping over a list. The `=map=` operator, like the `|>` operator, automatically places the _ identifier if it is not explicitly written, but it uses the _ identifier as the argument for iterations of the map loop rather than the entire list received from the previous pipeline segment.

In this example, the `|>` operator is used at the beginning of the pipeline as a prefix operator. All pipelines start with a prefix operator, but a default is inserted automatically when none is specified explicitly. The default prefix operator throughout this paper is the `|` operator, which specifies a subprocess pipeline segment, but it can be customized for different lexical regions of code. The `|>` operator has no pipeline argument to pass to the `csv-file->dicts` function when it is in prefix position, so the _ argument is not inserted, and explicit use of it would raise a compilation error. Here, the `csv-file->dicts` function is given a literal name of a file to open rather than receiving a port from a previous pipeline stage.

Alyssa decides she wants to save the results of some computations to variables. She types:

```
(define n-hardware-purchases
  (string->number
   (with-rash-config
      #:out (compose string-trim port->string)
      {grep -i "computer store" 2017/*/purchases.csv \
        | wc -l})))

def month-list in-dir 2017/* {
  |> csv-file->dicts "purchases.csv"
}
```

This example highlights more language features:

- If a line starts with an open parenthesis, line macro insertion is skipped, and the line is read as a normal Racket form.

- At the top level of the REPL and of modules written in `#lang rash`, pipeline input and output are connected to Racket's `current-input-port` and `current-output-port`, which are generally connected to `stdin` and `stdout`. Sections of Rash code can be wrapped with the `with-rash-config` macro, which accepts optional arguments to parameterize behaviors for the code region, including the default line macro, the default pipeline input and output ports, and the default prefix operator. Instead of a port, the output can be set to a function that accepts a port to convert subprocess output into Racket objects.

- Rash lines are expressions that can return values. The `in-dir` line macro returns a list of results from the executions of its body. The `def` identifier is a line-macro version of `define` that is modified to better support line macros within a definition.

Use of the `with-rash-config` macro above is unwieldy, and the definition of `n-hardware-purchases` in this case can be shortened to:

```
(define n-hardware-purchases
  (string->number
   #{grep -i "computer store" 2017/*/purchases.csv \
      | wc -l}))
```

The `#{}` form implicitly sets subprocess input to an empty port, converts subprocess output to a string, and trims trailing whitespace from it.

Alyssa is curious how much her spending varies from month to month. She runs:

```
(require math/statistics)
(stddev (for/list ([m month-list])
          {|> values m =map= hash-ref _ "amount" \
            =map= string->number |> apply +}))
```

The `(require math/statistics)` form makes the `stddev` function available. Rash users can `require` modules written in any Racket dialect, such as `#lang typed/racket`, `#lang lazy`, and, of course, `#lang rash`. Users can seamlessly switch between using S-expressions and line-oriented code, or between using subprocesses or functions and macros from their favorite Racket libraries.

As Alyssa runs all of these commands, she copies some of her favorites into a file. She puts the line `#lang rash` at the top of the file to signify that they are in the Rash dialect of Racket. As long

as she does not skip interactions that create intermediate definitions that are used later, a verbatim copy of Alyssa's interactions makes a working program.

When Alyssa is finished running commands and copying the relevant ones to her script file, she has a program for summarizing her finances, but it summarizes only her 2017 finances. To make her script more useful, she must make her script more general by making changes like replacing literals with variables and adding error-handling code. She may also extend her script over time with new features, such as creating graphs about her financial data. As she generalizes and extends her program, she has access to all of Racket, such as the `racket/cmdline` DSL to specify command-line parsing to set initial variable values, and the `plot` library for generating graphical plots.

This introductory example of Alyssa analyzing banking data demonstrates the strengths of Rash's "interactions to scripts" approach. At the beginning, Alyssa explored the filesystem and examined files in her banking directory using programs such as `cat` and `grep`. These operations are convenient in traditional shells like Bash (Free Software Foundation 2018b), but they are cumbersome in scripting languages like Python or Racket. Later Alyssa analyzed CSV data by parsing it with a Racket function, then mapping and filtering over the data with more Racket functions. These operations passed structured lists of objects and used mathematical functions like `stddev`, and they are easy to accomplish in general-purpose languages like Python or Racket. Operating over structured data, in contrast, is very difficult in shell languages like Bash without access to programs that can do the work, each one deserializing the data and then serializing it again in a format recognized by the next program. In the common case, structured data in shell scripts is handled with ad-hoc, error-prone parsing of lines of text. Traditional shell languages and general-purpose languages each handle half of the problem well, but the other half poorly. By embedding a shell language within a general-purpose language, Rash fits both aspects of the problem well.

3.3 Incautious Examples

To further demonstrate how embedding a shell language within a general-purpose language supports a broader part of the interactions-to-programs spectrum, we present the following examples.

3.3.1 Error Handling

During an interactive session a user manually executes most control flow and decides how to handle errors after seeing them. Once interactions are turned into scripts, control flow and error

handling need to be explicitly added. Since they are not part of the domain of interactive command execution, shell languages often have only rudimentary control-flow and error-handling support. By embedding a command language within a general-purpose language, the command language can inherit fully featured control and error handling forms.

As an example of a common type of error handling that can be done in Rash scripts, consider a script that uses the `pdflatex` command to generate pdf files. The `pdflatex` command generates various intermediate files that clutter the file system, so our example runs `pdflatex` in a temporary directory. We do not want to leave temporary files, so we want to remove the directory whether pdf generation is successful or not.

```
(define here (current-directory))
mkdir my-tmp-dir
try {
  in-dir my-tmp-dir {
    pdflatex $here/example.tex &< /dev/null
    mv example.pdf $here/
  }
} catch err {
  (raise err)
} finally {
  rm -rf my-tmp-dir
}
```

Since cleaning up a temporary directory even in the presence of errors is a common problem in shell scripts, it may be convenient to simplify the pattern with a custom line macro. The `in-tmp-dir` line macro used below encapsulates directory creation and removal as well as the `try` and `in-dir` line macros used above.

```
(define here (current-directory))
in-tmp-dir {
  pdflatex $here/example.tex &< /dev/null
  mv example.pdf $here/
}
```

3.3.2 Make

Building software is another case where a program — specifically, a build script — needs to accomplish something that parallels a programmer's interactive commands. Often this is done using the `make` program, which accepts *makefiles* written in a language that specifies build targets, their dependencies, and their build recipes. The recipes in makefiles are written in shell language, but the targets and dependencies can be specified using only a limited language specific to the `make` program. We have written a `make` replacement that allows Rash and ordinary Racket code in recipe

bodies as well as in target and dependency lists.

Here is an example in our `make` language that builds a "hello" program with a version-specific file name and makes a symbolic link to it with the generic "hello" name:

```
#lang rash/demo/make
(define version #{cat version.txt})

hello : (string-append "hello-" version) {
  ln -sf (current-dependencies) hello
}

hello-$version : hello.c version.txt {
  gcc -o (current-target) hello.c
}
```

The `#lang` line sets the reader and available identifiers of the file to those provided by our `rash/demo/make` module. The `rash/demo/make` module configures the file to have the same reader as normal Rash modules and makes all the same identifiers available. However, the `rash/demo/make` module provides extra definitions, including the `current-dependencies` and `current-target` functions, as well as the `make-target` line macro. The `rash/demo/make` language sets `make-target` as the default line macro at the top level of the module. The `make-target` line macro accepts lists of target and dependency files as well as a a recipe body to be executed when a target is built. The targets and dependencies can be listed literally, computed with string interpolation using dollar escapes, or computed with normal Racket code by escaping with parentheses. The `make-target` line macro sets `run-pipeline` to be the default line macro within its body and parameterizes the `current-target` and `current-dependencies` to return its target and dependency lists appropriately. The `rash/demo/make` language also automatically inserts code to handle command line arguments and build any specified targets.

Complicated target names, dependency names, or build recipes, such as dependencies whose names must be computed dynamically, can benefit from a full, general-purpose programming language. Meanwhile, straightforward build instructions benefit from being written as closely as possible to how the programmer writes them interactively.

3.4 Temerarious Lines

Lisp systems break up the process of turning program text into executable code into distinct passes, including *read*, *macro expand*, and *compile*. By separating reading and expansion stages, Lisps allow programmers to create macros to extend the language while reasoning about trees and

identifiers rather than at the level of byte stream parsing. The macro expander can also manage scope automatically during expansion, which is crucial to making macro-based extensions composable. While the separation of reading and expanding has primarily been used by S-expression-based languages, some languages with algebraic syntax like Honu (Rafkind and Flatt 2012) have also found it effective.

The *read* stage plays a role similar to the lexer in traditional languages. It transforms sequences of characters into numbers, symbols, string, etc. Lisp readers differ from traditional lexers, however, in that rather than producing a flat sequence of tokens, they produce a tree by grouping elements between parentheses. Linea extends the *read* stage with a reader function that transforms a line-oriented concrete syntax into a syntax tree.

The *expand* stage walks over the tree produced by the *read* pass. As it traverses the tree, the expander detects macro use-sites and invokes the code bound to each macro. Macros transform sub-trees, and can be seen as mini compilers, since they successively compile macro-enriched language variants into simpler languages until arriving at a core language. This core language can finally be interpreted or compiled to machine code. Linea extends the *expand* stage with the notion of line macros, which allow users to extend or replace the semantics of lines or blocks of code.

3.4.1 Reading Lines

As the reader layer for Rash, Linea was designed primarily with two goals: allowing convenient live interactions, and mixing seamlessly with normal Racket code. To enable users to simply type commands like `ls /etc`, line breaks are used to determine basic grouping. To enable seamless mixing, parentheses switch the reader to (mostly) traditional S-expression reading while braces switch the reader back into line mode.

While users can escape to Racket by using parentheses within a line, it is sometimes convenient to bypass the line reader and its line macro insertion to use plain Racket with S-expressions. Linea detects when lines start with an open parenthesis, and it uses only the traditional S-expression reader on them instead of grouping based on newlines. To allow the macro expander to distinguish between groupings derived by lines or by top-level S-expressions, all forms are implicitly wrapped with an extra symbol. Lines are prefixed with the `#%linea-line` symbol, and top-level S-expressions are wrapped with the `#%linea-s-exp` symbol. Implicit `#%` symbols are discussed further in section Section 3.5.2.

Top-level S-expression detection leads to an idiosyncrasy. A user may want the first argument of a line macro to be a parenthesised form, such as:

```
run-pipeline (if use-llvm? 'clang 'gcc) program.c
```

If this example is written without the explicit `run-pipeline` line macro name, it becomes

```
(if use-llvm? 'clang 'gcc) program.c
```

which starts with a parenthesis and thus would trigger top-level S-expression reading instead of line reading.

While it may seem that Linea without the top-level S-expression escape would be a more consistent system, and the reader can be configured to behave as such if desired, the escape embodies an important principle in Rash. Linea with the escape enabled is a near superset of S-expressions and Linea without the escape. The parts removed in this combination, specifically top-level non-parenthesised forms from pure S-expressions and `#%linea-line`s that start with a parenthesised form in pure Linea, are not frequently used. Allowing the common cases of both notations together greatly increases convenience, while requiring the uncommon cases to be written in a more round-about way is only little inconvenience. The interactive focus of Rash and Linea drive design to be maximized for convenience over conceptual simplicity, and adding the top-level S-expression escape is a trade-off similar to the added complexity of grouping by line and then by parentheses instead of just by parentheses.

Because Linea with top-level S-expression detection is a near-superset of S-expressions, most programs written in `#lang racket/base` can be switched to `#lang rash` without changing meanings. With this design line-based interactions can be recorded and saved to a script or inserted into an existing program, and then can be changed gradually as desired into the S-expression format more commonly used for general Racket programming features.

3.4.2 Linea Grammar

The grammar of Linea is as follows:

⟨*linea-program*⟩ ::= ⟨*nl*⟩* ⟨*linea-form*⟩+ [⟨*last-linea-line*⟩]
⟨*linea-form*⟩ ::= ⟨*linea-line*⟩
| ⟨*linea-s-exp*⟩
⟨*linea-s-exp*⟩ ::= ⟨*paren-form*⟩
⟨*linea-line*⟩ ::= ⟨*line-mode-form*⟩+ ⟨*nl*⟩+
⟨*last-linea-line*⟩ ::= ⟨*line-mode-form*⟩+
⟨*paren-form*⟩ ::= (⟨*nl*⟩* ⟨*s-exp-mode-form*⟩*)
| (⟨*nl*⟩* ⟨*s-exp-mode-form*⟩* ⟨*nl*⟩*
. ⟨*nl*⟩* ⟨*s-exp-mode-form*⟩ ⟨*nl*⟩*)
⟨*brace-form*⟩ ::= { ⟨*nl*⟩* ⟨*linea-form*⟩* [⟨*last-linea-line*⟩] }

⟨*hash-brace-form*⟩ ::= `#{` ⟨*nl*⟩* ⟨*linea-form*⟩* [⟨*last-linea-line*⟩] `}`
⟨*line-mode-form*⟩ ::= ⟨*line-mode-number*⟩
| ⟨*line-mode-symbol*⟩
| ⟨*string*⟩
| ⟨*paren-form*⟩
| ⟨*brace-form*⟩
| ⟨*hash-brace-form*⟩
| ⟨*hash-form*⟩
⟨*s-exp-mode-form*⟩ ::= ⟨*number*⟩ ⟨*nl*⟩*
| ⟨*symbol*⟩ ⟨*nl*⟩*
| ⟨*string*⟩ ⟨*nl*⟩*
| ⟨*paren-form*⟩ ⟨*nl*⟩*
| ⟨*brace-form*⟩ ⟨*nl*⟩*
| ⟨*hash-brace-form*⟩ ⟨*nl*⟩*
| ⟨*hash-form*⟩ ⟨*nl*⟩*

The following points clarify the grammar:

- Non-newline whitespace is implicitly allowed between nonterminals and repetitions in the grammar. Since newlines are different from other whitespace in Linea, however, the grammar makes explicit where newlines are allowed and required with ⟨*nl*⟩.

- There are differences between ⟨*line-mode-symbol*⟩ and ⟨*line-mode-number*⟩ in line-mode and ⟨*symbol*⟩ and ⟨*number*⟩ in S-expression mode. For instance, `-i` is a ⟨*line-mode-symbol*⟩, but as an ⟨*s-exp-mode-form*⟩ it is a ⟨*number*⟩, specifically the complex number `0-1i`. Additionally, the period character produces a symbol in line mode but is treated specially in S-expression mode to produce an improper list.

- The ⟨*paren-form*⟩ can also be substituted for a ⟨*bracket-form*⟩, which reads identically to the ⟨*paren-form*⟩ except that parentheses `()` are replaced with square brackets `[]`.

- ⟨*hash-form*⟩s are various forms prefixed with the `#` character, and include boolean literals `#t` and `#f`. The ⟨*hash-form*⟩s are taken directly from Racket's reader, and they are primarily for literal data such as hash tables and structs. The one exception is ⟨*hash-brace-form*⟩, which Rash treats differently than Racket.

Reading a program produces a list much like reading normal S-expressions, but a few implicit symbols are added by the reader function.

- Each list produced by reading a ⟨*linea-line*⟩ is prefixed with `#%linea-line`.

- Each list produced by reading a ⟨*linea-s-exp*⟩ is wrapped in another list prefixed with `#%linea-s-exp`, such that the result is `(#%linea-s-exp` ⟨*list-as-read*⟩`)`.

- Each list produced by a ⟨*brace-form*⟩ is prefixed with `#%linea-expressions-begin`.

- The ⟨*hash-brace-form*⟩ produces the same result as a ⟨*brace-form*⟩, except that it is wrapped in a list prefixed with `#%hash-braces`.

- When a Racket file is read, it must produce a module form with the name of a module to import identifier from, a name for the new module, and the code of the module within a `#%module-begin` form.

As an example, following Rash program:

```
#lang rash
echo (+ 1
        #{(+ 2 (* 3 4))})
echo goodbye
```

assuming it is in a file named `bye.rkt`, produces the same result that a traditional S-expression reader would produce if given this:

```
(module rash bye
  (#%module-begin
    (#%linea-line
     echo
     (+ 1
        (#%hash-braces
         (#%linea-expressions-begin
          (#%linea-s-exp (+ 2 (* 3 4)))))))
    (#%linea-line echo goodbye)))
```

Racket has a convention of prefixing identifiers with `#%` when they are added implicitly, including `#%app` and `#%datum`, which the macro expander implicitly adds to every function application or literal data form it encounters, respectively. These identifiers are used as extensibility hooks for language implementors to change the semantics of implicit forms for modules in their languages. Linea provides `#%` identifiers as extension hooks to follow the convention.

Similar to S-expressions, the Linea reader does not provide any initial meaning for code aside from determining the tree shape. Just as S-expressions can be used to encode languages with many different execution models (such as Typed Racket and the logic programming language Parenlog), Linea syntax can be used to encode many different line-oriented languages. However, the Linea package provides default meanings for its implicit identifiers for use in languages like Rash, which are described in Section 3.5.4.

3.4.3 Embedding Lines

Racket and many of its dialects use a `read` function that can be customized by an object called a *readtable*. Linea provides a function to add an escape to line mode with braces (or other desired

delimiters) to any language that uses Racket readtables, providing a convenient way to embed Rash into other languages. The `#lang rash` language enables braces that escape into a line-mode block in both line mode and S-expression mode by default.

Additionally, Rash code can be embedded within languages that do not support readtable customization by using the the `rash` macro, which accepts a literal string of Rash code during macro expansion, such as `(rash "ls -l")`. Essentially, the code string and the `rash` macro are used together to delay a portion of the *read* stage of compilation until the *macro expansion* stage. Because the `rash` macro reads the string during macro expansion time, no run-time string evaluation is performed. Racket's macro system also carries hygiene and source location information with identifiers and other syntax objects, and the `rash` macro uses this information from the string to produce output whose identifiers have proper scope and location reporting information.

The rest of Section 3.5.3 is an aside that discusses trade-offs between these two methods of embedding Rash code. It demonstrates interesting points about embedding multiple languages within the same file that are highlighted by Rash, but it does not contain information necessary to understand the Rash language.

Reading strings at expansion-time with the `rash` macro adds flexibility, since it allows Rash code to be embedded within languages that do not support readtable modification. However, embedding with strings and reading at macro expansion time causes problems with string escaping and macro inspection.

The first and more shallow problem is that nesting traditional string notation requires escaping various characters with backslashes. One solution to the backslash explosion problem is to use the *at-reader* provided by the Scribble (Flatt et al. 2009) documentation language. While this is an improvement, care must be used to invoke the at-reader in a way that does not allow any of its own escaping or eager reading of the inner strings. To ease the nesting of code strings, we created a new notation using «» (guillemet) quotation marks. Guillemet strings include all characters literally with no escaping, and they balance the guillemet delimiters inside the string. So `«a \ «b» ««c»»»` is read the same as `"a \\ «b» ««c»»"`. Guillemet strings allow easy nesting of code strings to arbitrary depths without escaping and are easy to implement and add to a language.

While guillemet strings fix the notation problem of nesting strings, a further issue is that delaying reading to expansion time makes less of the program tree available to macros. Most macros only shallowly inspect their subtrees, since the meaning of deeper subtrees can be changed by deeper

macros. However, some macros, notably the `syntax` macro that instantiates syntax templates, do dig deeper. The `syntax` macro takes a template and replaces pattern variables at arbitrary depths within the template with bindings created by macros such as `syntax-case`.

```
(syntax-case (syntax (first second third)) ()
  [(a b c) (syntax ((one a) (two b) (three c)))])
```

The above example matches the pattern `(a b c)` to the input `(first second third)`. In the right-hand-side of the match, the `syntax` macro replaces the pattern variables `a`, `b`, and `c` with `first`, `second`, and `third` respectively, producing the output `((one first) (two second) (three third))`.

In the following example using the `rash` macro, one would expect the `x` in the pattern to be substituted with the literal string `"world"`:

```
(syntax-case (syntax "world") ()
  [x (syntax (rash «echo hello |>> string-append _ x»))])
```

Because the `x` in the template is inside a yet-unread string, the substitution is missed and an unbound variable error, or worse, the capture of some other `x` variable, occurs. If we change the example to use braces, which do not delay the reading of the sub-form until macro expansion time, the substitution happens as expected.

```
(syntax-case (syntax "world") ()
  [x (syntax {echo hello |>> string-append _ x})])
```

While reader delaying with the `rash` macro can be useful for embedding Rash code within languages that do not support readtable extensions, it is not always composable with other macros. Fixing the syntax template problem by using a readtable extension that does all embedded line reading up-front strengthens the idea of the separate `read`, `expand`, `compile` pipeline.

3.4.4 Line Macros

After the *read* phase, Racket's macro expander determines the meaning of identifiers based on the language used for a module as well as definitions and imports found during expansion. Linea provides its `#%` identifiers with the following default meanings that are used in Rash:

- `#%hash-braces` parameterizes the default input, output, and error redirection for the `run-pipeline` macro and executes its sub-form.
- `#%linea-expressions-begin` desugars to Racket's `begin` form, which evaluates sub-forms in series.

- `#%linea-s-exp` is simply a pass-through macro, so `(#%linea-s-exp (+ 1 2))` desugars to `(+ 1 2)`

- `#%linea-line` detects whether a line starts with a line macro name and either passes through to the specified line macro or inserts a default when one is not given explicitly. `(#%linea-line a b c)` desugars to `(a b c)` if `a` is a line macro. Otherwise it desugars to `(lm a b c)`, where `lm` is the default line macro based on the lexical region containing the line.

Line macros are used to provide an extensible set of keywords for Rash lines. Most line-oriented languages have keywords like `for` and `if` that give special meaning to a line or block of code. Generally, languages have a fixed set of keywords that provide special meaning, and even language authors can not easily extend the set unless they have set aside a large supply of reserved words in advance. Rash uses line macros to provide an extensible set of keywords that change the meaning of the line, and it relies on Linea's reading of subforms with parentheses and braces to provide blocks for "line macros" that need multi-line bodies. With line macros, Rash programmers can use standard control flow forms like loops, conditionals, and try/catch forms that look much like they do in popular line-oriented languages, and also define new ones to taste.

```
in-dir /tmp {
  for f in (directory-list) {
    try {
      rm -rf $f
    } catch e {
      echo An error occured!
      echo $f could not be deleted!
    }
  }
}
```

Line macros enable different line-oriented languages based on Linea to share special forms like control flow and definition forms, and they enable languages to embed just one line of another line-oriented language by prefixing the line with its line macro name. Following the vein of interactive convenience, if two line macros both implement languages that are desireable as a default and that can be distinguished by some heuristic, a user can set the default to a line macro that applies that heuristic and defers to the appropriate macro. For instance, a line macro might apply a heuristic to determine whether to behave like a desktop calculator or a subprocess pipeline.

```
pipeline-or-math 5 + 10 * 3 ;; returns 35
pipeline-or-math ls -l ;; prints a file list
```

The `pipeline-or-math` line macro above simply checks whether the first argument is a number, applies the `infix-math` macro if it is, and applies the `run-pipeline` macro otherwise. If `pipeline-or-math` is set as the default, the example can be simplified to this:

```
5 + 10 * 3
ls -l
```

One may wonder why only macros specifically tagged as line macros are used to bypass the default behavior rather than allowing any macro to do so. A macro use may include other macro names as arguments, including as the first argument. For example, the first argument in a use of the `run-pipeline` macro could be the name of a macro that implements a command. If the command macro were to override the run-pipeline default line semantics, then it could not be pipelined together with other commands without explicitly writing `run-pipeline` at the start of the line.

3.5 Precipitate Pipelining

The primary domain of most command languages is subprocess creation and composition. Thousands of programs have been written to be used as commands in shell languages like Bash, and many programmers use them as their primary means of system interaction and automation. We designed Rash to support this style of programming and system interaction as a first-class citizen.

At its core, Rash's pipelining library includes a module that provides a simple function for pipelining processes. It provides functionality similar to DSL libraries like Scsh (Shivers 1994) and Plumbum (Filiba 2018) that provide functions or macros for pipelining processes in the syntax of their host language. Rash's core subprocess pipelining function can be used like this:

```
(run-subprocess-pipeline '(ls)
                         '(grep rkt)
                         '(wc -l))
```

The simplest line-macro language that we could define for use as a command language might be ordinary Racket with a layer of parentheses removed. However, the above example, even with the outer parentheses removed, still requires balancing several pairs of parentheses. To implement a flatter language, with less syntactic nesting and balancing required, we have designed the pipelining DSL with infix operators. Beyond providing the semantic meaning of an operation, infix operators are parsed to create groupings that appear flat in the source syntax.

While the pipelining DSL includes infix operators, it does not include variable precedence or associativity rules. Left-to-right pipelining is less powerful than other infix operator schemes that allow different precedence and associativity rules, but the simplicity of left-to-right pipelines is helpful in an interactive setting. The left-to-right flow of data is easy to read and reason about, and it also often matches the thought process of users as they write pipelines from left to right. Left-to-right pipelines also make it easy to make the most common edits to commands. Languages with precedence rules for infix operators frequently require edits to several parts of an expression to balance or add new parentheses when adding a new operator with higher precedence. With left-to-right processing, adding new stages to a pipeline requires only appending to the right side of the command. If a user appends an incorrect pipeline stage to a correct prefix, the command can be fixed by only editing the end of the command line. These right-side-only edits require less text editing sophistication to perform with speed and convenience than edits that change several parts of a command buffer.

Beyond providing easy process pipelining, we designed Rash to support similar pipelining notation for Racket functions and objects. Command languages like PowerShell (Snover 2002) have shown that pipelines communicating with objects provide a much richer interface than pipelines communicating with byte streams. For instance, users can more easily filter based on object attributes or compute on tree or graph data in pipelines without intermediate serialization and deserialization steps between processes.

To increase convenience, Rash allows users to define new pipeline operators. Users can write pipeline segments more succinctly with operators tailored for different method calls, and operators can be defined to map or filter over sequences or treat a list as a single object. Custom operators also allow users to customize behavior of common operators, such as customizing the `|` operator to use different policies for automatic globbing or string interpolation.

3.5.1 Pipeline Specification

The grammar of the `run-pipeline` line macro is as follows:

⟨*invocation*⟩ ::= `run-pipeline` ⟨*option*⟩* ⟨*start-op-expr*⟩
⟨*op-expr*⟩* ⟨*option*⟩*
⟨*option*⟩ ::= `&in` ⟨*port-expression*⟩
| `&<` ⟨*filename*⟩
| `&out` ⟨*port-expression*⟩
| `&>` ⟨*filename*⟩
| `&>>` ⟨*filename*⟩
| `&>!` ⟨*filename*⟩
| `&err` ⟨*port-expression*⟩
| `&bg`

```
                |  &pipeline-ret
                |  &strict
                |  &permissive
                |  &lazy
                |  &lazy-timeout ⟨number-expression⟩
⟨start-op-expr⟩ ::= [⟨pipeline-operator⟩] ⟨pipeline-argument⟩*
⟨op-expr⟩       ::= ⟨pipeline-operator⟩ ⟨pipeline-argument⟩*
```

The `run-pipeline` line macro desugars to an invocation of a pipelining function, with the pipeline operator expressions providing the specifications for the pipeline stages. The *start-op-expr* may omit its operator, in which case a lexical default is substituted.

The ⟨*option*⟩ arguments may be written at the beginning or end of the macro invocation for convenience, and influence various aspects of pipeline evaluation.

- The `&in`, `&out`, and `&err` options take an argument expression that should produce a port for the default stdin, stdout, or stderr of subprocesses in the pipeline.

- The `&>`, `&>>`, and `&>!` options are shorthands for `&out` that accept a file name as an identifier or a string and open it in write, append, or truncate mode, respectively.

- If the `&pipeline-ret` option is given, an object representing the pipeline is returned and can be inspected with functions like `pipeline-success?`.

- If the `&bg` option is given, the `&pipeline-ret` option is implied, and the pipeline object is returned immediately rather than waiting for the pipeline to terminate. The pipeline object returned is a Racket *evt*, which can be combined with other evts in complex synchronization patterns in the manner of Concurrent ML (Reppy 1999). As a simple example, passing the pipeline object to the `sync` function causes the current thread of execution to block until the pipeline terminates.

- If neither `&bg` nor `&pipeline-ret` is given, the `run-pipeline` macro returns the value returned by the last stage of the pipeline or raises an exception if the pipeline was unsuccessful.

Pipeline success is determined by subprocess return codes and whether function segments raise exceptions. A pipeline is always unsuccessful if a function segment raises an exception. The handling of subprocesses is determined by the `&strict`, `&permissive`, and `&lazy` options.

- A `&permissive` pipeline is not considered unsuccessful if subprocesses that aren't the last segment of the pipeline return unsuccessful codes. Permissive pipelines kill any subprocesses

that haven't finished before the last pipeline segment does. Permissive pipelines match the behavior of pipelines in common shells like Bash.

- A `&strict` pipeline is unsuccessful if any subprocess returns an unsuccessful return code. Strict pipelines require that all subprocesses terminate before the pipeline is considered finished. Strict pipelines prevent silent failures in intermediate pipeline stages, but can not be used with infinitely running subprocesses without causing the host script to run indefinitely.

- A `&lazy` pipeline strictly checks the return code of all subprocesses that have terminated before the last pipeline segment, or within a timeout period afterward, but assumes any subprocesses that have not terminated by the timeout to be successful and kills them. The lazy mode is a compromise between strict and permissive pipelines to allow infinite subprocesses while catching most failures in intermediate pipeline stages.

Each pipeline operator is itself a macro that is invoked with all arguments to its right up to the next operator, and must desugar to an expression that returns a pipeline segment specification. For example, in this pipeline:

```
echo hello |> string-append _ " world"
```

The `|>` operator is invoked with the syntax list `(|> string-append _ "world")`, and returns code to generate an object specifying an object pipeline segment containing the function `(λ (x) (string-append x " world"))`. The implicit `|` operator receives the syntax list `(| echo hello)`, and returns code to generate an object specifying a subprocess pipeline segment using the argument list `'("echo" "hello")`. Objects specifying subprocess segments include an argument list and an optional redirection port for stderr. Objects specifying function segments include a function that accepts one or zero arguments. Compound segments can also be returned, and contain a list of specification objects. Since pipelines starting with a function segment pass no argument to the function, pipeline operators may provide two transformer functions instead of one: the first is invoked when the operator is in starting position, the second is invoked otherwise.

3.6 Risky Review of Related Work

Rash is not the first attempt to serve a broader portion of the spectrum by embedding a shell into a general-purpose language or by adding general-purpose features to a shell. We discuss related systems in the following subsections, but none of them seamlessly cover the full spectrum from one-

off interactions to a pervasively extensible programming ecosystem (with hundreds or thousands of libraries and packages) like Rash on Racket.

3.6.1 Object Pipelines

PowerShell (Snover 2002) is an object-oriented shell language created by Microsoft. PowerShell aims to replace operating system processes and byte streams with *cmdlets*, which are .Net CLR classes designed to be run as commands, and object streams, respectively. External processes can be used as well as cmdlets, with their output treated as .Net strings. Using .Net objects allows the language to have rich inspection and interactions with objects returned by cmdlets. Due to its use of the .Net CLR, it has rich communication with other CLR languages like C#, and a PowerShell interpreter class can be loaded by other CLR languages to evaluate strings of PowerShell code dynamically.

Like PowerShell, Rash provides a convenient means of using object-oriented system administration commands and pipelines. Rash improves upon the ability to combine and even mix shell-style code with other languages. By leveraging the Racket macro system, Rash expressions can be embedded into files written in other Racket languages and still enjoy the benefits of compilation and static error checking rather than being only dynamically evaluated.

3.6.2 Stand-Alone Shells

Several projects including Oil Shell (Chu 2018) and Elvish (Xiao 2018) are attempts to build command languages that serve a broader section of the maturity spectrum. These stand-alone shell languages improve upon older shells such as Bash by using more composable and reliable general-purpose language constructs. While they serve a broader part of the spectrum than many other shell languages, as stand-alone languages they impose much more implementation and maintenance burden on authors than an embedded shell like Rash. Because of this, stand-alone shells often lack advanced general-purpose language features. Stand-alone shells also tend to have few third party libraries compared to general-purpose languages.

By embedding itself within the general-purpose Racket language, Rash inherits advanced features such as a hygienic macro expander, multi-threading, and delimited continuations, as well as automatically supporting Racket's catalog of third party libraries.

3.6.3 Process Pipelining Libraries

There are many projects that aim to provide a natural means of composing external processes within the syntax of a general-purpose language. These include Scsh (Shivers 1994) for Scheme, Caml-Shcaml (Heller and Tov 2008) for Ocaml, Turtle (Gonzalez 2018) and Shelly (Weber and Rockai 2018) for Haskell, Plumbum (Filiba 2018) for Python, and many more. Most of these provide means of mixing host-language code in pipelines, such as by inserting a function that reads and writes bytes into a subprocess pipeline. Some include creative and useful ways of integrating the process and byte-stream model into the host language. For instance, Caml-Shcaml provides adaptors to give various types to byte streams and allows rich interaction within Ocaml code while preserving the original text of lines for programs later in the pipeline to process the lines unchanged. These pipelining libraries are designed for creating scripts and not to provide a syntax for convenient interactive use of processes or functions for system administration. The shell-pipeline library within Rash provides similar functionality to these, and Rash combines it with syntactic changes to add a focus on interaction.

3.6.4 Shells Embedded in General-Purpose Languages

Eshell (Free Software Foundation 2018a) embeds a Bourne-like shell in Emacs Lisp. It allows program arguments to be computed with Eshell expressions, it allows lines of parenthesised elisp to be used in place of lines of shell code, and it implements many commands as Eshell functions. Scripting with Eshell is possible but discouraged, and scripts must be executed within an instance of Emacs. Eshell's shell pipelines are character oriented, and communication occurs by placing command output into an emacs buffer. Eshell does not have a facility such as line macros to customize the meaning of its line-oriented syntax or provide custom keywords for different block constructs.

Xonsh (Scopatz 2018) is a language built using a superset of the Python grammar that includes Bourne-shell-like syntax for subprocess pipelines. To resolve ambiguities in the combination of grammars, it uses heuristics such as whether names are bound as variables, generally preferring the interpretation of the standard Python grammar over its alternative Bourne-style grammar. It also includes syntax for explicitly switching between Python and shell code, and allows Python functions with appropriate interfaces to be used in place of processes in byte-stream pipelines. It includes import hooks to allow Python files to be imported by Xonsh files and vice versa. It

has many features to provide a convenient and useful interactive shell for system administration and general computer use. Xonsh does not have tools to allow Python functions that accept and return arbitrary objects to be used for system administration with the same ease and convenience as external processes, syntactic support for shell-style pipelines of objects like PowerShell and Rash, or a macro system for user-defined syntax extensions.

Ammonite (Haoyi 2018) extends Scala with things like top-level expressions for easier use in writing small scripts. It provides a library of functions for shell-style file system manipulation, and a shell-style REPL for live interaction. It does not provide any extra support for running or pipelining external processes, but rather focuses on object-oriented shell scripting using scala functions. The interactive REPL uses a syntax that includes many elements that may be considered heavy for daily interactive use, particularly by programmers used to the light-weight Bourne shell syntax.

Neugram (Crawshaw 2018) is similar to Ammonite in that it provides a more scriptable layer on top of a general-purpose language, in this case Go. It provides a method of embedding process pipelines in shell syntax. It can be used interactively but doesn't emphasize use as an interactive shell.

There are various other efforts attempting to mix process pipelining and shell-style programming with more general-purpose languages. Salient examples include Zoidberg (Berger 2018) (based on Perl) and Closh (Dundalek 2018) (based on Clojure). They generally include a syntax modified to be easier to use in an interactive command line, subprocess pipelines, and a way to switch between code in the host syntax and the shell syntax. These languages usually have limited compatibility with their host languages, for instance not being able to provide functions as a library to programs written in the host language. Many of them focus on interactive use, with little or no support for scripting. Rash differs from other shells embedded in general-purpose languages by having full compatibility with other Racket languages, supporting pipelines of objects as well as byte streams, allowing racket functions to be used as interactive commands that have syntactic convenience equal to that of subprocesses, and enabling user extensibility with hygienic macros. These features allow rash to more fully enable interactions and scripts to advance along the maturity spectrum.

3.7 Hasty Conclusion

Programmers want to use highly dynamic, terse languages specialized for system administration and other interactive tasks, but they would also like to automate these tasks by transforming their

interactions into scripts. We have demonstrated how Rash's approach of embedding a shell language within a general-purpose language allows programmers to work in a suitable command language while also growing their scripts gradually into mature programs using general-purpose language features and libraries. Rash accomplishes this by including a terse line-oriented syntax, by allowing its line notation and Racket's notation to be nested together, by preserving and expanding on Racket's pervasive extensibility, and by providing a pipelining DSL that generalizes Unix pipelines.

3.7.1 References

Joel Berger. Zoidberg - A modular perl shell. 2018. https://metacpan.org/pod/Zoidberg

S. R. Bourne. Unix Time-Sharing System: The UNIX Shell. *Bell System Technical Journal* 57(6), 1978.

Andy Chu. Oil. 2018. https://www.oilshell.org/

David Crawshaw. Neugram, Go Scripting. 2018. https://neugram.io/

Jakub Dundalek. Bash-like shell based on Clojure. 2018. https://github.com/dundalek/closh

Tomer Filiba. Plumbum: Shell Combinators and More. 2018. https://plumbum.readthedocs.io/

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: closing the book on ad hoc documentation tools. In *Proc. SIGPLAN International Conference on Functional Programming*, 2009.

Free Software Foundation. Eshell Manual. 2018a. https://www.gnu.org/software/emacs/manual/html_mono/eshell.html

Free Software Foundation. GNU Bash. 2018b. https://www.gnu.org/software/bash/

Gabriel Gonzalez. turtle: Shell proogramming, Haskell-style. 2018. https://hackage.haskell.org/package/turtle

Li Haoyi. Ammonite Documentation. 2018. http://ammonite.io/

William Gallard Hatch and Matthew Flatt. Rash: From Reckless Interactions to Reliable Programs. In *Proc. 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2018. https://doi.org/10.1145/3278122.3278129

Alec Heller and Jesse A. Tov. Caml-Shcaml. In *Proc. ML Workshop*, 2008.

Microsoft. Typescript Language Specification. 2018. http://www.typescriptlang.org/

Jon Rafkind and Matthew Flatt. Honu: syntactic extension for algebraic notation through enforestation. In *Proc. International Conference on Generative Programming and Component Engineering*, 2012.

J. H. Reppy. Concurrent Programming in ML. Cambridge University Press, 1999.

Anthony Scopatz. Xonsh Documentation. 2018. http://xonsh.org/

Olin Shivers. A Scheme shell. Laboratory for Computer Science, MIT, TR-635, 1994.

Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Scheme and Functional Programming*, 2006.

Jeffrey P. Snover. Monad Manifesto. Microsoft, , 2002.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. Dynamic Languages Symposium*, 2006.

Sam Tobin-Hochstadt, Matthias Felleisen, and T. Stephen Strickland. The Design and Implementation of Typed Scheme. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2008.

Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. ACM Symposium on Dynamic Languages*, 2014.

Greg Weber and Petr Rockai. shelly: shell-like (systems) programming in Haskell. 2018. https://hackage.haskell.org/package/shelly

Qi Xiao. Elvish. 2018. https://elv.sh/

CHAPTER 4

GENERATING CONFORMING PROGRAMS WITH XSMITH

Fuzz testing is an effective tool for finding bugs in software, including programming language compilers and interpreters. Advanced fuzz testers like Csmith can find deep semantic bugs in language implementations through differential testing. However, input programs used for differential testing must not only be syntactically and semantically valid, but also be free from nondeterminism and undefined or unspecified behaviors. Developing a fuzzer like Csmith that produces such programs can require tens of thousands of lines of code and hundreds of person-hours. Despite this significant investment, fuzzers designed for differential testing of different languages include many of the same features and analyses in their implementations.

To make the implementation of language fuzz testers for differential testing easier, we introduce Xsmith. Xsmith is a Racket library and domain-specific language that provides mechanisms for implementing a feature-complete fuzz tester in only a few hundred lines of code. By sharing infrastructure, allowing declarative language specification, by allowing procedural extensions, Xsmith allows developers to write correct fuzzers for differential testing with little effort. Using Xsmith, fuzzers have been developed for a variety of languages.

Xsmith's domain of differential test-case generation is a good application for an embedded DSL. While much of each Xsmith fuzzer is written declaratively, mature fuzzers take advantage of the opportunity to use arbitrary Racket code to extend Xsmith with generation rules that are specific to the target language of the fuzzer. A stand-alone DSL would necessarily either omit this capability, or implement a set of general-purpose programming features alongside the core domain-specific concerns of the language.

4.1 Introduction

The effectiveness of random testing, or "fuzzing," is determined both by the chosen input generation strategy and the method used to detect failing tests, or *test oracle*. The generation or

mutation of test cases using random bytes can in theory generate any test and therefore cover any code path, but typically exercises only "shallow" code in parsing and input validation stages of a program. Meanwhile, grammar- and type-aware input generators can exercise "deep" code paths that pass these initial validation steps. The test oracle of detecting crashes can be used with any test case generator, but can only find obvious errors such as memory or assertion violations, and not subtle semantic bugs. Property-based testing can find semantic bugs, but requires users to write invariant properties of test results or side effects, which is expensive.

The test oracle of interest for this chapter is *differential testing* (McKeeman 1998), where the same input is given to multiple implementations of a system—in our case, a programming language implementation. If the multiple implementations are correct, then giving them all the same input program (and executing the returned output if the implementation is a compiler) should produce the same result. When there is a difference in program output, a bug has been found. While differential testing can find subtle semantic bugs without writing extra properties, there is a catch. If the input program relies on any behavior that is not guaranteed to be the same between multiple executions and multiple implementations of the language, differences in output do *not* necessarily indicate a bug. Therefore, differential testing requires programs that conform strictly to the language specification, including avoiding undefined, implementation-defined, or nondeterministic behavior. We call such inputs *conforming* inputs.

One notable generator of conforming programs is Csmith (Yang et al. 2011). Csmith successfully identified hundreds of bugs in mainstream C compilers, including LLVM and GCC. However, the development of Csmith took hundreds of person-hours and resulted in nearly 40,000 lines of code. Although practically any language could benefit from such a program generator, it is impractical for most language developers to write a variant of Csmith targeting their own language.

To ease the development of fuzzers that generate conforming programs, we created *Xsmith*. Xsmith is a domain-specific language (DSL), implemented as a Racket library, that allows for the rapid and concise implementation of conforming program generators for arbitrary programming languages.[1]

The primary contributions of this chapter are:

- A domain-specific language for creating conforming program generators.

[1] Xsmith is open source, with code available at https://gitlab.flux.utah.edu/xsmith/xsmith/

- A generic framework for declaring type and effect systems for program generation used in the DSL implementation.

- Several example fuzzers implemented using the DSL, some of which have been used to find bugs in language implementations used in production.

- An analysis of the effectiveness and cost of the example fuzzers.

4.2 Design

Because Xsmith is intended to support the specification of fuzzers for a wide range of programming languages, its design and implementation are complex. In this section we describe the overall design of Xsmith at a high level. Throughout the section, we develop a simple fuzzer for a small toy calculator language (named `calc`) as an example. The grammar of this language is shown in Figure 4.1.

Xsmith fuzzers generate program trees by starting with a "hole" node for the top-level production of the grammar. Xsmith iteratively fills hole nodes in the tree with nodes corresponding to appropriate grammar productions, which may themselves have holes as children, as shown in Figure 4.2. A fuzzer author provides Xsmith with a grammar declaration that determines the grammar of program generation. The grammar used by an Xsmith fuzzer is generally an abstract grammar that matches the logical structure of the language or a subset of the language. This abstract grammar is necessarily the same as the grammar that might be used for parsing the language or the grammar of a compiler's AST representation.

The declared grammar is compiled to generate an object-oriented attribute grammar specification for the RACR attribute grammar library (Bürger 2015). The generated attribute grammar matches the input grammar but adds a hole production as an alternative for each user-provided production. While Xsmith program generation is grammar-driven at its core, generation of conforming programs requires many considerations beyond the grammar. To enable filtering and probability weighting based on these many other considerations, the grammar is also compiled into a set of *choice classes*, one for each given production. When the generation algorithm selects the next hole to fill, a choice object is instantiated for each production that could be used to replace the hole, and the resulting list of choice objects is used to make the decision.

Below is an example Xsmith specification that implements the grammar of the `calc` language.

Each arm of the `add-to-grammar` form contains the name of a production, the supertype of that production, and a list of fields for that production. The `Add` and `Div` productions each have two children, and these children must themselves be `Exp` (expression) productions. These `Exp` children will be instantiated as `Exp` holes. The `Int` production specifies an integer literal, whose `val` child may contain arbitrary Racket data. In this example, `val` is initialized to a random integer between 1 and 100. (This random selection occurs each time an `Int` hole is filled.)

```
(add-to-grammar calc
 [Exp #f ()
      #:prop may-be-generated #f]
 [Add Exp ([lhs : Exp]
           [rhs : Exp])]
 [Div Exp ([lhs : Exp]
           [rhs : Exp])]
 [Int Exp ([val = (random 1 100)])])
```

In addition to writing a grammar, a fuzzer author can declare various *properties* of each grammar production. Properties are used to specify semantic details of grammar productions, such as their types and binding structure. Each property has its own form, potentially accepting declarative data, arbitrary user-provided Racket code, or some mix of the two. In the following code, `choice-weight` of the `Div` and `Int` nodes is adjusted to change their generation probabilities.

```
(add-property calc choice-weight
 [Div 10]
 [Int 5])
```

Properties are compiled into *attributes* — methods for attribute grammar nodes — or *choice methods* — methods for choice classes. Internally, an Xsmith-based fuzzer uses attributes to compute information about the program being generated and its constituent nodes. For example, the `xsmith_type` attribute computes the type of a node, and the `_xsmith_visible-bindings` attribute computes a list of bindings available for reference at a given point in the tree. Choice methods are used to compute details for potential choices. For example, the `_xsmith_satisfies-type?` choice method is used to filter out choices with invalid types, the `_xsmith_wont-over-deepen` method is used to make choices that will keep the generated program size bounded, and the `_xsmith_fresh` choice method determines how a chosen node is initialized.[2]

[2]Because attribute and method names are bare symbols without namespacing, we use the `xsmith_` prefix by convention for names defined by Xsmith itself, and we use a leading underscore by convention for private attributes and methods intended for use only in Xsmith's implementation.

Attributes may query other attributes during computation, while choice methods may query both attributes and other choice methods. Custom attributes and choice methods may be defined directly, but must be written in a procedural style. The majority of attributes and choice methods are compiled from properties, since they allow users to declaratively specify details at a high level. These include semantic details such as whether a grammar node represents a definition, whether a node causes a read or write to a variable, or whether a node's children have a specified execution order. Users may additionally define custom properties with their own compilation transformers to abstract patterns between fuzzers, although Xsmith is designed to support the majority of properties most languages' fuzzers would need out of the box.

When a tree with no holes is finally completed, it must be converted to text for programming language implementations to consume. The `xsmith_render-node` attribute, defined by the `render-node-info` property, is used to convert the tree into text or an intermediate representation to facilitate pretty-printing, such as s-expressions or document objects from Racket's `pprint` library. When an intermediate representation is used, a final text converter must be specified as well. Below is an example renderer for our `calc` language.

```
(define (render-infix operator)
  (lambda (n)
    (format "(~a ~a ~a)"
            (att-value 'xsmith_render-node
                       (ast-child 'lhs n))
            operator
            (att-value 'xsmith_render-node
                       (ast-child 'rhs n)))))
(add-property calc-grammar render-node-info
 [Add (render-infix "+")]
 [Div (render-infix "/")]
 [Int (lambda (n)
        (number->string (ast-child 'val n)))])
```

4.3 Cost Reduction Features

Xsmith has many features that work together to make the creation of conforming program generators inexpensive. These features include forms for declaring grammar, types, and name scoping and resolution, as well as a "canned components" library to encapsulate language similarities, features for undefined behavior handling, and so on. In this section we give an overview of these features, discussing their usage and design.

4.3.1 Grammar and Syntax

The first step to generating programs that are conforming is to follow a grammar. Xsmith generates program trees according to an abstract grammar provided by the user.

Usage. A user can define an abstract grammar for their generator by using the `add-to-grammar` form. Each grammar production is declared as a subtype of another grammar production (including the abstract base grammar production, `#f`). Grammar productions may have any number of children, which can be specified as either being grammar productions (of a given type) or storage locations for arbitrary Racket data. Children may be annotated with a Kleene star to indicate repetition (zero or more repetitions). Below is an example of a partial grammar definition.

```
(add-to-grammar js-component
 [ArrayLiteral Expression ([elem : Expression *])]
 [IntLiteral Expression ([value])])
```

Design and Implementation. Xsmith's grammar and AST data structures rely on the RACR (Bürger 2015) attribute grammar library. RACR allows Xsmith grammar nodes to include dynamically calculated attributes that can depend dynamically on attributes or data from parent or child nodes. As an AST grows, RACR automatically keeps track of which attributes need to be recomputed.

Xsmith relies on grammars to help users more easily define and re-use language components. To transform the AST into an input that is valid for a compiler or interpreter, Xsmith includes a multi-step render phase. The goal of the render phase is to produce a program string that can be output to a file. However, rather than defining a transformation from each grammar node to a string, it can be easier to have pleasantly formatted output by using an intermediate format. For example, Lisp code can be rendered more easily and legibly by transforming the abstract tree first into s-expression data structures, then pretty printed by Racket's s-expression `pretty-print` function. For more line-oriented languages like JavaScript, more readable printing can be achieved by rendering to the data structures of Racket's `pprint` pretty-printing library, then finalizing the string conversion using the library's `pretty-format` function.

4.3.2 Types

Generators of conforming programs need to produce well-typed code to pass the type-checking stage of programming language implementations. The requirement for type-correct code is perhaps less strict for dynamically typed languages than for statically typed languages. However, if code for dynamically typed languages is generated without regard to types, most expressions will raise

run-time type errors. Xsmith includes a type system framework that allows fuzzers to generate well-typed code for a variety of languages.

Usage. The type system used by a fuzzer is defined by the `type-info` property. This property is two-armed. The first arm specifies the types a grammar node can inhabit. The second arm is a function that receives a tree node of the specified production and its type and returns a dictionary of types for the node's children. In the code below, the `LiteralString` and `StringAppend` productions are declared to always have type `string`, while the `VariableReference` is declared to use a type variable that can be unified with any type. The `LiteralString` and `VariableReference` productions have no children, but the `StringAppend` production constrains its children to inherit its type.

```
(define no-child-types (lambda (n t) (hash)))
(add-property
 js-component
 type-info
 [LiteralString [string no-child-types]]
 [StringAppend [string (lambda (n t) (hash 'l t 'r t))]]
 [VariableReference [(fresh-type-variable) no-child-types]])
```

Design and Implementation. Xsmith allows its user to specify type systems that contain base types, function types, product types, generic types (such as lists and arrays), nominal records, and structural records. Xsmith supports less expressive type systems than some other tools do, such as PLT Redex (Felleisen et al. 2009). Some of these tools support the programmer writing arbitrary type judgments that are compiled to first-order logic and subsequently used for type checking and generating random terms (Fetscher et al. 2015). However, Xsmith has more constraints on generated programs than merely being well-typed, and other analyses—such as those for the effect system—require the analysis of types. Therefore, we have built a more limited type definition framework that allows this cross-analysis. Despite these limitations, Xsmith's type checking framework is sufficient to support fuzzing most features of popular programming languages.

Type systems specified in Xsmith may also support subtyping. During type checking, Xsmith performs *subtype unification* between the types that a tree node declares that it supports, the types provided by its parent node, and any types declared by relevant definition nodes for references. Subtype unification, like traditional variable unification during type inference, mutates type variables to indicate relationships between type variables and between type variables and concrete types. However, unlike traditional unification, subtype unification reflects the asymmetric relationship of subtyping. Type variables in subtype relationships form a lattice of related types, where `(subtype-`

`unify! a b)` relates `a` as a subtype of `b`, placing `a` below `b` in the lattice. Symmetric unification in this model is implemented merely as two subtype unifications:

```
(define (unify! a b)
  (subtype-unify! a b)
  (subtype-unify! b a))
```

When a type variable `a` is already related as a subtype to type variable `b` and `(subtype-unify! b a)` is executed, the relationship lattice is squashed such that `a`, `b`, and all variables between them in the lattice are unified into a single type variable.

While it is well known that unification-based type inference is incompatible with subtyping for type checking of existing programs, Xsmith can use unification because it type checks program fragments while generating a fresh program. Thus, Xsmith can always choose a term for any hole that will satisfy the type checker, assuming the type system specification is correct.

4.3.3 Language Similarities

Many programming languages have similar language features. To help Xsmith users avoid implementing these features afresh for each language they write an Xsmith fuzzer for, Xsmith provides a library of "canned components" that automatically define the necessary grammar nodes and properties.

Usage. The main forms provided by the library are the `add-basic-statements` and `add-basic-expressions` macros. Each of these has a variety of optional keyword arguments and extends a grammar with forms specified by those arguments. These productions include literals, accessors and mutators for mutable arrays and dictionaries, and function application and definition. The code below demonstrates how many standard productions can be added to a grammar, with appropriate type rules and other properties, with a `canned-components` macro.

```
(add-basic-statements js-component
                      #:Block #t
                      #:ReturnStatement #t
                      #:IfElseStatement #t
                      #:AssignmentStatement #t
                      ...
                      )
```

The `canned-components` library also provides the `add-loop-over-container` macro, which has various keyword arguments allowing a user to specify whether the loop form is a statement or an expression, which types of containers it can loop over, and the type of the loop's result. These productions are added with all relevant properties for the type and effect systems, name analysis,

etc. The only non-optional property that the user must add is the `render-node-info` property. The code below demonstrates how a loop form can be added to a grammar.

```
(add-loop-over-container js-component
 #:name ForLoopOverArray
 #:loop-ast-type Statement
 #:body-ast-type Block
 #:collection-type-constructor
 (λ (elem-type) (mutable (array-type elem-type)))
 ...)
```

Design and Implementation. The canned components are implemented as a library of macros that generate the most common patterns of grammar and property definitions. The `canned-components` library reduces the amount of code required to write a new fuzzer, and it reduces the duplication of tedious and error-prone type rules and other properties that are easy to get slightly wrong.

4.3.4 Unspecified and Implementation-defined Behavior

For practical reasons, some programming languages leave the semantics of certain constructs either up to individual implementations or completely undefined. A generator of conforming programs must avoid every type of unspecified or non-deterministic behavior in the programs that it generates. One common unspecified behavior concerns the order of evaluation of subexpressions, such as multiple arguments in a function call. While the evaluation order is unimportant in the evaluation of purely functional code, effectful code that assigns variables or mutates values requires a consistent ordering to be conforming.

Usage. To avoid generating code with an unspecified effect order, a user simply annotates which nodes include different effects, such as reading and writing to variables. This is demonstrated in the following code.

```
(add-property js-component reference-info
   [Reference (read)]
   [Assignment (write)])
```

Design and Implementation. Xsmith includes an effect analysis that enumerates the possible effects of code evaluation and conservatively avoids ordering conflicts. Tracked effects include variable reference and assignment, projection and mutation of values like arrays, and higher-order function application. Whenever a potential conflict arises, such as referencing a variable in one function argument while assigning to the same variable in another argument, Xsmith filters out the choices that would lead to the generation of offending programs. A user may annotate grammar

nodes that impose a specified ordering on their children, such as block and sequence constructs, with the `strict-child-order?` property.

Besides effect ordering, programming languages have a wide and inconsistent variety of features that cause undefined, or at least unhelpful, behavior. For example, out-of-bounds array access is an undefined behavior in C, while in many other languages it is defined to raise an exception. Although a raised exception is well defined and potentially an interesting part of the language API to fuzz test, in typical fuzz testing an array access exception is likely not a useful behavior. For example, because the set of possible values of type `int` in a given programming language is likely much larger than the set of usable array sizes, array access with approximately uniformly generated `int` values will raise exceptions much more often than it will yield values. For both defined and undefined semantics of array access, it is usually best to generate code that wraps such accesses to convert the index to a number within bounds or provide a fallback result value.

Because these undefined or unhelpful behaviors are language-specific, each fuzzer needs some amount of unique attention to them. The common pattern used in our example fuzzers is to include program header text that defines safe wrapper functions for potentially problematic functionality, possibly including extra fallback arguments (e.g., for accessing an empty list). The grammar can then target the safe wrappers instead of the raw unsafe operations. Some of these behaviors are fairly common and have been captured in the canned-components library.

4.3.5 Name Scoping and Resolution

Generators of conforming programs need to produce programs where variables referenced are defined in scope. Xsmith includes a generic analysis to ensure that variables are well scoped. If a reference is generated in a position where no appropriately typed variable is in scope, Xsmith will automatically add an appropriate definition node into a scope that is visible to the new reference.

Usage. Users can annotate which grammar nodes bind and reference variables using the `binder-info` and `reference-info` properties, such as with the following code.

```
(add-property js-component binder-info
   [Definition ()]
   [FormalParam (#:binder-style parameter)])
```

However, common patterns for binders and references have also been captured in the canned-components library, so most users do not need to interact with these properties directly.

Design and Implementation. Our resolution system is based on scope graphs (Néron et al.

2015), which is a generic system for representing variable scoping in programming languages. Based on user-provided annotations, Xsmith will generate scope graph models for generated programs, and use them to find which variables that are in scope at any position.

4.3.6 Language-Specific Analyses

While Xsmith includes several generic analyses, an advanced fuzzer may benefit from a language-specific analysis. Because they are language-specific, such analyses can not reasonably be included in the Xsmith framework. However, Xsmith provides several features that aid a user in writing their own custom analyses.

Usage. Users can directly write custom attributes with the `add-attribute` form, and add custom choice methods with the `add-choice-method` form. The following code defines a custom `choice-method` that can be added to the set of generation filters to prevent variable references from being generated as the direct children of `Division` nodes.

```
(add-choice-method js-component allow-ref
 [VariableReference
  (λ () (not (eq? (ast-node-type
                   (ast-parent (current-hole)))
                  'Division)))])
```

Additionally, users may define custom Xsmith properties, essentially making a mini-DSL to compile declarative data into attributes and choice methods. Defining custom properties requires familiarity with Racket macro writing techniques, and we will leave discussion of custom properties to the Xsmith documentation. Finally, users can leverage Xsmith's generic analyses as dependencies of their analyses, such as by querying a node's type during a custom analysis.

Design and Implementation. Xsmith uses the RACR attribute grammar library (Bürger 2015) to implement its attribute system, and uses the Racket class library to define choice classes. Properties are a syntactic form specific to Xsmith. Properties each have a syntax transformer function that accepts the syntax fragments from each place `add-property` was used with that property. Each property requires a specification of which attributes, choice methods, and other properties will be generated. Additionally, properties may specify other properties whose syntax fragments they may read in in their transformation function. All properties used in an Xsmith fuzzer have their transformers run in an ordered based on read and write dependencies.

Ultimately, custom properties, attributes, and choice-methods provide a way for Xsmith users to extend Xsmith with arbitrary Racket code. This allows Xsmith fuzzers to include features never

imagined by Xsmith's authors.

4.3.7 Making Decisions

Xsmith includes features for both filtering potential decisions and for adjusting the probability of different choices when filling holes in the generated AST.

Usage. A user can add custom choice methods as filters by using the `choice-filters-to-apply` property. The following code applies filter `choice-method` defined above to restrict `VariableReference` generation.

```
(add-property choice-filters-to-apply js-component
  [VariableReference (allow-ref)])
```

A user can adjust the frequency of different grammar node choices with the `choice-weight` property, shown below.

```
(add-property choice-weight js-component
              [IfStatement 50]
              [AssignmentStatement
               (λ (hole) (if (eq? (ast-node-type
                                   (ast-parent hole))
                                  'IfStatement)
                             20
                             30))])
```

The choice weight may be given a positive integer value or a function that returns a positive integer based on an analysis of the tree.

Design and Implementation. When filling a hole, Xsmith instantiates one choice object with the appropriate class for each subtype of the required node type. For example, in a hole of type `Expression`, Xsmith will instantiate a choice object for each of `IntegerLiteral`, `VariableReference`, `Addition`, and so on. Each of these choices is filtered based on the specifications given to the `choice-filters-to-apply` property, including default filters such as type satisfaction. After a list of valid choices has been filtered, each remaining choice has its `choice-weight` computed. A choice is then randomly made, with each choice having probability *choiceWeight / weightSum*.

4.3.8 Refinement

For some language features, you may wish to modify a complete program using whole-program analysis. For example, a generator may include a value analysis that can prove that no undefined or non-conforming behavior can happen at a given program node. That node can then be transformed to use a raw "unsafe" operation instead of a safe wrapper version.

Usage. Refinement is run after a complete tree is generated. A refiner is defined as a series of predicates to test nodes of the tree for relevance, and finally a transformer for nodes that match the criteria. Refiners walk over the tree finding nodes that can be transformed, and execute until the tree reaches a fixpoint.

Design and Implementation. Refinement uses essentially the same algorithm as generation. The differences are that refiners use custom predicates instead of simply checking for holes, and use custom transformers instead of the standard node generation function.

4.3.9 Parametric Generation

The Zest fuzzer (Padhye et al. 2019) introduced the concept of *parametric generation* of test inputs. Parametric generation allows a generative fuzzer to be feedback directed.

Usage. When generating a program using Xsmith's command line, users may provide optional flags to configure parametric generation.

Design and Implementation. Xsmith uses the Clotho (Darragh et al. 2020) Racket library to interpose on pseudo-random number generation with a byte vector parameter when given, or to save a vector of pseudo-random bytes used. Instead of a call to a `random` function producing a pseudo-random 32-bit integer, it returns the next 32-bit slice of the input parameter. By mutating the byte vector parameter, a random generator can preserve the structure of some choices while changing others. If a byte vector parameter is too short to make all choices, part of the vector can be used to seed future pseudo-random number generation. Future pseudo-random numbers can be saved to append to the longer byte vector, allowing the full choice history to be saved for future modification.

When fuzzing, feedback direction, such as line coverage of the system under test, can be used as an indication of whether a given test candidate is "interesting" even if it has not directly uncovered a bug. If a recording is made of the byte vector used to make choices when generating an interesting candidate, it can be modified to create different but similar test cases. Mutant byte vectors may lead to test cases that preserve the coverage (or other interesting aspects) of a previous test while finding more coverage.

The Zest fuzzer found parametric generation to be useful for finding more bugs with generative fuzzers. However, we have not yet built test environments for using our fuzzers that can capture feedback data such as line coverage from instrumented compilers, or extended our test harness to

collect parameters, identify interesting test cases, and create mutant parameters. Thus we have not evaluated feedback direction with Xsmith.

4.3.10 Automatic Test-Case Reducer

When a generated test case triggers a bug in a language implementation, it is not always obvious what the bug is, or what the aspects of the program triggered the bug. Yang et al. (2011) found that bugs tended to be found more frequently with relatively large programs, meaning that generating programs likely to find bugs implies generating programs where the specific bug is hard to see. In practice, test cases must be minimized to find a small, understandable test case that still triggers the bug.

There are reducers for specific programming languages, such as C-Reduce (Regehr et al. 2012) for C. Sometimes they include fairly generic passes that work reasonably well for other languages, for example C-Reduce has been used successfully for programs in various languages. However, C-Reduce can not effectively use its semantics-aware passes on languages that differ from C, and even its relatively generic passes struggle when a language's syntax differs significantly from C. Xsmith includes a generic, built-in test case reducer that operates on the generated AST of a program.

Usage. When generating a program using Xsmith's command line, in addition to flags for specifying a specific seed and configuration, a user may provide the optional `--reduction` flag. The flag takes as an argument a path to a script to run on each reduction candidate. The script should exit with status 0 when the candidate exhibits the interesting behavior of the original, and exit with nonzero status otherwise.

Design and Implementation. Xsmith's reducer performs a series of simple reductions to the generated AST while maintaining correctness guarantees. These reductions include replacing AST nodes with atomic literal nodes, replacing variable references with references to the outermost variable of the appropriate type, removing unreferenced definition nodes, and removing optional repeated elements in a grammar.

At each step, the program is re-rendered and sent to the reduction script to check whether the reduction preserves the interesting behavior. When a node is considered for replacement, it is tentatively replaced by a hole node, and fresh choice objects are generated. However, only choices fitting a specific replacement strategy are considered, and if no valid choice is relevant to the criteria or successfully passes the script test, the replacement is undone. Deleting optional

repeating elements in a grammar, such as elements in a list literal, is done by simply removing the node from the AST and testing with the script. To preserve program correctness, a property annotation (`reducible-list-fields`) is required to specify which repeating fields are actually optional.

Xsmith's built-in reducer only generates reduced versions of test cases that are valid and could in principle have been generated by Xsmith. Additionally, Xsmith doesn't reduce any literal code, such as wrapper definitions for preventing undefined behavior, that are included with the renderer. Thus, Xsmith's reducer can not reduce test cases to truly minimal examples. However, it can quickly perform a variety of possible minimization steps while preserving semantic guarantees for differential testing. Reduced output from Xsmith can be further reduced by hand or by a secondary automatic reducer, such as C-Reduce. Our experience so far has shown that Xsmith's built-in reducer is useful and effective despite its simplicity and limitations, and that a secondary manual reduction pass to eliminate rendering boilerplate is relatively easy.

4.3.11 Command-Line Interface

Xsmith automatically generates a command-line interface for fuzzers. The command-line interface includes flags for configuring standard options, such as maximum depth, generation seed (or parametric generation byte vector), single-run or server mode, and options for printing debug info. The command line can be arbitrarily extended with fuzzer-specific flags that can be queried throughout the fuzzer. In particular, grammar productions can be labeled with *feature* properties, and the command-line interface can be declaratively extended to include optional flags with default values for including or excluding each of those features.

Usage. Each Xsmith fuzzer uses the `define-xsmith-interface-functions` form that compiles the specification and defines a function that parses a command line and runs the fuzzer. The following code compiles the grammar definition and properties, and runs the command-line code when its containing file is run as a program.

```
(define-xsmith-interface-functions [js-component]
  #:fuzzer-name simple-javascript
  ;; Optional arguments may add command-line options.
  ...)
(module+ main (simple-javascript-command-line))
```

4.3.12 Modularity

Xsmith fuzzers may be declared in a modular fashion. Grammar productions, properties, and other aspects of fuzzers are added to fuzzer *components*, which can be composed when defining a complete fuzzer. Fuzzer components can be split between multiple files, and even shared between multiple fuzzers. A family of related fuzzers can be defined as extensions to the same base fuzzer.

4.4 Example

To give a sense of what a small but still featureful Xsmith fuzzer looks like, we present a small JavaScript fuzzer. Figure 4.3 is an abbreviated example of a simple JavaScript fuzzer, with elided code sections marked by "...". A full version of this example is included in the Xsmith source repository. While the full version is longer, it is still only 412 lines as measured by the `wc` utility.

This example demonstrates a fuzzer that takes advantage of the canned-components library to generate conforming JavaScript programs that may utilize arrays, first-class functions, objects (encoded in Xsmith as structural record types), if statements, and loops. The largest amount of code elided from the full version is in program rendering. The rendering step tends to be verbose and varies by language, but is not complicated or difficult to code.

This example uses a `safe_divide` function, defined in a header, to avoid issues that arise from dividing by zero. Similarly, the full version defines more safe wrappers for array reference and assignment. While these operations are not undefined or even necessarily troublesome behavior in JavaScript, we use them to avoid having values collapsing to JavaScript's `undefined` value, which would otherwise be overwhelmingly common. This demonstrates a common pattern used when creating fuzzers with Xsmith to avoid undefined behavior or the raising of common exceptions.

This example also does not directly use any `add-to-grammar` forms, because the entire abstract grammar used is provided by the canned components library. A larger fuzzer will generally include canned components as well as `add-to-grammar` forms that add various built-in functions specific to the language.

4.5 Evaluation

We evaluate Xsmith by considering a set of fuzzers built with Xsmith as well as bugs found with those fuzzers. We assess the difficulty of creating fuzzers with Xsmith and the number and quality of bugs found. In particular, we examine a particular case study of fuzzing Dafny.

4.5.1 Fuzzers

Generators of conforming programs typically require a lot of effort to create. Csmith, a predecessor to Xsmith and its major inspiration, required hundreds of person-hours and tens of thousands of lines of code. Xsmith fuzzers require substantially less effort and code. Figure 4.4 compares sizes of a selection of conforming program generators in terms of code size.

While the implementations of Xsmith-based fuzzers are significantly smaller than similar conforming program generators like Csmith (Yang et al. 2011), Verismith (Herklotz and Wickerson 2020), and SQLSmith (Seltenreich 2020), they still produce programs that are syntactically and semantically valid as well as free from undefined or nondeterministic behavior. Additionally, Xsmith fuzzers can be featureful, generating correct code for conditionals, rich types, variable references, and so on.

Xsmith is not the only generic framework for creating programming language fuzzers. Other generic frameworks include Polyglot (Chen et al. 2021) and StarSmith (Kreutzer et al. 2020). While Xsmith has the greatest focus on differential testing compared to other generic frameworks, it compares well in terms of implementation effort per fuzzer, as shown in Figure 4.5.

Some StarSmith line counts are approximate, because for SQL and SMT their repository contains multiple versions of each fuzzer with various modifications. The size of the Polyglot framework is difficult to ascertain, as the bulk of its implementation is a modification to AFL, and the Polyglot authors keep the entire modified copy of AFL in their source tree. Polyglot grammar specifications are given with a mix of JSON specification, Python code, and other formats that are specific to Polyglot.

4.5.2 Fuzzing Dafny

In the summer of 2021, a collaborator began a project using Xsmith to fuzz Dafny (Leino 2010), a verification-aware programming language. He was experienced with Racket but had no previous experience with Xsmith. In about a week, he prototyped his Xsmith-based Dafny generator. During a three-month period, he improved his fuzzer and found 28 Dafny bugs. His fuzzer implementation has 1,666 lines of code, as well as differential testing and verification testing code totaling 722 more lines.

While Dafny fuzzing primarily used differential testing (comparing different Dafny compiler back ends) and compiler error detection, it also included a verification oracle that found one bug.

Additionally, one bug was found as a side-effect of writing the fuzzer. While the author was trying to determine the proper type constraint to write for one feature of the fuzzer, he decided to manually test violations of the constraint, and found a bug.

This experience shows that Xsmith can be utilized to write an effective fuzzer in a small time period with a small amount of code.

4.5.3 Summary of Bugs Found

We have found bugs using Xsmith fuzzers for various programming languages, listed in Figure 4.6. All reported bugs are unique.

Aside from one Racket bug that had been fixed before we found it, all bugs listed are new (not publicly reported or fixed before we discovered them through fuzzing). All Racket bugs were confirmed by Racket's maintainers, and all but one have been fixed. All Dafny bugs and issues were confirmed by Dafny maintainers, and 6 have been fixed. All of the WebAssembly and Standard ML bugs we found have been confirmed and fixed.

We have written fuzzers for various other languages besides those in the bug table, such as Python, JavaScript, and Lua. However, we have not performed extensive fuzzing with them or with the WebAssembly or Standard ML fuzzers. These less-used fuzzers likely all need at least minor improvements to be effective at finding bugs.

Approximately half of the bugs found by Xsmith-based fuzzers so far have been semantic errors detected by differential testing. These bugs can effectively only be found by program generators that reliably generate conforming test cases. Otherwise, if semantically valid but non-conforming (or semantically invalid) programs are regularly generated, differential testing oracles would be overrun with false positive results, and would be practically useless.

Similarly, approximately half of the bugs found could only effectively be found by generators of semantically correct (though not necessarily conforming) program generators. Bugs characterized by valid programs failing to compile would have too many false positives if tested using a generator that does not reliably generate semantically valid test cases. Some bugs found were characterized by a "successful" compilation that produced ill-formed output. Testing for ill-formed output when the compiler is successful could reasonably be performed with generators of semantically or even syntactically invalid code, but such bugs would likely be difficult for such generators to trigger.

In our experiments, Xsmith program generators have not found any crash bugs, the bug class

most commonly found by fuzz testing. Differential fuzz testing with Xsmith appears to be very complementary to other fuzzing practices, such as fuzzing with generators of random bytes such as AFL (Zalewski 2020). Xsmith seems well suited to finding a different class of bugs than tools like AFL, and Xsmith does not effectively stress early compiler stages such as parsing.

4.5.4 Bug Discussion

We present and discuss a small selection of bugs found. The code snippets presented are simplified presentations to illustrate the bugs, not the actual code generated by Xsmith.

4.5.4.1 Racket and Chez Scheme Float Modulo Bug When using a large floating point number, Racket CS (Racket built on Chez Scheme) would give wrong answers to the modulo operator. The bug was found through differential testing.[3]

```
#lang racket/base
;; This number is big enough that despite
;; the .1 it passes `integer?`.
(define num
  584242343082809367481151042431520556233146321109447725441723 2.1)
(println (integer? num)) ;; Prints true
;; But modulo doesn't stay in bounds of the divisor.
(println (modulo num 10))  ;; Prints a number greater than 10
```

It was determined that it was actually a bug in Chez Scheme itself, and was fixed.[4]

4.5.4.2 Racket BC GCD bug This is an example of a bignum boundary bug. The bug was determined to be at least 20 years old, and included in the oldest repository import to CVS. The bug was found by differential testing.[5]

```
#lang racket/base
(define num -4611686018427387904)
;; The gcd function should always return non-negative
;; numbers, but RacketBC returns a negative number.
(gcd num num)
```

4.5.4.3 Racket Serialization Bug Besides differential testing, some bugs are found with various properties. For example, our Racket fuzzer was designed to produce code that does not raise exceptions. Thus, a program that raises an exception is evidence of a bug (in Racket or in our

---

[3]This bug was submitted as https://github.com/racket/racket/issues/3469.

[4]The Chez Scheme bug was fixed in https://github.com/cisco/ChezScheme/pull/537.

[5]Reported as https://github.com/racket/racket/issues/3484.

fuzzer). This bug was present in both Racket BC (the old C back end for Racket) and Racket CS (the new Chez Scheme back end), and was found with an interesting property.

The `write` and `read` functions are primitive serialization and deserialization functions in Racket and Scheme. In version 7.9, the latest release at the time of fuzzing, the handling in the `read` function for Unicode character U+FEFF, the byte-order-mark, was changed. However, the `write` function was not changed. Thus, data could be altered in a round-trip between the `read` and `write` functions.

When printing generated Racket programs, our Racket fuzzer pretty prints them using a function that ultimately uses the `write` function. When compiling our generated programs, the Racket compiler uses the `read` function. The bug was found because the compiler was rejecting ill-formed programs that were mangled between generation and compilation by the `write` and `read` mismatch.[6] We found multiple `write` and `read` mismatch bugs in this manner.

4.5.4.4 Dafny Bugs Related to Zero Multiplicity This class of bugs was found with differential testing. Internally, the multiset data structure in Dafny is implemented as a dictionary mapping from elements to the multiplicity. Many multiset operations assumed the invariant that the multiplicities will always be positive. However, this invariant in fact did not hold, as the multiplicity changing operation can break the invariant. The following code would produce `true false` when compiled to C#, `false true` when compiled to Go, `true true` when compiled to JavaScript, and `false false` (which is the correct output) when compiled to Java.[7]

```
method Main ()
{
  var a := multiset{12}[12 := 0];
  var b := multiset{42};
  print 12 in a, " ", a == b, "\n";
}
```

4.5.4.5 Dafny Bug Found by Verification Testing In addition to differential testing, generated Dafny programs were also used for verification testing, which aims to find soundness and precision issues in the Dafny verifier. Additionally, it is useful for finding discrepancies of the underlying semantics between the verifier and the compilers. Given a generated Dafny program with `print` statements, verification testing compiles and runs the program, and correlates `print`

[6] This bug was reported as https://github.com/racket/racket/issues/3486.

[7] These bugs were reported as https://github.com/dafny-lang/dafny/issues/1359 and https://github.com/dafny-lang/dafny/issues/1361.

statements with their outputs. The program is subsequently transformed to turn the `print` statements into assertions. In the most basic form of verification testing, we expect that these assertions should be verified by the verifier. These assertions therefore test the Dafny verifier's ability to predict the output emitted by the compiled program.

In the following bug, all compiled Dafny programs returned the same incorrect answer, so the bug could not be discovered by differential testing. Yet, it was determined to be incorrect by the verification testing. The problem was that the superset operation was compiled into a subset operation.[8]

```
method Main ()
{
  var a := {1};
  var b := {1, 2};
  print a > b;
}
```

4.6 Discussion

We discuss a qualitative comparison of Xsmith to other systems for creating programming language fuzzers, and discuss Xsmith's limitations.

4.6.1 Comparison to Polyglot

Polyglot (Chen et al. 2021) is a generic language processor that can be configured to produce programs in different languages by providing configuration in a custom format. Polyglot has been very effective at finding crash bugs, finding over 100 bugs in implementations of 9 programming languages. Polyglot uses constrained mutation and a semantic validation step that improves its probability of generating semantically valid programs. These features prevent Polyglot from generating programs with problems such as references to undefined variables, and prevents many type errors. However, these steps do not guarantee semantic correctness. In their evaluation of Polyglot, Chen et al. show that none of their generators produces semantically valid test cases more than 60% of the time. This rate of producing semantically invalid test cases renders it ineffective as a generator for oracles that consistently require semantically valid test cases to prevent false positives, including differential testing.

---

[8]This bug was reported as https://github.com/dafny-lang/dafny/issues/1357. Because the output of the `print` statement is incorrect, when it is turned into an assertion the verifier detects that the assertion is violated, thus revealing the bug.

4.6.2 Comparison to StarSmith

The StarSmith (Kreutzer et al. 2020) program generator is a generic framework for generating semantically correct programs. It is configured using a DSL called LaLa, in which users specify the grammar and other rules for program generation, similar to Xsmith. While StarSmith has no built-in or default steps to prevent undefined or other behaviors that are unsuitable for differential testing, users may write custom LaLa code to prevent some such behaviors. For example, the StarSmith authors created a Lua fuzzer that includes a filter preventing the generation of any events in positions where the order of operations is not guaranteed.

While LaLa makes it possible to create a fuzzer for differential testing, it does not encapsulate common patterns to make it easy. If StarSmith users want to make a similar fuzzer that prevents unspecified effect ordering, they would need to write a similar filter. Additionally, this filter is stricter than Xsmith's generic effect analysis, which can still allow non-conflicting effects when evaluation order is unspecified.

Aside from this observation about encapsulating patterns, it is difficult to compare the suitability of Xsmith and StarSmith for differential testing, since StarSmith's authors expressed a lack of confidence that their programs were actually free of undefined behavior. They reported total instances of test cases uncovering a bug, from fuzzing with conforming configurations, rather than unique bugs found or confirmed as they did for other generators. This makes it difficult to know if they found many bugs or few bugs repeated many times. However, they did report unique bugs for their differential testing of SQL, in which they found 11 semantic bugs and 13 segfault bugs among 3 SQL implementations using a SQL generator of approximately 3500 lines of code in their LaLa DSL. This generator was significantly larger than their other generators. Creating specifications for StarSmith to create conforming program generators is possible, but its focus appears to be on program generators for crash fuzzing.

4.6.3 Limitations of Xsmith

While Xsmith inhabits a new and useful point in the space of fuzzing tools, it has various limitations.

4.6.3.1 Type System Xsmith's type system is flexible but not comprehensive. For example, Xsmith can not currently generate functions with parameters that can be multiple different types but that are not fully generic enough to allow any type, such as a function that accepts either a

string or an integer but no other type. Additionally Xsmith can not generate functions with optional arguments or other forms of variadic functions. Built-in functions with such types can be specified as productions in an Xsmith grammar, allowing Xsmith to generate calls to them. However, for sufficiently complicated types, multiple grammar nodes may need to be specified to allow Xsmith to generate uses of all potential types of a function.

The type system also lacks a notion of classes or objects. While we have implemented structural records with subtyping and nominal records, which can be used to represent some aspects of object-oriented languages, we have not yet decided on a generic system of encoding the semantics of classes and objects that will be well-suited to a variety of programming languages. Because classes, objects, and methods have widely varying semantics in different languages, it is difficult to determine what essential features such an encoding should have.

Another limitation of the type system currently is a lack of "negative types" to constrain type variables. There are often situations where one or more particular types are disallowed but where any other type is allowed. An example situation is a position where functions are disallowed, but base types like `int` and `string` are allowed, as well as composite types such as `list` and `dictionary` composed of other allowed types (perhaps recursively). Rather than allowing users to specify that function types are disallowed, users must enumerate all allowed types. Since the set of allowed types may be infinite, users must realistically only enumerate a small subset of allowed types.

4.6.3.2 Probabilities Xsmith follows a long history of using weight specifications to determine the probability of generating any given grammar production (Sirer and Bershad 1999). Besides weights, the probability of generating any given production is also affected by filters that consider the type system, the effect system, AST depth, and other factors. While Xsmith's weighting system is flexible and allows for dynamic weight determination, it is difficult to determine how any weighting scheme will ultimately affect the probability of generating a particular production or combination of productions.

Xsmith provides a logging mechanism to view the frequency at which different productions were generated or under consideration for generation. However, logging does not provide a guide to understand how to improve a weighting scheme to achieve a more desired probability distribution, nor does it provide insight into what distributions would be effective at finding bugs.

4.6.3.3 Effects The generic effect analysis in Xsmith is both conservative and fairly limited. Because it is conservative, limiting generation to only programs that are guaranteed to be free of unspecified effect ordering, Xsmith filters out many choices that would lead to generating valid, conforming programs. This limitation particularly affects higher-order values. For example, Xsmith has no analysis to determine whether two container values (such as arrays) are the same, so it conservatively assumes that any mutation to a particular container type may conflict with any other access or mutation of that same container type. Because the effect analysis must be generic enough to analyze programs in many languages with varying semantics (most of which is unspecified within an Xsmith fuzzer), the analysis is very limited. Therefore the set of rejected programs is very large, and has great potential to include many interesting and bug-inducing programs.

Xsmith could potentially include a more precise effect analysis by including a way for users to specify language-specific value- and control-flow analyses. However, that would greatly increase the difficulty of writing a new fuzzer, and it is unclear how much it would improve a fuzzer's bug-finding capabilities.

4.7 Related Work

We compare Xsmith to related systems. In particular, we discuss conforming program generators, program generator generators, and grammar-directed fuzzers.

4.7.1 Conforming Program Generators

Some random testers for programming languages generate conforming programs, or programs that are syntactically and semantically valid as well as being free from nondeterminism, undefined behavior, and unspecified behavior. An early major conforming program generator was Csmith (Yang et al. 2011). Similar systems include Verismith (Herklotz and Wickerson 2020) and SQLSmith (Seltenreich 2020). Yarpgen (Livinskii et al. 2020) statically generates programs that are free from undefined behavior with no dynamic checks. Csmith and other conforming program generators have been very successful at finding bugs in programming language implementations. These bugs include semantic bugs found with differential testing that can not be found by more common crash oracles. However, conforming program generators like Csmith require much programmer time and tend to be tens of thousands of lines of code to implement.

Xsmith is a DSL and library for building conforming program generators like Csmith, but with

less time and code. Xsmith eases development of conforming program generators by providing a declarative DSL, along with generic analyses, generation strategies, and other tools as a library. Thus Xsmith allows users to build conforming program generators at a fraction of the cost of stand-alone generators like Csmith.

4.7.2 Program Generator Generators

Polyglot (Chen et al. 2021) is a framework that takes a language specification in a custom format and produces random programs. Polyglot includes a semantic validation step that improves the percentage of syntactically and semantically valid test cases generated. However, programs generated by Polyglot are not guaranteed to be syntactically or semantically correct, or to be conforming. Polyglot has been successfully used to find well over 100 bugs in implementations of at least 9 programming languages. However, it is not a suitable system for finding semantic bugs via differential testing.

StarSmith (Kreutzer et al. 2020) is a framework that takes a language specification in a DSL called LaLa and produces random programs. StarSmith generates programs that are syntactically valid and well-typed. Some StarSmith generators have been further crafted to generate programs that are free of unspecified behavior for differential testing. StarSmith generators that have filters to generate only conforming programs must individually be constrained to not generate offending code. For example, the StarSmith authors created a Lua generator that included custom code to prevent any effects when the order of evaluation isn't guaranteed. Such code would need to be repeated for each generator in StarSmith for languages without guarantees of evaluation order.

PLT Redex (Felleisen et al. 2009) is a framework that allows users to specify a semantics for a programming language. It includes a testing feature that generates random well-typed programs (Fetscher et al. 2015). However, it does not generate programs that are free from undefined behavior. This is useful in Redex, since it helps users to find where undefined behavior exists in their semantics definitions, however it limits its use as a generator for differential testing.

Xsmith has been designed to generate programs suitable for differential testing. Xsmith fuzzers generate programs that are syntactically and semantically valid, in addition to being free from undefined or nondeterministic behavior. Xsmith includes generic effect analyses for the common case of unspecified order of evaluation, allowing programs to include effects while still preserving a well-defined relationship between effects. While some unspecified behaviors need to be handled on

a language-per-language basis, Xsmith's canned components library is helpful with some common patterns.

4.7.3 Grammar-Directed Fuzzers

Some fuzzers, such as Lava (Sirer and Bershad 1999) and Yagg (Coppit and Lian 2005), are grammar-directed (Beyene and Andrews 2012; Godefroid et al. 2008; Hanford 1970; Maurer 1990), meaning they only building program trees that match a given grammar. These fuzzers are useful because they can generate syntactically valid test cases that pass early stages of a compiler or other language processor, allowing fuzzing to exercise deeper code paths. However, grammar-directed fuzzers, without other guidance, do not guarantee that their outputs semantically correct or conforming. Thus they can not reliably be used to find semantic bugs through differential testing.

Some grammar-directed fuzzers such as Grimoire (Blazytko et al. 2019) automatically learn a grammar instead of taking a user-supplied grammar is input. While this automatic learning step means that the fuzzer requires less up-front effort to craft a grammar specification, it provides even weaker guarantees about the syntactic and semantic validity of produced outputs.

Xsmith is grammar-directed in addition to being directed by other analyses, such as type and effect analyses. These analyses constrain program generation to produce conforming programs that are useful for finding semantic bugs.

4.7.4 Parametric Generation

The Zest (Padhye et al. 2019) fuzzer is a grammar-based generative fuzzer that introduces parametric generation. Parametric generation is generation based on a parameter vector. The parameters used to generate an interesting test case may be mutated to cause different but similar test cases to be produced in subsequent generation. Parametric generation provides a way for generative fuzzers to enjoy some benefits of mutational fuzzers, such as feedback-directed guidance. Xsmith also allows Zest-style parametric program generation, though we have not yet evaluated that aspect of Xsmith.

4.8 Conclusion

Differential fuzz testing can be a powerful tool for finding semantic bugs in programming languages. Xsmith is a library and DSL that provides shared infrastructure, declarative specification, and extension hooks that allow users to easily build featureful fuzz testers for differential testing.

Xsmith has been used to create a variety of fuzzers requiring modest effort and code size that have found bugs in different language implementations. Compared to related work, Xsmith is the first tool focused on easily creating fuzzers for differential testing. Xsmith fuzzers can be used synergistically with other fuzzing techniques to find bugs in many language implementations.

4.8.1 References

Michael Beyene and James H. Andrews. Generating String Test Data for Code Coverage. In *Proc. 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012. https://doi.org/10.1109/ICST.2012.107

Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *Proc. 28th USENIX Security Symposium*, 2019. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

Christoff Bürger. Reference Attribute Grammar Controlled Graph Rewriting: Motivation and Overview. In *Proc. 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, 2015. https://doi.org/10.1145/2814251.2814257

Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *Proc. 2021 IEEE Symposium on Security and Privacy (Oakland)*, 2021. https://doi.org/10.1109/SP40001.2021.00071

David Coppit and Jiexin Lian. Yagg: An Easy-to-Use Generator for Structured Test Inputs. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005. https://doi.org/10.1145/1101908.1101969

Pierce Darragh, William Gallard Hatch, and Eric Eide. Clotho: A Racket Library for Parametric Randomness. In *Proc. 2020 Scheme and Functional Programming Workshop*, 2020.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. Semantics Engineering with PLT Redex. MIT Press, 2009.

Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Proc. Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015*, 2015. http://dx.doi.org/10.1007/978-3-662-46669-8_16

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008. https://doi.org/10.1145/1375581.1375607

K. V. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal* 9(4), pp. 242–257, 1970. https://doi.org/10.1147/sj.94.0242

Yann Herklotz and John Wickerson. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proc. 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020. https://doi.org/10.1145/3373087.3375310

Patrick Kreutzer, Stefan Kraus, and Michael Philippsen. Language-Agnostic Generation of Compilable Test Programs. In *Proc. IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2020. https://doi.org/10.1109/ICST46399.2020.00015

K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010. https://doi.org/10.1007/978-3-642-17511-4_20

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4(OOPSLA), 2020. https://doi.org/10.1145/3428264

Peter M. Maurer. Generating Test Data with Enhanced Context-free Grammars. *IEEE Software* 7, pp. 50–55, 1990. https://doi.org/10.1109/52.56422

William M. McKeeman. Differential Testing for Software. *Digital Technical Journal* 10(1), pp. 100–107, 1998.

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In *Proc. Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015*, 2015. http://dx.doi.org/10.1007/978-3-662-46669-8_9

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic Fuzzing with Zest. In *Proc. 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019. https://doi.org/10.1145/3293882.3330576

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-Case Reduction for C Compiler Bugs. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

Andreas Seltenreich. SQLsmith software repository. 2020. https://github.com/anse1/sqlsmith

Emin Gün Sirer and Brian N. Bershad. Using Production Grammars in Software Testing. In *Proc. 2nd Conference on Domain-specific Languages (DSL)*, 1999. https://www.usenix.org/conference/dsl-99/using-production-grammars-software-testing

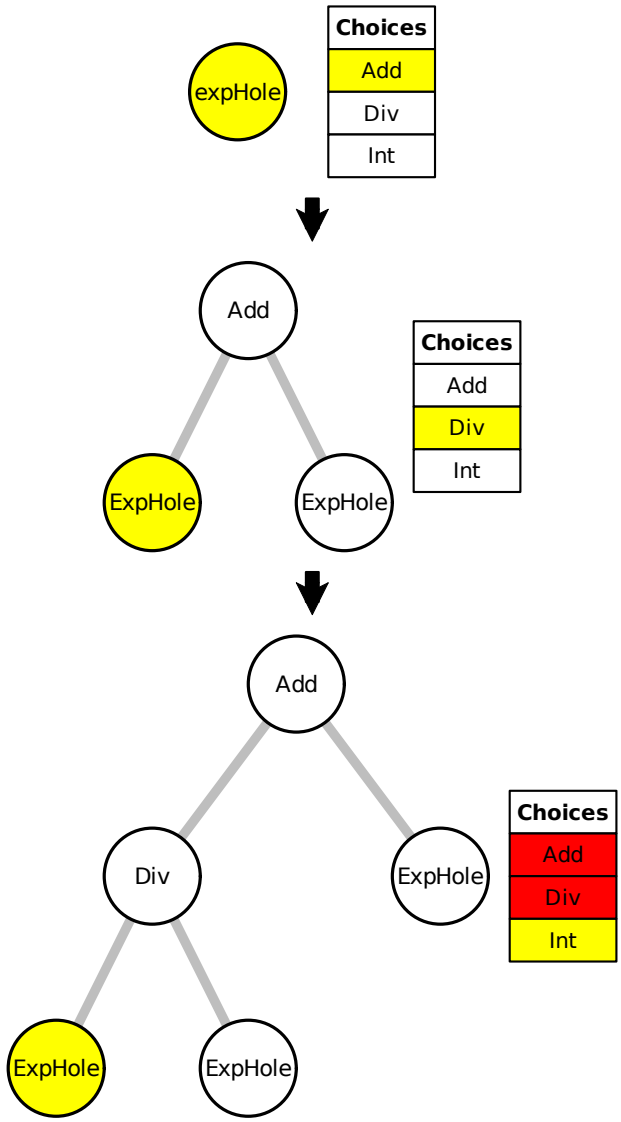
Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011. https://doi.org/10.1145/1993498.1993532

Michał Zalewski. American Fuzzy Lop. 2020. https://lcamtuf.coredump.cx/afl/

4.9 Figures

$$
\begin{array}{lcl}
\langle int \rangle & ::= & z \mid 0 < z < 100 \\
\langle exp \rangle & ::= & \langle exp \rangle \quad + \quad \langle exp \rangle \\
& \mid & \langle exp \rangle \quad / \quad \langle exp \rangle \\
& \mid & \langle int \rangle
\end{array}
$$

Figure 4.1: The Grammar of the `calc` Language



The tree starts as a single `ExpHole` node. Choices that are alternates for `Exp` are listed and one is chosen at random. The process repeats with a new hole until no holes are left. At some points choices are filtered. For example, non-atomic choices are filtered when the tree gets too deep.

Figure 4.2: The Process of Hole Filling

```
;; We use #lang clotho instead of racket for parametric randomness.
#lang clotho
(require xsmith xsmith/canned-components racr pprint ...)

;; An Xsmith specification starts with a "spec-component"
(define-basic-spec-component js-component)

;; Use canned-components to get common grammar definitions.
(add-basic-expressions js-component
                       #:LambdaWithBlock #t
                       #:MutableArray #t
                       ...)

(add-basic-statements js-component
                      #:ProgramWithBlock #t
                      #:IfElseStatement #t
                      ...)

;; Use canned-component loop generator.  It has many options, some elided.
(add-loop-over-container js-component
                         #:name ForLoopOverArray
                         #:loop-ast-type Statement
                         #:body-ast-type Block
                         #:collection-type-constructor
                         (λ (elem-type) (mutable (array-type elem-type)))
                         ...)

;; This header defines safe wrapper operations, and is included
;; when rendering the program.
(define header-definitions-block
  "safe_divide = function(a,b){return b == 0 ? a : a / b} ...")

(add-property js-component render-node-info
              [VariableReference (λ (n) (text (ast-child 'name n)))]
              [SafeDivide
               (λ (n) (h-append (text "safe_divide(") (render-child 'l n)
                                (text ", ")
                                (render-child 'r n) (text ")")))]
              [IfElseStatement
               (λ (n)
                 (h-append (text "if (") (render-child 'test n) (text ")")
                           (render-child 'then n)
                           (text " else ") (render-child 'else n)))]
              ...)

;; This macro defines, among other things, a function to run the
;; command line parser and start generation with the given parameters.
(define-xsmith-interface-functions [js-component]
  #:program-node ProgramWithBlock
  #:format-render (λ (doc) (pretty-format doc 120))
  ...)
```

Figure 4.3: Sample JavaScript generator written with Xsmith

Generator	LOC	Language
Csmith	38,988	C++
Verismith	10,139	Haskell
SQLSmith	3,909	C++
Xsmith Racket Fuzzer	1,265	Racket
Xsmith Dafny Fuzzer	1,666	Racket
Xsmith Standard ML Fuzzer	1,151	Racket
Xsmith WebAssembly Fuzzer	1,433	Racket
Xsmith Python Fuzzer *	1,800	Racket
Xsmith Lua Fuzzer *	450	Racket
Xsmith Javascript Fuzzer *	412	Racket

All line counting was done with Unix `wc`. Fuzzers marked with * have not been exercised in substantial fuzzing campaigns.

Figure 4.4: Comparison of Conforming Program Generators

Generator	LOC	Language
Xsmith Framework	13,325	Racket
Xsmith Racket Fuzzer	1,265	Racket
Xsmith Dafny Fuzzer	1,666	Racket
Xsmith Standard ML Fuzzer	1,151	Racket
Xsmith WebAssembly Fuzzer	1,433	Racket
Xsmith Python Fuzzer *	1,800	Racket
Xsmith Lua Fuzzer *	450	Racket
Xsmith Javascript Fuzzer *	412	Racket
StarSmith Framework	19,524	Java
StarSmith C Fuzzer	1,702	LaLa
StarSmith Lua Fuzzer	1,578	LaLa
StarSmith SQL Fuzzer	$\sim$ 3,500	LaLa
StarSmith SMT Fuzzer	$\sim$ 700-900	LaLa
Polyglot Framework		Mostly C++
Polyglot C Fuzzer	1,508	Mix
Polyglot JavaScript Fuzzer	1,618	Mix
Polyglot PHP Fuzzer	2,013	Mix
Polyglot Solidity Fuzzer	2,090	Mix

Counts prefixed with $\sim$ are approximate. Fuzzers marked with * have not been used in substantial fuzzing campaigns.

Figure 4.5: Comparison of Generic Fuzzing Frameworks

Implementation	Crash Bugs	Semantic Bugs	Static Non-crash Bugs	Total
Racket				
BC	0	5	0	5
CS	0	4	0	4
Both Back Ends	0	2	0	2
Total Racket Bugs				**11**
Dafny				
Java Back End	0	0	8	8
C# Back End	0	3	3	6
Go Back End	0	3	0	3
JavaScript Back End	0	3	1	4
All Back Ends	0	1	7	8
Total Dafny Bugs				**28**
WebAssembly				
Wasmer	0	2	0	2
Standard ML				
ML MLKit	0	0	1	1

Crash Bugs: Bugs characterized by a memory error or assertion violation in the compiler or interpreter causing the system under test to exit abnormally. This is not simply failure to compile an input or an interpreter exiting with an exception.
Semantic Bugs: Bugs characterized by a wrong program result. Found primarily by differential testing, but also by testing various properties. For example, a raised exception when the fuzzer has been constrained not to generate code that raises exceptions.
Static Non-crash Bugs: This category includes various kinds of bugs whose finding would have required syntactically and semantically correct programs but not necessarily conforming programs. For example, this category includes failure to compile correct programs, ill-formed compiler output (eg. Dafny compiler outputting ill-formed Java code that the Java compiler can't compile), etc.

Figure 4.6: Bugs Found With Xsmith-Based Fuzzers

CHAPTER 5

REFLECTIONS

While the chapters of this dissertation are all linked to the theme of embedded DSLs, the motivations and future outlook for each project are distinct.

5.1 Parsing With Delimited Continuations

Chido Parse provides real and meaningful improvements for usability, composability, and extensibility. However, it is currently too slow for use outside of niches where those improvements are of paramount importance and programs are relatively short. The clear next step for parsing with delimited continuations is to find a way to make it faster. Based on experience, it seems that the slow speed is mostly due to the machinery necessary to facilitate ambiguity. Additionally, global ambiguity is usually not required or even desired in programming language parsing. Therefore, finding ways to curtail or remove support for global ambiguity seems like the most promising direction. One possibility is the addition of a Prolog-like *cut*, which commits to a certain result or family of results during parsing. Another possibility is to simply support only limited, local ambiguity.

I hope to make another parsing system that captures delimeted continuations to detect and resolve left recursion, but that removes the caching mechanism of Chido Parse. Instead, I would allow alternate parsers to re-run, with a protocol for determining if a new result supercedes the previous result. Associativity and precidence can be handled with a mixture of result filtering (ruling new results as invalid replacements to a previous result) and a protocol for operator parsers to abort based on the operator used in the left operand. With limited caching, it would be possible for parsers to be re-run many times. However, in practice I believe this would be limited.

Once the performance issues have been solved, I believe that the improvements to parsing expressiveness gained by parsing with delimited continuations will be very useful for implementors of embedded DSLs.

Another future improvement to Chido Parse and successors is to use delimited continuation

capture not only for handling left recursion, but also for adding support for incremental parsing. Incremental parsing allows a parser to be re-run on a modified input (such as when a programmer edits the text of a program) without recomputing the entire parse. In order to simplify the implementation, I did not add support to incremental parsing. However, once a more performant version exists, support for an incremental mode would be a straightforward and useful improvement.

An incremental mode to Chido Parse or similar parsers based on delimited continuations could capture delimited continuations as well as parse derivations to allow incremental parsing. When re-parsing, all derivations that were computed by reading only input to the left of the edit can be reused without change. Derivations that used the edited portion of the input must be recomputed, but that computation can be sped up by using a cached continuation to continue from just before the point of the edit. Derivations from after the edited portion are a little less straightforward, but can still see a benefit. If results from the edited portion of the input do not affect the parameters of the procedure calls that parse the latter segment of the input, then the results from those parse calls may be reused. The one major issue is that input positions (such as line and column count) may be different. If position data is not semantically relevant, then the results may be reused with a simple fix-up phase that edits position data in the output derivations (including potentially editing semantic results such as Racket syntax objects that contain position data for debugging purposes).

5.2 Rash

The enduring popularity of shell languages like Bash show that there is important value in interactive shell languages. However, Bash and similar shells are riddled with limitations and design flaws. In recent years, many new shells have been developed that solve many of those issues. However, unless a shell is embedded in a general-purpose language, it will not be able to gracefully support the entire range of the interactions to scripts to mature programs growth spectrum that many programs traverse.

Previous embedded shell languages embody some of the benefits of Rash, but suffer from various issues, such as supporting only interactions or only scripting, course granularity for host and shell mixing, and lacking a way to recursively embed the host and shell language into each other. Rash provides a new benchmark for embedded shells to measure against for tight integration and extensibility of the embedded shell and host.

Rash's tight integration could be improved by implementing its parser with a system like Chido

Parse. The work on Chido Parse was originally motivated by limitations of Racket's parsing libraries found when writing Rash. Some of the design of Rash's notation worked around these limitations, and could be improved with a more flexible parsing system. But more importantly, integration with other embedded DSLs that use custom notation could be improved by using a more expressive and composable parsing system like Chido Parse.

Beyond integration improvements, Rash still has many limitations. While the language design of Rash cleanly supports the whole interaction-to-program growth spectrum, the features of the interactive shell are lacking. Rash's interactive mode currently uses a primitive line editor with very crude and flawed support for completion and other advanced interaction features. Shells like Zsh and Fish have far better interactive modes due to mature ecosystems of programmatic completion and other editing add-ons. I believe the most important next step for Rash to more completely fulfill its vision is to pair it with a powerful and flexible editor that provides a path to useful completion and other editing features. In addition, language-agnostic completion specifications for programs can help Rash and other young shell languages compete with older shells like Zsh in terms of completion.

Another feature of Rash that is underdeveloped is pipelines that use Racket functions and objects instead of operating system processes and byte streams. While Rash is fully capable of such pipelines, there are currently few functions that provide the functionality that Unix programs provide. This is a similar problem that Microsoft PowerShell has. PowerShell is very useful and powerful when used on Microsoft Windows, because many "commandlets" have been written to provide functionality on the Windows platform. However, PowerShell is much less useful on Unix because commandlets that provide rich object results on Unix have not yet been written. Thus, PowerShell users on Unix must fall back to using Unix processes that communicate via byte streams. If Racket functions that return rich objects are created for system administration in Rash, it could provide much more utility than it can when merely using Unix processes.

Finally, I believe a fruitful direction for embedded shells like Rash is in education. Popular shells like Bash are abtruse and difficult to learn, with strange syntax and semantics for common features such as condition expressions, if statements, and error handling. An embedded shell like Rash could piggy-back on existing programming knowledge easier. While Rash is itself rather complex, a simplified educational version could be created in the same spirit as Racket's student languages.

5.3 Xsmith

Xsmith is an important contribution to the field of fuzz testing because it is the first system to provide a low-effort way to create program generators suitable for differential testing. However, Xsmith has so far been only modestly successful in its goals. In part, this is because our focus in its development was perhaps too much in development rather than usage and evaluation. Xsmith includes various features, such as support for parametric guidance and multi-stage program refinement, that we hope will be helpful but have not yet been seriously used. Additionally, I spent much time creating new fuzzers for different languages despite not having fuzzing environments set up. The process of setting up a fuzzing environment, including installing several implementations of a language, and initial testing of a fuzzer to fix incorrect assumptions in the fuzzer specification, can be a lengthy and somewhat daunting. While setting up a fuzz environment was not very difficult for Racket, a language that I use regularly, it was much more challenging for languages like Standard ML with which I have less familiarity.

The results from fuzzing Dafny show that a programmer who is new to Xsmith but knowledgable with Racket and the fuzzing target can quickly make a useful fuzzer. However, attempts to make Xsmith fuzzers by programmers with less Racket familiarity have so far been slower and less successful. This suggests that a more effective method might be to pair an Xsmith or Racket expert with partners who are an experts with various fuzzing targets. By pairing complementary specialties, it may be easier to more quickly specify a language with Xsmith, build a fuzzing environment, and fix issues that arise when first trying to use a new fuzzer.

APPENDIX

This appendix describes a model implementation of our parsing with delimited continuations algorithm as a literate Racket program. The model has a slightly different API than Chido Parse to facilitate a simpler implementation. For example, the `parse` function is elided since it is a simple wrapper around the `parse*` function. The program text is inset directly between prose paragraphs, starting with the library imports.

```
(require racket/match racket/stream "stream-flatten.rkt")
```

The `"stream-flatten.rkt"` import is just a helper library supplying the functions `flatten-stream` and `stream-flattened?`.

Our two parser primitives are `proc-parser`s and `alt-parser`s. In this model `proc-parser`s contain a procedure that must accept an input string and an offset and return a stream tree. Meanwhile, `alt-parser`s store a list of parsers. Because parsers are often intended to refer to each other mutually recursively, we have an issue for parser construction. We can't refer to a parser that has not yet been constructed while constructing another parser. To "tie the knot," we additionally allow thunks that produce instances of these two structs to be used as parsers.

```
(struct proc-parser (procedure))
(struct alt-parser (parsers))
```

To keep track of parsers and their results, we use `parser-job` objects. In addition to `parser`, `position`, `result-index`, and `result` fields, an additional `worker` field is used to keep track of the state of a job. When first constructed, the `worker` field is `#f`. After a job is started, a worker struct is used to keep track of progress.

The `alt-worker` struct is the only worker used for jobs containing `alt-parser`s, and keeps track of the dependencies that have yet to be have their results included in the results for the worker's job. The `continuation-worker` stores the captured continuations of `proc-parser`s when they are descheduled. The `stream-worker` is used for jobs with result index 1 or higher in a `proc-parser`'s job series. The `stream-worker` stores the stream returned by a previous job so it can be forced to return the next result.

Additionally, we define a `current-job` parameter, which allows procedures to detect the currently running job without explicitly threading it as an argument through all function calls.

```
(struct parser-job
  (parser position result-index [worker #:mutable] [result #:mutable]))
(struct alt-worker ([remaining-jobs #:mutable]))
(struct continuation-worker (k [dependency #:mutable]))
(struct stream-worker (result-stream))
(define current-job (make-parameter #f))
```

Besides a semantic result, `parse-derivation`s store start and end positions to determine where future parses must be performed. Additionally, derivations store a list of sub-derivations and a reference to the parser which created it, which are both useful when filtering derivations for operator precidence and associativity. To ensure that parsers maintain pointer equality even when using thunks as parsers, a cache of parsers is used so thunks are only forced once.

```
(struct parse-derivation (result parser start end sub-derivations))
(define (make-parse-derivation result derivations end)
  (match (current-job)
    [(parser-job parser start _ _ _)
     (parse-derivation result parser start end derivations)]
    [else (error
           'make-parse-derivation
           "Not called during the dynamic extent of parsing...")]))

(define parser-cache (make-weak-hasheq))
(define (canonical-parser p)
  (match p
    [(or (? proc-parser?) (? alt-parser?)) p]
    [(? procedure?)
     (hash-ref parser-cache p (λ () (let ([p* (canonical-parser (p))])
                                      (hash-set! parser-cache p p*)
                                      p*)))]))
```

The `scheduler` object keeps track of all progress made in parsing. We keep a cache mapping input strings to schedulers to ensure we use the same `scheduler` consistently. A `scheduler` stores a cache of jobs keyed by parser, position, and index.

```
(struct scheduler (input-string job-cache))
(define string->scheduler-cache (make-weak-hash))
(define (string->scheduler str)
  (hash-ref string->scheduler-cache str
            (λ ()
              (define s (scheduler str (make-job-cache)))
              (hash-set! string->scheduler-cache str s)
              s)))

(define (make-job-cache) (make-hash))
(define (get-job s parser position result-index)
  (define cp (canonical-parser parser))
  (define cache (scheduler-job-cache s))
  (define key (list cp position result-index))
  (hash-ref cache key
            (λ () (let* ([fresh-job (parser-job cp position
                                                result-index #f #f)])
                    (hash-set! cache key fresh-job)
                    fresh-job))))
(define (get-next-job scheduler job)
  (match job
    [(parser-job parser position result-index _ _)
     (get-job scheduler parser position (add1 result-index))]))
```

Now that we have all of our data structures, it is time to start parsing. When `parse*` is called, we get the target job out of the `scheduler`'s cache and enter the parsing machinery. If this is the outermost call to `parse*`, we will simply start the scheduler. But if we were already running a job, we capture and abort its continuation, storing the continuation and dependency in the job's worker

field. When a continuation is aborted, computation will continue inside the scheduler at a point that we will see later.

```
(define (parse* in-string parser pos)
  (define scheduler (string->scheduler in-string))
  (enter-the-parser scheduler (get-job scheduler parser pos 0)))

(define (enter-the-parser scheduler job)
  (if (not (current-job))
      (run-scheduler scheduler job)
      (let ([parent-job (current-job)])
        (call-with-composable-continuation
         (λ (k)
           (set-parser-job-worker!
            parent-job (continuation-worker k job))
           (abort-current-continuation chido-parse-prompt))
         chido-parse-prompt))))

(define chido-parse-prompt (make-continuation-prompt-tag 'chido-parse))
```

When we run the scheduler, we first check whether our original goal has been completed, and if so, return its result. When there is work yet to be done, we find a runnable job and schedule it. If there is no runnable job in the dependency graph of the goal job, we break a dependency cycle and try again.

```
(define (run-scheduler s goal-job)
  (or (parser-job-result goal-job)
      (let ([next-job (find-work s '() (list goal-job))])
        (if (not next-job)
            (begin (fail-cycle! s goal-job) (run-scheduler s goal-job))
            (schedule-job! s goal-job next-job)))))

(define (find-work s blocked-jobs to-check)
```

```
(and (not (null? to-check))
     (let* ([j (car to-check)])
       (match (parser-job-worker j)
         [(continuation-worker k dependency)
          (cond
            [(parser-job-result dependency) j]
            [else (define new-blocked (cons j blocked-jobs))
                  (define new-to-check
                    (if (memq dependency new-blocked)
                        (cdr to-check)
                        (cons dependency (cdr to-check))))
                  (find-work s new-blocked new-to-check)])]
         [(alt-worker remaining-jobs)
          (cond
            [(null? remaining-jobs) j]
            [(findf parser-job-result remaining-jobs) j]
            [else (define new-blocked (cons j blocked-jobs))
                  (define add-to-check
                    (filter (λ (x) (not (memq x new-blocked)))
                            remaining-jobs))
                  (define new-to-check
                    (append add-to-check (cdr to-check)))
                  (find-work s new-blocked new-to-check)])]
         [(stream-worker _) j]
         [#f j]))))
```

To break a cycle, we perform a depth-first search of the dependency graph until we reach a job that depends on a job that we have already seen. We then mutate its dependency to a fake job whose result is a parse failure.

When no work is available, it would be valid to simply return a parse failure for the goal job instead of bothering with cycle breaking. However, breaking the cycle allows better failure positions and messages to be computed. Additionally, breaking the cycle allows parsing procedures

to tentatively depend on one parser, then fall back on another, as with a biased `or` parser, or have ordered dependencies. However, without a guarantee about cycle breaking order, the result of such parsers is undefined.

```
(define (fail-cycle! scheduler goal-job)
  (define (rec job jobs)
    (match job
      [(parser-job _ _ _ (alt-worker remaining-jobs) _)
       (rec (car remaining-jobs) (cons job jobs))]
      [(parser-job _ _ _ (and w (continuation-worker k dependency)) _)
       (if (memq dependency jobs)
           (set-continuation-worker-dependency! w cycle-breaker-job)
           (rec dependency (cons job jobs)))]))
  (rec goal-job '()))
(define cycle-breaker-job (parser-job #f #f #f #f empty-stream))
```

Scheduling a job behaves differently for `alt-parser`s and `proc-parser`s.

```
(define (schedule-job! scheduler goal job)
  (if (alt-parser? (parser-job-parser job))
      (schedule-alt-job! scheduler goal job)
      (schedule-proc-job! scheduler goal job)))
```

When an `alt-parser` job is scheduled, we simply process the results of its dependencies. When a dependency is successful, a result is copied out of its result stream into the result stream of the `alt-parser`, and the dependency's next sibling job is added to the dependency list. When a dependency has failed (by returning an empty stream), its result is ignored and the dependency is removed. When an `alt-worker` has no more dependencies, the result of its job is an empty stream.

```
(define (schedule-alt-job! scheduler goal job)
  (match job
    [(parser-job (alt-parser parsers) pos 0 #f _)
     (define deps (map (λ (p) (get-job scheduler p pos 0)) parsers))
     (set-parser-job-worker! job (alt-worker deps))]
```

```
    [(parser-job _ _ _ (alt-worker (list)) _)
     (cache-result! scheduler job empty-stream)]
    [(parser-job _ _ _ (and w (alt-worker remaining-jobs)) _)
     (define inner-ready-job (findf parser-job-result remaining-jobs))
     (define result (parser-job-result inner-ready-job))
     (define other-remaining-jobs (remq inner-ready-job remaining-jobs))
     (if (parse-failure? result)
         (set-alt-worker-remaining-jobs! w other-remaining-jobs)
         (let ([this-next-job (get-next-job scheduler job)]
               [dep-next-job (get-next-job scheduler inner-ready-job)])
           (set-alt-worker-remaining-jobs!
            w (cons dep-next-job other-remaining-jobs))
           (cache-result! scheduler job result)
           (set-parser-job-worker! this-next-job w)
           (set-parser-job-worker! job #f)))])
  (run-scheduler scheduler goal))
```

When a `proc-parser` job is scheduled, we execute its procedure, wrapped with the `do-run!` helpers. If the procedure has never been run, we simply apply it. If the procedure previously returned a stream result, we force the stream to find more results. If a job was previously descheduled by a recursive call to `parse*`, we resume the job by applying its continuation to the result of its dependency.

```
(define (schedule-proc-job! scheduler goal job)
  (match job
    [(parser-job (proc-parser proc) pos 0 #f #f)
     (do-run!/thunk scheduler goal job
                    (λ () (proc (scheduler-input-string scheduler)
                                pos)))]
    [(parser-job _ _ _ (stream-worker result-stream) #f)
     (do-run!/thunk scheduler goal job
                    (λ () (stream-rest result-stream)))]
```

```
    [(parser-job _ _ _ (continuation-worker k dependency) #f)
     (do-run!/continuation scheduler goal job k
                           (parser-job-result dependency))]))
```

The `do-run!/thunk` procedure wraps the thunk with the `chido-parse-prompt` after also delimiting `parse*-direct`.

```
(define (do-run!/thunk scheduler goal job thunk)
  (define result
    (call-with-continuation-prompt
     (λ ()
       (parameterize ([current-job job])
         (delimit-parse*-direct (thunk))))
     chido-parse-prompt
     abort-handler))
  (result-check-loop scheduler goal job result))
```

When running a continuation, we do not need to apply `delimit-parse*-direct`, since the `parse*-direct-prompt` is still in the captured continuation. But we do need to re-wrap the continuation with the `chido-parse-prompt`.

```
(define (do-run!/continuation scheduler goal job k k-arg)
  (define result
    (call-with-continuation-prompt
     k chido-parse-prompt abort-handler k-arg))
  (result-check-loop scheduler goal job result))
```

When we get a result from either `do-run!/thunk` or `do-run!/continuation`, we check the result with `result-check-loop`. When a procedure delimited with the `chido-parse-prompt` is aborted, the result given to the `result-check-loop` is a unique `recursive-enter-flag` value. When we get the `recursive-enter-flag`, we can simply jump back to `run-scheduler`, since the abort code in `enter-the-parser` has already saved the continuation to a `continuation-worker`.

When we get an actual result from the job's procedure, it must be a stream tree which we will (lazily) flatten into a stream which contains only non-stream elements. While stream flattening

is mostly lazy, `flatten-stream` must force the stream tree until it finds a non-stream result or finds the flattened stream to be empty. Because our flattening procedure forces part of the stream, flattening is also wrapped with the `chido-parse-prompt`. We must loop back to the initial check with the flattened result in case forcing the stream causes recursion, aborting the continuation and returning another `recursive-enter-flag`. In any case, the `result-check-loop` finishes by calling back into `run-scheduler`.

```
(define (result-check-loop scheduler goal-job job result)
  (cond [(eq? result recursive-enter-flag)
         (run-scheduler scheduler goal-job)]
        [(and (stream? result)
              (not (flattened-stream? result))
              (not (stream-empty? result)))
         (result-check-loop scheduler goal-job job
                            (call-with-continuation-prompt
                             (λ () (parameterize ([current-job job])
                                     (delimit-parse*-direct
                                      (stream-flatten result))))
                             chido-parse-prompt
                             abort-handler))]
        [else (begin (cache-result! scheduler job result)
                     (run-scheduler scheduler goal-job))]))

(define (abort-handler) recursive-enter-flag)
(define recursive-enter-flag (gensym 'recursive-enter-flag))
```

When we cache results, we take an element out of a result stream and wrap it in a new stream whose `stream-rest` causes computation to re-enter the scheduler. For jobs with `proc-parser`s, we additionally save the result stream in a `stream-worker`.

```
(define (cache-result! scheduler job result)
  (match result
    [(? parse-failure?) (set-parser-job-result! job result)]
```

```
    [(? stream?)
     (define next-job (get-next-job scheduler job))
     (when (proc-parser? (parser-job-parser next-job))
       (set-parser-job-worker! next-job (stream-worker result)))
     (set-parser-job-result!
      job
      (stream-cons (stream-first result)
                   (enter-the-parser scheduler next-job)))]))

(define (parse-failure? x) (and (stream? x) (stream-empty? x)))
```

Finally, we need a form for parsing procedures to conveniently loop over parse results. The `parse*-direct` procedure simply captures its current continuation up to the `parse*-direct-prompt`, then aborts the continuation. The `delimit-parse*-direct` function applies the `parse*-direct-prompt` with the default abort handler, which accepts a thunk to call after aborting the continuation. Therefore, `parse*-direct` replaces its continuation with a thunk that lazily applies the captured continuation to each parse result using `for/parse`.

```
(define (parse*-direct in-string parser pos)
  (call-with-composable-continuation
   (λ (k)
     (abort-current-continuation
      parse*-direct-prompt
      (λ () (for/parse ([d (parse* in-string parser pos)])
                       (k d)))))
   parse*-direct-prompt))

(define-syntax-rule (delimit-parse*-direct e)
  (call-with-continuation-prompt (λ () e) parse*-direct-prompt))
(define parse*-direct-prompt
  (make-continuation-prompt-tag 'parse*-direct))
```

```
(define-syntax-rule (for/parse ([arg-name input-stream]) body)
  (let loop ([stream input-stream])
    (cond [(stream-empty? stream) stream]
          [else (let ([arg-name (stream-first stream)])
                  (stream-cons body
                               (loop (stream-rest stream))))])))
```

While this model implementation does not have practical performance characteristics, it is almost as expressive as the full implementation. Using `proc-parser` and `alt-parser` as a base, users could write and compose ad hoc procedural parsers, parser combinators, BNF parser constructors, and other parsing abstractions. This model supports all context-free grammars, including ambiguous and left-recursive grammars, dynamic extensibility, data-dependent grammars, and other non-context-free grammars.