Rash: From Reckless Interactions to Reliable Programs

William Gallard Hatch
University of Utah
USA
william@hatch.uno

Matthew Flatt
University of Utah
USA
mflatt@cs.utah.edu

Abstract

Command languages like the Bourne Shell provide a terse syntax for exploratory programming and system interaction. Shell users can begin to write programs that automate their tasks by simply copying their interactions verbatim into a script file. However, command languages usually scale poorly beyond small scripts, and they can be difficult to integrate into larger programs. General-purpose languages scale well, but are verbose and unwieldy for common interactive actions such as process composition.

We present Rash, a domain-specific command language embedded in Racket. Rash provides a terse and extensible syntax for interactive system administration and scripting, as well as easy composition of both Racket functions and operating system processes. Rash and normal Racket code can be nested together at the expression level, providing the benefits of a shell language and a general-purpose language together. Thus, Rash provides a gradual scale between shell-style interactions and general-purpose programming.

CCS Concepts • **Software and its engineering → Command and control languages; Macro languages; Domain specific languages; Scripting languages**;

Keywords Domain specific language, shell, macros

ACM Reference Format:
William Gallard Hatch and Matthew Flatt. 2018. Rash: From Reckless Interactions to Reliable Programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3278122.3278129

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 [18, 20]. 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 [13], Reticulated Python [22], and Typed Racket [21] 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

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

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 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 [19] 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.

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 [2]. 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 3.2). 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 [9], 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 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.

3.1 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/
}
```

3.2 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.

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 [14] 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 subtrees, 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.

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 4.2.

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-lines` 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.

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 4.4.

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 4.3 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 [7] 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.

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.

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 [17] and Plumbum [6] 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 [19] 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.

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 [15]. 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.

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.

6.1 Object Pipelines

PowerShell [19] 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.

6.2 Stand-Alone Shells

Several projects including Oil Shell [3] and Elvish [24] 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.

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 [17] for Scheme, Caml-Shcaml [12] for Ocaml, Turtle [10] and Shelly [23] for Haskell, Plumbum [6] 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.

6.4 Shells Embedded in General-Purpose Languages

Eshell [8] 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 [16] 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 [11] 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 [4] 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 [1] (based on Perl) and Closh [5] (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.

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.

Acknowledgments

The authors would like to thank Michael Adams, Robby Findler, Eric Eide, and the anonymous reviewers for their feedback on the paper. This work was supported by the National Science Foundation through grant number CNS-1526324.

References

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

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

[3] Andy Chu. Oil. 2018. https://www.oilshell.org/

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

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

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

[7] 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.

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

[9] Free Software Foundation. GNU Bash. 2018. https://www.gnu.org/software/bash/

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

[11] Li Haoyi. Ammonite Documentation. 2018. http://ammonite.io/

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

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

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

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

[16] Anthony Scopatz. Xonsh Documentation. 2018. http://xonsh.org/

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

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

[19] Jeffrey P. Snover. Monad Manifesto. Microsoft, , 2002.

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

[21] 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.

[22] 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.

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

[24] Qi Xiao. Elvish. 2018. https://elv.sh/