Herbarium Racketensis: A Stroll through the Woods (Functional Pearl)

VINCENT ST-AMOUR, DANIEL FELTEY, SPENCER P. FLORENCE, SHU-HUNG YOU, and ROBERT BRUCE FINDLER, PLT @ Northwestern University, USA

Domain-specific languages are the ultimate abstraction, dixit Paul Hudak. But what abstraction should we use to *build* such ultimate abstractions? What is sauce for the goose is sauce for the gander: a language, of course!

Racket is the ultimate abstraction-abstraction, a platform for quickly and easily building new ultimate abstractions. This pearl demonstrates Racket's power by taking a leisurely walk through the implementation of a DSL for Lindenmayer systems, the computational model par excellence of theoretical botany.

CCS Concepts: •**Software and its engineering → Extensible languages; Domain specific languages;**

Additional Key Words and Phrases: Racket, Lindenmayer systems

ACM Reference format:
Vincent St-Amour, Daniel Feltey, Spencer P. Florence, Shu-Hung You, and Robert Bruce Findler. 2017. Herbarium Racketensis: A Stroll through the Woods. *PACM Progr. Lang.* 1, 1, Article 1 (September 2017), 15 pages.
DOI: 10.1145/3110245

1 LINDENMAYER SYSTEMS

Lindenmayer systems (Lindenmayer 1968) (or *L-systems* for short) are a domain-specific language in the truest sense of the term. Aristid Lindenmayer was a theoretical biologist—the archetypal domain expert—studying plant growth. Lindenmayer needed a language in which to express his models, and thus developed one whose model of computation mirrors plant growth.

A Lindenmayer system consists of an initial string (or *axiom*), and a set of rewriting rules. The system evolves by applying the rewriting rules to the axiom, applying them again to the result, and so on. Whereas rewriting rules in an operational semantics or a context-free grammar are applied one at a time, rules in a Lindenmayer system are *all* applied in parallel, at each step of the computation. Plants do not grow one leaf at a time; many parts all grow at once.

```
## axiom ##
X

## rules ##
F -> FF
X -> F-[[X]+X]+F[+FX]-X
```

Fig. 1. A branch and its Lindenmayer system (Prusinkiewicz and Lindenmayer 1990)

DOI: 10.1145/3110245

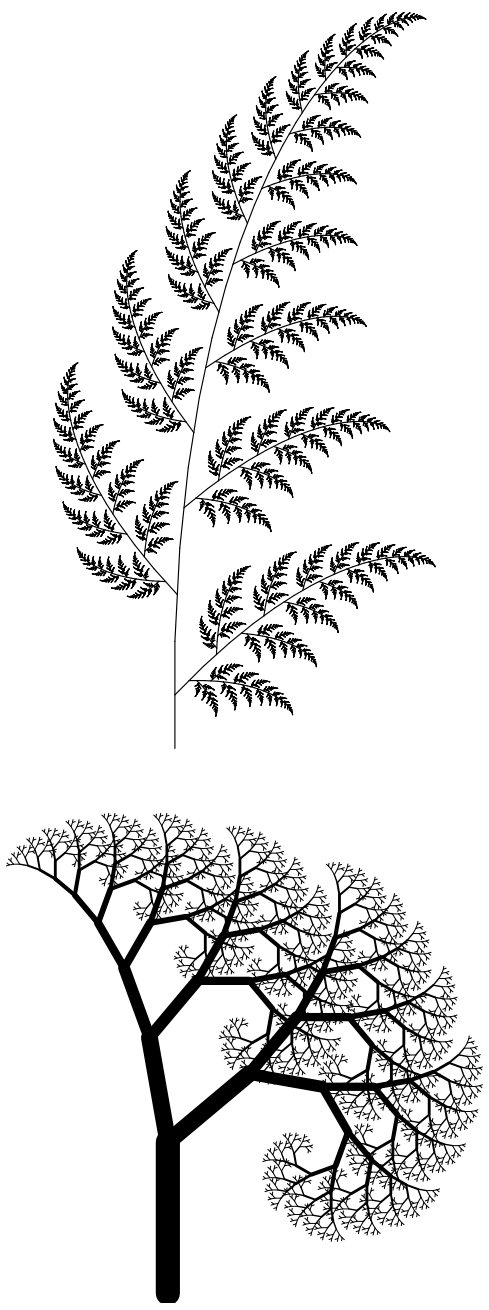


Fig. 2. A fern and a tree generated by Lindenmayer systems (Kurth 2007)

To make the computational model concrete, consider a Lindenmayer system that models the growth of algae. It has the axiom `A` and two rewriting rules: `B` → `A` and `A` → `AB`. The first rule captures a juvenile alga becoming an adult and the second represents an adult asexually reproducing. After one step, we get the string `AB`, rewriting the axiom with the first rule. Next, we have two opportunities to rewrite and we take them both, using each rule once to produce the string `ABA`. In the third step, we can apply the first rule to the two `A`s in the string and the second rule to `B`, and this produces the string `ABAAB`.

By associating each character in the string's alphabet to a semantic action, we obtain Lindenmayer systems that produce various kinds of output. For example, we can obtain the picture of a branch in figure 1 by adding LOGO-like turtle graphics operations as semantic actions to one of Lindenmayer's plant growth models. As is standard in the L-systems world, we associate `F` with an action that moves the turtle forward while drawing a line, `+` and `-` with ones that rotate the turtle in opposite directions, and `[` and `]` with actions that save and restore the turtle's state.

From Botany to Language. From our canopy flyby of L-systems, we can tell that any Lindenmayer system DSL must be able to express axioms, rules, and semantic actions. Furthermore, a truly lush DSL must also provide a pleasant interface for authors, informative error checking, safe escape hatches to a general-purpose language, as well as tooling to support software development and evolution.

For the rest of this pearl, we set out on a walk through the forest of Racket (Felleisen et al. 2015) using the implementation of this Lindenmayer system DSL as our guide. Along the way, we will stop at various clearings with nice vistas on Racket's design and engineering, which makes possible each of the above DSL aspects. As we stroll, take a moment to admire the foliage in the figures, all of which is generated by Lindenmayer systems implemented in our DSL. The botanical wonders we encounter on our hike will demonstrate that Racket makes writing effective DSLs easy, and—dare we say—breathtaking.

2 TRAILHEAD: COMPUTING LINDENMAYER STRINGS

We begin our journey with an overview of the computational engine that underlies our Lindenmayer system DSL. From a DSL technology perspective, there is not much to see yet. DSL implementations, however, are programs first of all, and thus depend on a modern programming language, with all the amenities. Before beginning our trek into the heart of DSL technology, we must pick a good starting point.

Racket—and Typed Racket (Tobin-Hochstadt et al. 2017) in particular—provides just the seed we need. It supports all the sophisticated constructs programmers have come to expect, including higher-order functions, mutable state, types, and parametricity, along with novel features like occurrence typing (Tobin-Hochstadt and Felleisen 2010). The rest of this section explains how our Lindenmayer system computational engine is nourished by these roots.

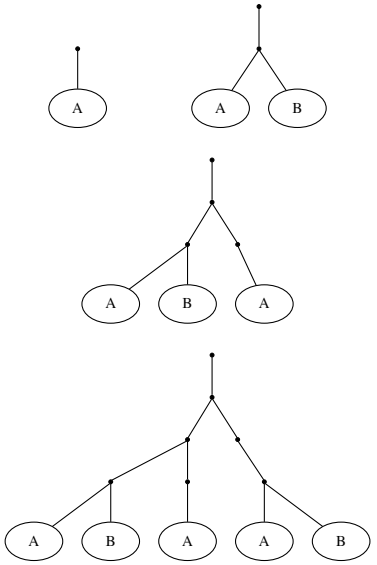


Fig. 3. Trees, where successive generations each add a new layer of leaves

2.1 The Lay of the Land

A straightforward approach to implementing a Lindenmayer system is to explicitly keep the current string in a linear, sequential data structure and to scan it at each step, applying the rules in turn. Unfortunately, this process takes time proportional to the size of the current string, which typically doubles in each step.

Instead, as befits our domain, think of the state of the system as a tree, where we apply the rewriting rules at the leaves to generate a larger tree. Figure 3 shows the first four steps of this process for our example Lindenmayer system.

Simply keeping this tree at each step does not improve on the linear data structure, but the tree can be represented much more compactly, taking advantage of sharing. More precisely, at each step we can use the same new leaf for each new `A`. Figure 4 shows the DAGs that correspond to the trees in figure 3 for our Lindenmayer system. An in-order traversal of the DAGs visits the same leaf nodes in the same order as an in-order traversal of the trees, but extending one DAG to the next requires applying each of the rules only once.

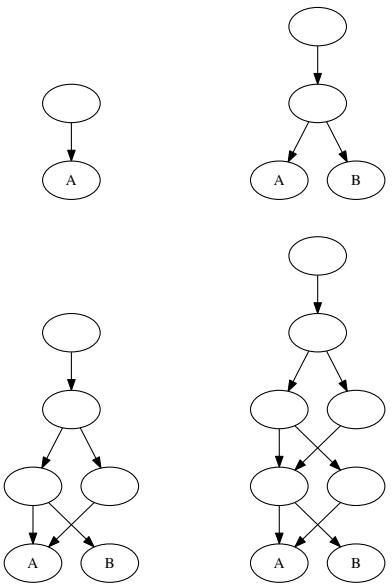


Fig. 4. DAGs, representing figure 3, but with sharing

```
1 #lang typed/racket
2
3 (struct (α) cell ([item : (Exp α)])
4   #:mutable)
5 (define-type (Exp α)
6   (U (-> α α)
7      (Listof (cell α))))
8
9 (: run-lindenmayer
10    (All (α)
11         (-> Natural (cell α)
12             (cell α) (cell α)
13             (-> (cell α) (cell α)
14                 (Listof (cell α)))
15             (-> (cell α) (cell α)
16                 (Listof (cell α)))
17             α
18             α)))
19 (define (run-lindenmayer iterations
20                          axiom
21                          nt1 nt2
22                          rule1 rule2
23                          init)
24   (for/fold ([nt1 nt1][nt2 nt2])
25             ([i (in-range iterations)])
26     (define nt3 (cell (cell-item nt1)))
27     (define nt4 (cell (cell-item nt2)))
28     (set-cell-item! nt1 (rule1 nt3 nt4))
29     (set-cell-item! nt2 (rule2 nt3 nt4))
30     (values nt3 nt4))
31   (collect axiom init))
32
33 (: collect (All (α) (-> (cell α) α α)))
34 (define (collect axiom init)
35   (define current init)
36   (let loop ([ele (cell-item axiom)])
37     (cond
38       [(list? ele)
39        (for ([ele (in-list ele)])
40          (loop (cell-item ele)))]
41       [else
42        (set! current (ele current))]))
43   current)
44
45 (provide run-lindenmayer)
```

Fig. 5. Runtime code

2.2 A Sapling

The code in figure 5 implements the essence of this idea, and is a distilled version of the engine behind our DSL, pruned to only accept exactly two non-terminals (`nt1` and `nt2`) and two rules (`rule1` and `rule2`). How the precise Typed Racket constructs in this code fit together is not essential, but let us discuss the key parts.

The functions `run-lindenmayer` and `collect` match the two phases of the process, namely building up the DAG and then traversing it. The accumulators in the `for/fold` loop on line 24, `nt1` and `nt2`, hold the current leaves. Lines 26–27 build the new leaves for the next iteration and lines 28–29 update the old leaves based on the given rules. Line 30 returns the new leaves as the new accumulators for the next loop iteration.

The `collect` function is similarly direct: the local `loop` iterates over the DAG, ignoring the sharing and uses functions stored in the leaves to collect a result in the variable `current`.

The first argument to `run-lindenmayer` is the desired number of iterations. The second is the axiom, represented as a `cell`. The third and fourth arguments specify the non-terminals and the fifth and sixth specify the right-hand sides of the two rules.

The right-hand sides of the rules are given as functions whose input is a complete set of non-terminals (both of them, in this case) and whose output is the particular non-terminals on the right-hand side of the rule. That is, each function selects from the available non-terminals, returning the ones that the corresponding rule uses.

The last argument to `run-lindenmayer` is the initial value of an accumulator which is passed through the the leaf node functions in the order in which they appear in the string; the result of the final symbol in the string is the result of `run-lindenmayer`.

3 1st CLEARING: A PLEASANT INTERFACE

Calling into the computational engine directly is fairly complex and subtle. For example, if we were to encode our earlier algae growth system, we need these 11 lines:

```
(let ([A (cell (λ ([x : (Listof Symbol)])
                 (cons 'A x)))]
      [B (cell (λ ([x : (Listof Symbol)])
                 (cons 'B x)))])
  ((inst run-lindenmayer (Listof Symbol))
   4                      ; iterations
   (cell (list A))        ; axiom
   A B                    ; non-terminals
   (λ (A B) (list A B))   ; rule 1 (A)
   (λ (A B) (list A))     ; rule 2 (B)
   '())) ; initial value for result
```

DSLs, as the ultimate abstractions, really ought to isolate their users from the complexity of their computational underbrush. Racket allows the programmer to extend the language with a new construct, `lindenmayer`, that is translated to a call to `run-lindenmayer`. In Racket terminology, `lindenmayer` is a "syntax transformer" (Dybvig et al. 1992) that compiles this expression

```
(lindenmayer 4
             (A)
             (A -> A B)
             (B -> A))
```

into the complex call to `run-lindenmayer` above.

The transformer is mostly straightforward. It is written using a pattern language (Culpepper and Felleisen 2010) that treats ellipses specially:

```
#lang racket
(require (for-syntax syntax/parse)
         "run-lindenmayer.rkt")

(define-syntax (lindenmayer stx)
  (syntax-parse stx
    [(_ iterations
        (X:id ...)
        (Y:id -> Z:id ...) ...)
     #'(let ([Y (cell
                 (λ (l) (cons 'Y l)))]
             ...)
         (run-lindenmayer
          iterations
          (cell (list X ...))
          Y ...
          (λ (Y ...) (list Z ...)) ...
          '()))]))

(provide lindenmayer)
```

Fig. 6. Two Lindenmayer systems that share rewrites for leaves and branching, but generate the overall plant structure differently (Prusinkiewicz 1986a)

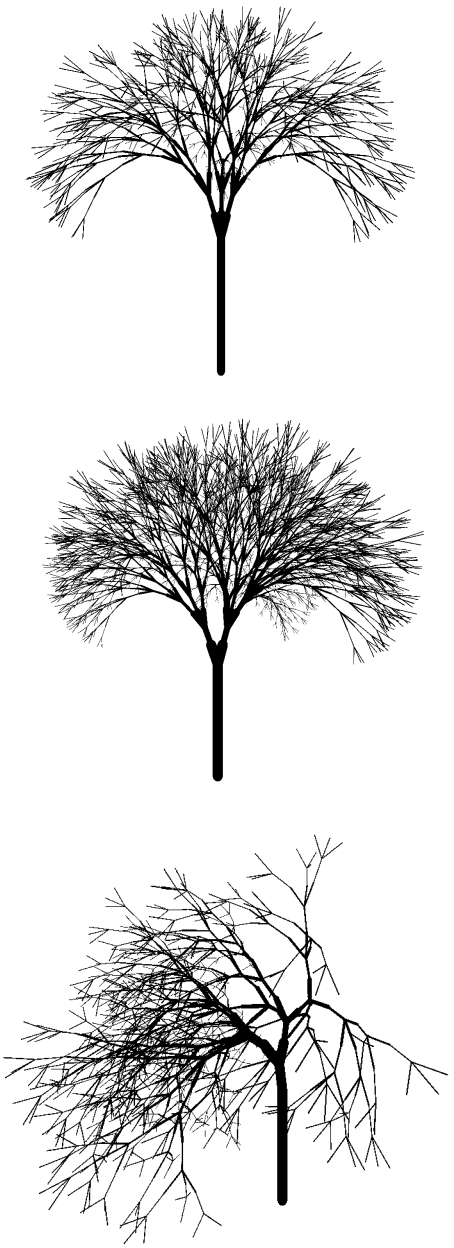


Fig. 7. Three trees generated by the same Lindenmayer system from Aono and Kunii (1984), with different parameters

In this case, the left-hand side of the syntax pattern rewrite is `(_ iterations (X:id ...) (Y:id -> Z:id ...) ...)` and it is used to match against the concrete syntax in our use of `lindenmayer`. The annotation `:id` rejects the match unless `X`, `Y`, and `Z` are identifiers. Accordingly, the pattern variable `X` binds to a sequence of one identifier containing only the concrete variable `A`. The two layers of ellipses around `Z` indicate that it is bound to a sequence of sequences, namely the right-hand sides of each of the rules.

The rewrite produces exactly the same code as we wrote in the example call to `run-lindenmayer`. The transformer, however, guarantees that the result is well-formed. In particular, no inputs to the transformer can result in a ill-typed call to `run-lindenmayer` and there is also no way to generate a call that supplies functions that misbehave, e.g. use continuations or signal errors. Note that the use of ellipses ensures that the non-terminal arguments and the rule arguments to `run-lindenmayer` line up *by construction*; the first rule argument will necessarily be the one corresponding to the first non-terminal, and so on.

From here it is a simple step to write a parser that accepts non-parenthesized concrete syntax and inserts parentheses in the right places, turning the following program into the one above (`n` as the number of iterations is common in the Lindenmayer system literature):

```
1 #lang lindenmayer
2
3 ## axiom ##
4 A
5
6 ## rules ##
7 A -> AB
8 B -> A
9
10 ## variables ##
11 n=4
```

The first line declares the language that the rest of the file uses. The part which follows the `#lang` delimiter must refer to a Racket module that defines (or re-exports) the three components of a Racket language: a reader, a set of transformers, and runtime support functions. A *reader* is

a function from input streams to S-expressions, which possibly relies on a lexer and parser. A language's *transformers* are responsible for compiling the language's syntactic forms to existing, more primitive forms, which can be handled either by another language or by the Racket compiler itself. Finally, *runtime support functions* provide functionality that programs in the language can use when executing. The Racket runtime system simply looks up the module referred to by the `#lang` line, then leaves[1] the rest to the language's implementation, which makes sense of the rest of the file and produces a module from its contents. Languages are therefore free to branch out from the default Racket language as far as they want.

Our earlier example starts with `#lang lindenmayer`, which causes the Racket runtime system to look for the reader, transformers, and runtime support of the `lindenmayer` language. Its reader calls out to a parser for the above syntax, whose details are quite thorny and un-pearl like, and thus omitted here. The `lindenmayer` transformer above is the principal transformer provided by the language. Finally, section 5 elaborates on the runtime support portion of the language. As expected, running this file iterates the Lindenmayer system four times, which produces the output ABAABABA.

4 2nd CLEARING: BETTER ERROR CHECKING

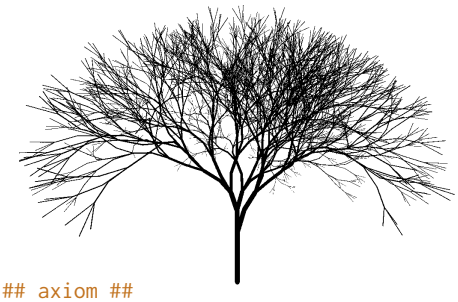
As languages—DSLs included—become richer, the potential for errors in programs in those languages increases. As such, it is absolutely vital that languages provide their users with early and accurate error detection. Thankfully, Racket's syntax system (Flatt 2002) provides considerable support for error checking.

Therefore, as we enrich our DSL to support more powerful models—parameterized models, stochastic models, conditional models, etc.—we also introduce novel classes of errors programmers can make. For example, parameterized Lindenmayer systems bring in the notion of arguments, and with them that of arity errors.

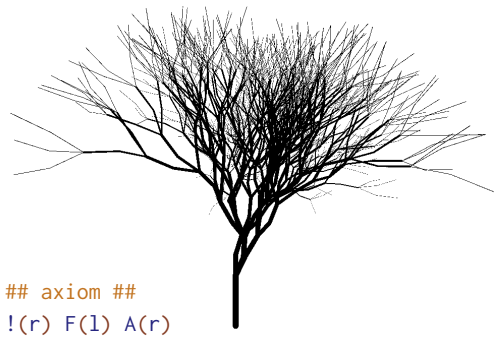
[1] I walnut allow puns to proleaferate. They arboring. –Robby

⟨program⟩ ::= `#lang` ⟨identifier⟩ ⟨anything⟩+

Fig. 8. The grammar for valid Racket programs

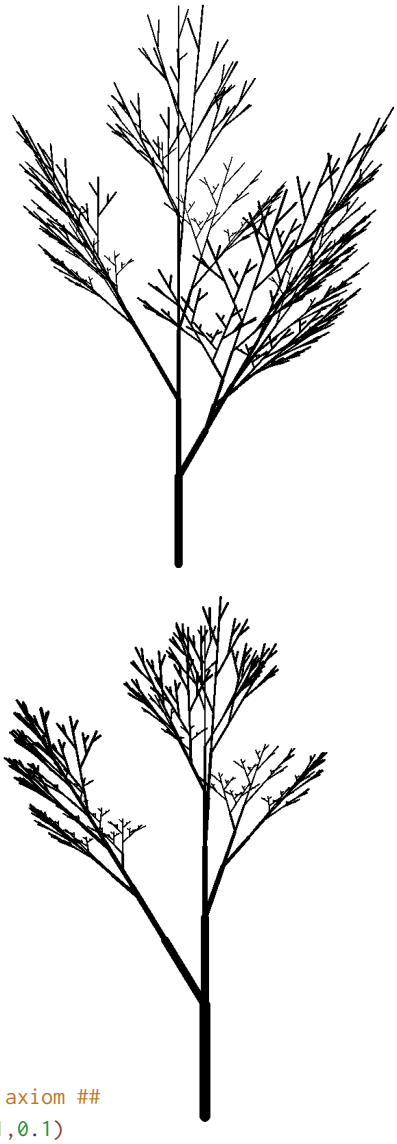


```
## axiom ##
!(r) F(l) A(r,l)

## rules ##
A(r,l) -> ^F(l)  B(r,l)
          >(θ*3) B(r,l)
          >(θ*4) B(r,l)
B(r,l) -> [!(r) ^F(l) >(θ*2) A(r*s,l*v)]
```



```
## axiom ##
!(r) F(l) A(r)

## rules ##
A(r) -> ^F(l)  B(r+s)
        >(δ*3) B(r+s)
        >(δ*5) B(r+s)
B(w) -> [!(w) ^^F(l) >(δ*6) A(w)]
```

Fig. 9. Trees based on the Lindenmayer systems at http://www.geekyblogger.com/2008/04/tree-and-l-system.html



```
## axiom ##
A(1,0.1)
## rules ##
A(l,w) -> !(w)F(l)[&(a)B(l*s,w*y)]
          /(d)A(l*r,w*y)
B(l,w) -> !(w)F(l)[-(b)$C(l*s,w*y)]
          C(l*r,w*y)
C(l,w) -> !(w)F(l)[+(b)$B(l*s,w*y)]
          B(l*r,w*y)
```

Fig. 10. A parameterized Lindenmayer system and its output for different values of `r`, `a`, and `b` (Aono and Kunii 1984; Prusinkiewicz and Lindenmayer 1990)

In a parameterized Lindenmayer system, the symbols carry values (typically numbers) that can change as the system evolves. Syntactically, parameter lists are surrounded with parentheses and their elements are separated by commas, much like the mathematical notation for function application. Parameter lists are written in a separate arithmetic language meaning, e.g., that the context determines if a `+` is rotating the turtle (when it appears outside of a parameter list) or addition (when it appears inside a parameter list).

Figure 10 contains an example. In that system, the symbols `A`, `B`, and `C` each carry two values; the `F`, `+`, `-`, `!`, `&`, and `/` symbols each carry one value; and `]`, `[`, and `$` carry zero.

The `A`, `B`, and `C` symbols are non-terminals, so their definitions accept two arguments, which can be used when computing the parameter values on the right-hand sides of the rules.

The others are terminals, so they accept an appropriate number of arguments and the values are recorded alongside the symbol for use by the semantic actions.

The runtime support for our parameterized Lindenmayer system implementation consists of functions, one for each symbol, whose arity is determined by the arity of the corresponding symbol. Accordingly, that runtime support signals confusing errors about its internal state if the arities are not used consistently.

To avoid those internal errors, we can design a syntax transformer that behaves just like Racket's function definition form behaves, except that it statically checks the arity of all uses of the function and disallows uses that are not directly in the function position of an application. The implementation of this transformer illustrates an interesting technique, showing how Racket's syntax transformation system relies deeply on lexical scope (Flatt 2016).

We call our new syntactic form `define/arity` and an example use is

```
(define/arity (f x y)
  (sqrt (+ (* x x)
           (* y y))))
```

In this example, `define/arity` transforms itself into these two definitions:

```
1 (define-syntax (f stx)
2   (syntax-parse stx
3     [(use-site-f x y)
4      #'(f-proc x y)]
5     [(use-site-f actual ...)
6      (signal-length-error
7       #'(actual ...)
8       #'(x y)
9       #'f #'use-site-f)]))
10 (define (f-proc x y)
11   (sqrt (+ (* x x) (* y y))))
```

The first case of the `syntax-parse` pattern matching form (lines 3–4) matches when there are two arguments and it transforms directly into a call to `f-proc`. The other case (lines 5–9) is a fall-through case which does not produce any result syntax; instead it calls `signal-length-error` which raises a syntax error.

Because `f` must do compile-time checking of its arity, it must be able to inspect its call sites. As functions are incapable of doing so, `f` must be a transformer, which means `define/arity` must itself be a *transformer-defining transformer* (Dybvig et al. 1992).

The implementation of `define/arity` in figure 12 makes its transformer-defining nature readily apparent. On line 9, a new transformer is defined *within the generated code* and given the name that was passed in to `define/arity`, e.g., `def-f`. The generated transformer is responsible for calling a function (line 12) whose body does the actual computation (lines 18–19) if the arity at the use site is correct, and throwing an arity error otherwise (lines 13–17).

Two particular aspects of this code are especially interesting. First, the identifier `f-proc` is the same for all uses of `define/arity` and yet it has the desired scope. The transformer works properly because each use of `f-proc` comes from a different use of `define/arity` and thus the scope of `f-proc` is tied to the particular newly introduced transformer. The syntax system keeps each `f-proc` private, even if `define/arity` is used multiple times in the same scope.

Second, because the definition of `define/arity` operates at two levels of transformers simultaneously, it is crucial to distinguish what belongs to

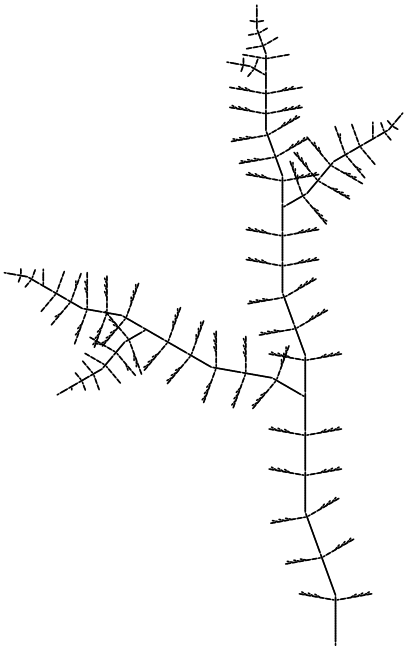


Fig. 11. Rendering of A. Tenuissimum, generated by a Lindenmayer system (Corbit and Garbary 1993)

```
1 #lang racket
2 (require (for-syntax syntax/parse))
3 (provide define/arity)
4
5 (define-syntax (define/arity stx)
6   (syntax-parse stx
7     [(_ (def-f formal:id ...) e)
8      #`(begin
9          (define-syntax (def-f stx)
10           (syntax-parse stx
11             [(use-f formal ...)
12              #'(f-proc formal ...)]
13             [(use-f actual (... ...))
14              (signal-length-error
15               #'(actual (... ...))
16               #'(formal ...)
17               #'def-f #'use-f)]))
18         (define (f-proc formal ...)
19           e))]))
```

Fig. 12. The implementation of the `define-arity` transformer-defining transformer

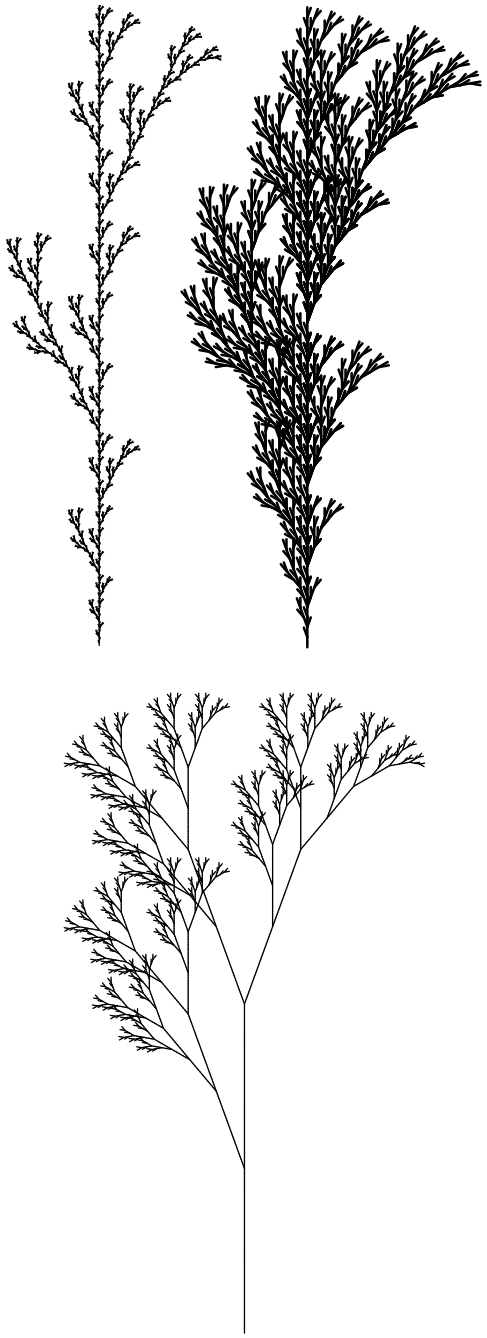


Fig. 13. Plant-like Structures from Lindenmayer systems in Prusinkiewicz (1986b) and Prusinkiewicz and Lindenmayer (1990)

each level. In particular, when using the `...` syntax to express repetition, it is important to specify at which level the repetition is happening. Repetition at the `define/arity` level is expressed using the usual `...` syntax, whereas repetition inside the template of the generated transformer must escape the repetition notation and use the `(... ...)` notation for its own repetitions. Thus, in the first step of transformation, we replace `formal ...` with the precise arguments `x` and `y` in the example, but we leave `actual` as it is instead of expanding it into a concrete set of variables, so that the inner `define-syntax` can match any number of actual arguments and properly signal an error.

5 3rd CLEARING: ESCAPE HATCH TO RACKET

With what we have seen so far, our DSL can express most of the components of a Lindenmayer system, namely axioms, rules, and variables, but semantic actions are missing. Semantic actions are necessary to produce meaningful outputs from the strings generated by the computational engine. Because the nature of these outputs varies based on the domain of each Lindenmayer system—flora (Prusinkiewicz and Lindenmayer 1990), musical compositions (Manousakis 2006), street maps (Parish and Müller 2001), buildings (Müller et al. 2006), etc.—there is no hope in having our DSL itself provide support for all of them directly.

What is needed is a way to fall back to a general-purpose programming language, to handle each different kind of output. As such, our DSL provides a way to access the greater Racket ecosystem. And because Racket is Racket, this really is a multiplexed escape hatch to any of the languages in the Racket ecosystem: Racket itself, Typed Racket, Datalog, Lazy Racket (Barzilay and Clements 2005), Scribble (Flatt et al. 2009), or even POP-PL (Florence et al. 2015).

Racket's basic unit of interoperability is a module. Unlike SML's module system, a Racket module is a compilation unit (and layers like functors are built on top of Racket modules (Flatt and Felleisen 1998)). Each file that starts with `#lang` compiles to a module and each module in Racket

may export a set of names. A module may bring another module's exported identifiers into scope via `require`. In general, different languages export different classes of values. Nevertheless, languages can interoperate by exporting common classes of values and then document how such exports work in terms of lower-level languages.

Most languages in the Racket ecosystem cooperate in some way with a well-known set of low-level values, but the precise details vary from language to language. For example, in `#lang typed/racket`, a function compiles into a raw Racket function (Tobin-Hochstadt et al. 2011) that is wrapped with a contract. In `#lang lazy`, functions are also compiled into the primitive notion of a function, but the arguments and results may have to be promises.

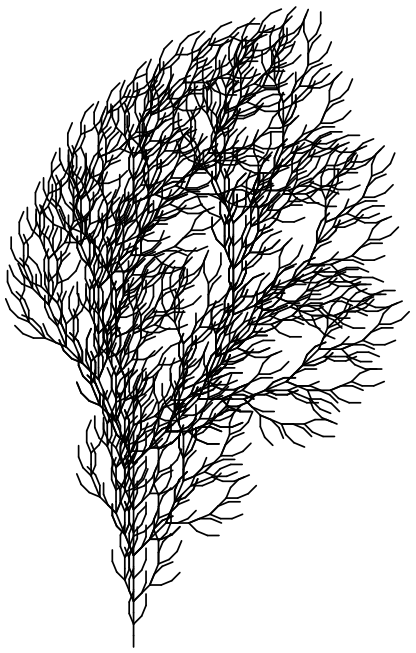
For our Lindenmayer language, we expect that the language we interoperate with uses Racket's primitive notion of functions and that we can call them using `#lang racket/base` (a basic, stripped down Lisp-like language)'s notion of application.

Concretely, figure 14 shows this interoperability in action. The first line specifies that the program is in the `lindenmayer` language, but now there is also a second language specified. That language is used for any text following the separator formed by equal signs (lines 14 and after).

As described in section 3, every `#lang` language must provide a reader function that consumes a stream of input and returns a raw S-expression, which is then transformed into a module. The implementation of `#lang lindenmayer` uses this mechanism, of course, but also piggybacks on Racket's definitions to find the appropriate function to construct the S-expression that corresponds to the content after the separator, transforming it into a module. It then places the module whose body follows the separator as a submodule (Flatt 2013) of the `#lang` module and uses `require` to extract bindings for all the symbols in the Lindenmayer system (in this case `A` and `B`), as well as `start` and `finish`. The variables for the symbols and `finish` are expected to be functions that are each called with two arguments. The first argument is the current state of

```
1  #lang lindenmayer typed/racket
2
3  ## axiom ##
4  A
5
6  ## rules ##
7  A -> AB
8  B -> A
9
10 ## variables ##
11 n=20
12
13 ====================================
14
15 (provide (all-defined-out))
16
17 (: start (-> (HashTable Symbol Real)
18              (Pair Natural Natural)))
19 (define (start variables)
20   (cons 0 0))
21
22 (: finish (-> (Pair Natural Natural)
23                (HashTable Symbol Real)
24                Real))
25 (define (finish pr variables)
26   (/ (car pr) (cdr pr)))
27
28 (: A (-> (Pair Natural Natural)
29          (HashTable Symbol Real)
30          (Pair Natural Natural)))
31 (define (A pr variables)
32   (cons (+ (car pr) 1)
33         (cdr pr)))
34
35 (: B (-> (Pair Natural Natural)
36          (HashTable Symbol Real)
37          (Pair Natural Natural)))
38 (define (B pr variables)
39   (cons (car pr)
40         (+ (cdr pr) 1)))
```

Fig. 14. Lindenmayer system that computes the golden mean using the algae system from section 1



```
#lang lindenmayer racket

## axiom ##
F

## rules ##
F -> FF-
     [-F+F+F]+
     [+F-F-F]

## variables ##
n=4
θ=22.5

==================================
(require lindenmayer/turtle)
(provide
 (all-from-out lindenmayer/turtle))
```

Fig. 15. Example tree from Prusinkiewicz (1986b)

traversal of the Lindenmayer string and the second one is a key-value data structure that contains the information from the `## variables ##` section. The `start` function accepts only the key-value data structure and it returns the `init` argument of `run-lindenmayer`.

Returning to figure 14, because the Lindenmayer system has two non-terminals, `A` and `B`, the `typed/racket` program must export functions named `A` and `B`, as well as `start` and `finish`. First `start` is called, which produces a pair of zeros. Then `A` and `B` are called some number of times depending on the string. These functions increment the natural number in their corresponding component and then `finish` computes their ratio.

Replacing this code with code that calls into Racket's turtle graphics library provides access to the standard drawing toolkit for Lindenmayer systems. For example the plant in figure 15 was produced by running the code in that figure. Because the symbols used in that Lindenmayer system are `F`, `+`, and `-`, as well as the square brackets, the code below the separator must export identifiers with those names. In this case, these are standard names used by many Lindenmayer systems to control the turtle in a LOGO-like manner, so our library contains a simple adapter layer on top of a LOGO graphics library that is part of the standard Racket distribution.

The astute reader will notice that the language which the `lindenmayer` language interoperates with must support identifiers whose names are open and close square brackets, which may seem gnarly. Luckily Racket's identifier syntax is flexible enough to accommodate. And if it were not, we could design a new language that was a slight change to Racket that did (or use a different symbol in the Lindenmayer system).

6 4th CLEARING: TOOLING

DSLs are first and foremost Ls, that is languages. As easy as it may be to forget as programming language researchers, languages are much more than semantics and type systems, or even interpreters and compilers. Programming tools are but one of the other necessary pieces that make languages truly usable and useful.

Therefore, for our Lindenmayer system DSL to truly live up to programmer expectations, it should support the kinds of tools programmers rely on when authoring and evolving programs: code intelligence, renaming, refactoring, etc.

Racket, as the ultimate abstraction-abstraction, makes it almost effortless to add tooling support to DSLs. By virtue of compiling to Racket, and at the modest cost of additional meta-information, Racket-based DSLs get to reuse the considerable tooling support that already exists for Racket.

Consider figure 16, which shows the program for the tree in figure 15 inside DrRacket. Because the symbols in the Lindenmayer program compile into Racket variables in `run-lindenmayer` expressions, DrRacket (Findler et al. 2002) sees them as bound and binding occurrences and is able to correlate them with the original source locations in the program. These arrows also inform DrRacket's renaming functionality, allowing the user to rename all of the `F`s to some other symbol.

Renaming comes for free, simply because of the compilation process, and because of the support already built into DrRacket. DrRacket also provides a platform for other tools that require only a little bit of work. For example, the coloring of the text in the window required us to implement a function that accepts an input stream and returns the first token in the stream and its coloring information. DrRacket can then use that function to support an efficient, incremental syntax colorer. The concurrency required to support interactivity is implemented only once as a part of DrRacket, and each language can (optionally) plug in language-specific lexeme information to enable the coloring.

DrRacket also supports the ability to add arbitrary other tools, using a more complex interface. We used this interface to add a refactoring tool for Lindenmayer systems. A user can select a range of text in the right-hand side of a rule and DrRacket will create a new non-terminal that expands into the selected text, replacing it with a reference to the new non-terminal. This change is not semantics-preserving, but can sometimes be a useful device to delay specific parts of the system relative to others.

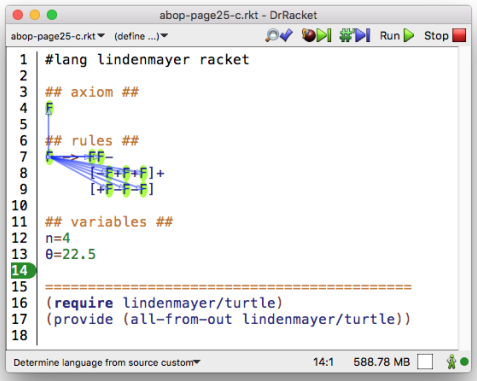


Fig. 16. DrRacket editing the program in figure 15

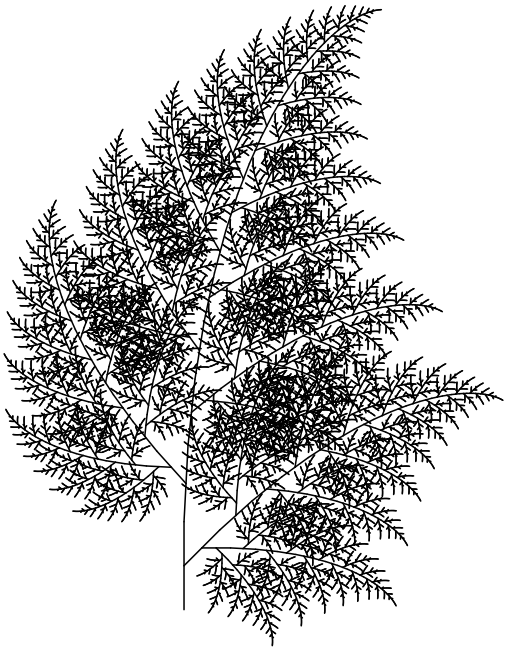


Fig. 17. Fern leaf from Kurth (2007)

```
## axiom ##
X

## rules ##
F -> FF
X -> F-[[X]+X]+F[+FX]-X
```

```
## axiom ##
X

## rules ##
Y -> [X]+X
X -> F-[Y]+F[+FX]-X
F -> FF
```

Fig. 18. Before and after refactoring

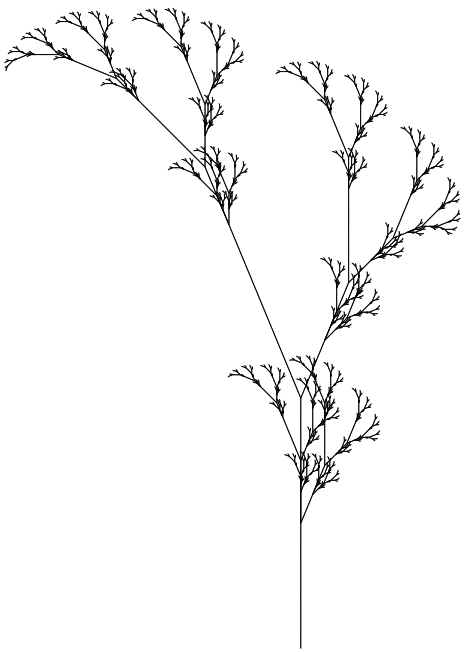


Fig. 19. A refactored version of the tree in figure 1

P.S. One of the Lindenmayer system pictures shows a plant that is not found in a forest. Did you spot it?

For example, the refactoring tool can introduce the `Y` non-terminal into the Lindenmayer system from figure 1 to rewrite the top portion of figure 18 to the bottom portion, which generates the tree in figure 19 (running for two more iterations than the original). The structure of the refactored tree shows how the region of the system that corresponds to `Y` affected the overall drawing. That portion is now delayed, relative to the rest.

The tool cooperates with the implementation of `#lang lindenmayer`. Its compilation process adds metadata for each symbol in each rule that records its name and its source location. Then, when the user selects a region inside a rule, the tool can map backwards to ensure that the selection is indeed in the right-hand side of the rule.

Each of these tools requires more programming effort than the one before, but even the most complex took only two days of programmer effort, considerably less than would be required to implement similar support in another editor or on another platform, because they leverage Racket's language-building support.

7 EMERGING FROM THE FOREST

Our woodland stroll following the implementation of our Lindenmayer system DSL has taken us through a series of clearings, from which we could admire the power, beauty, and majesty of Racket's language-building facilities.

Most of this beauty comes not from any particular design choice—from any particular grove, thicket, or coppice—but rather from how all the pieces fit together—every leaf, branch, and tree. Racket's entire gestalt aligns to make it a root and branch abstraction-abstraction.

Acknowledgements. We would like to thank the anonymous reviewers for helping us prune overgrown prose and water wilted explanations, as well as providing us with a flowering bouquet of additional botanical puns.

Return for a self-guided tour. To explore on your own, our implementation is available online (including the source code for all of the Lindenmayer systems in the paper, and more) at:
https://github.com/rfindler/lindenmayer/

REFERENCES

Masaki Aono and Tosiyasu L. Kunii. Botanical tree image generation. *IEEE Computer Graphics and Applications* 4(5), 1984.

Eli Barzilay and John Clements. Laziness without all the hard work: combining lazy and strict languages for teaching. In *Proc. Functional and Declarative Programming in Education (@ICFP)*, 2005.

John D. Corbit and David J. Garbary. Computer simulation of the morphology and development of several species of seaweed Using Lindenmayer systems. *Computers & Graphics* 17(1), 1993.

Ryan Culpepper and Matthias Felleisen. Fortifying macros. In *Proc. ICFP*, pp. 235–246, 2010.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), pp. 295–326, 1992.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket manifesto. In *Proc. SNAPL*, 2015.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *JFP* 12(2), pp. 159–182, 2002.

Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. ICFP*, pp. 72–83, 2002.

Matthew Flatt. Submodules in Racket: you want it when, again? In *Proc. GPCE*, pp. 13–22, 2013.

Matthew Flatt. Bindings as sets of scopes. In *Proc. POPL*, pp. 705–717, 2016.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: closing the book on ad-hoc documentation tools. In *Proc. ICFP*, pp. 109–120, 2009.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. PLDI*, pp. 236–248, 1998.

Spencer Florence, Burke Fetscher, Matthew Flatt, Tina Kiguradze, Dennis P. West, Charlotte Niznik, Paul R. Yarnold, Robert Bruce Findler, and Steven M. Belknap. POP-PL: A patient-oriented prescription programming language. In *Proc. GPCE*, pp. 131–140, 2015.

Winfried Kurth. Specification of morphological models with L-systems and relational growth grammars. *Journal of Interdisciplinary Image Science* 5, 2007.

Aristid Lindenmayer. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology* 18, pp. 280–315, 1968.

Stelios Manousakis. *Musical L-Systems*. MS dissertation, The Royal Conservatory, The Hague, 2006.

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Proc. Conference on Computer Graphics and Interactive Techniques*, pp. 614–623, 2006.

Yoav I H Parish and Pascal Müller. Procedural generation of cities. In *Proc. Conference on Computer Graphics and Interactive Techniques*, pp. 301–308, 2001.

Przemyslaw Prusinkiewicz. Applications of L-systems to computer imagery. In *Proc. International Workshop on Graph Grammars and Their Application to Computer Science*, 1986a.

Przemyslaw Prusinkiewicz. Graphical applications of L-systems. In *Proc. Proceedings of Graphics Interface / Vision Interface*, 1986b.

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. The Algorithmic Beauty of Plants. Springer Verlag, 1990.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proc. ICFP*, pp. 117–128, 2010.

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory typing: Ten years later. In *Proc. SNAPL*, 2017.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. PLDI*, pp. 132–141, 2011.