Speaking for the Trees: a New (Old) Approach to Languages and Syntax

by

Moss Prescott

B.A., University of California Berkeley, 2004

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2011

This thesis entitled:
Speaking for the Trees: a New (Old) Approach to Languages and Syntax
written by Moss Prescott
has been approved for the Department of Computer Science

Jeremy Siek

Bor-Yuh Evan Chang

Clayton Lewis

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Prescott, Moss (M.S., Computer Science)

Speaking for the Trees: a New (Old) Approach to Languages and Syntax

Thesis directed by Assistant Professor Jeremy Siek

Programmers in many fields are interested in building new languages. In the same way that high-level languages increase productivity by allowing programmers to accomplish more with a given amount of code, a special-purpose language can reduce repetitive code and hide implementation detail, making a program's structure and function more evident. Sometimes an entire new language is called for, and other times an existing language serves most needs but could benefit from a few additional elements. However, current tools support this kind of extension poorly.

Language tools typically represent programs internally as trees which are easily extended with new types of nodes. However, in today's languages the programmer only indirectly manipulates this tree—through a parser which analyzes free-form text (that is, concrete syntax) and builds an abstract syntax tree (AST). The limitations of parsers make languages difficult to extend and severely constrain the choice of notation. In other words, what one writes and reads is dictated by what the parser is able to handle.

I explore an alternative approach: represent source code directly as an AST and derive both an executable program and a readable presentation from it. I present a flexible representation for ASTs, a general mechanism for transforming these trees, and a language for grammars which allows concrete syntax and semantics to be defined via these transformations. I show that this approach is modular, easy to understand, and expressive enough to define novel syntax and semantics.

My prototype system, Lorax, demonstrates the new approach. Reductions for presentation and execution are written in a functional language with meta-programming features. Syntax is not limited to simple text but may include richer notation for easier reading and understanding. A structure editor renders this rich syntax, using algorithms from TEX. In Lorax, the barrier to entry for the creation of languages is lowered, making it practical for programmers to express solutions

in the terms and the notation which are closest to the problem domain.

Dedication

To PJ and Kevin.

Contents

Tables

Figures

Chapter 1

Introduction

Many kinds of programmers are interested in building new languages. General-purpose languages evolve slowly, but even so a dominant language like Java undergoes change over the years, adding new concepts and syntax [**?**]. New computer architectures spawn new languages [**?**][**?**][**?**], software engineers talk about how and when to build and use Domain Specific Languages [**?**], and of course programming language researchers are continually designing new languages to use or to analyze. Yet the technology that is currently used to construct programming languages does not support these kinds of creative activities very well, and often the lack of good tool support is a major factor discouraging the creation and use of a new language.

The design and implementation of languages, undertaken by a relatively small group of "language vendors," is often seen as a separate task from the creation of programs by the much larger population of language users [**?**][**?**]. However, another approach, often identified with the Lisp community, views the task of writing a program as combined with the building of a language into one activity [**?**]. These two perspectives go along with different tool sets, and the large disparity in goals suggests an opportunity exists to combine some of the strengths of both approaches in a single environment.

Furthermore, computer language technology could potentially be used in many more contexts. For instance, any class of structured documents (say recipes, essays, math problems) could be represented with a language defining the required elements and their relationships to each other. The tools used to create these documents are often far more user-friendly and visually rich than

programming tools, but they fail to identify the underlying structure of the documents, trapping the data in a panoply of closed formats which obscure the valuable content. If the tools programmers use to create and work with languages offered better expressivity and were simpler to employ, software for creating many kind of documents could be built on similar foundations.

1.1 Existing Approaches

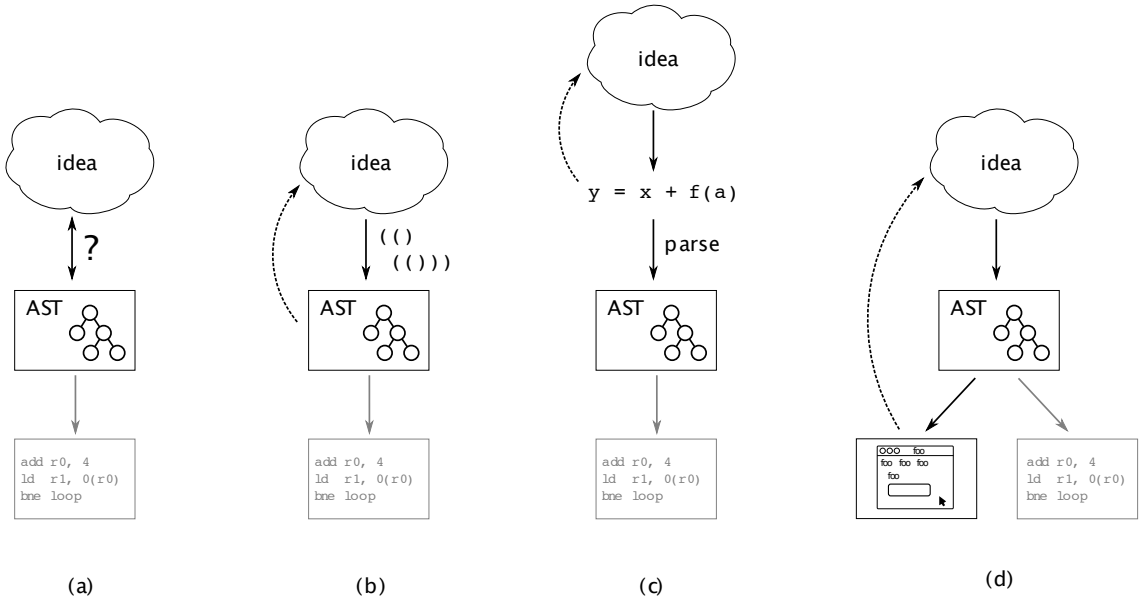


Figure 1.1: The problem of turning an idea into a program, and three different solutions.

Figure **??**(a) illustrates the basic situation. A programmer has an idea for a program she wants her computer to run. To accomplish that, she needs to put the image in her mind into a form that can be understood by her computer. Since the advent of structured programming, virtually all programming languages have represented programs in terms of computational constructs arranged into a hierarchical structure. This tree-shaped structure, often called an Abstract Syntax Tree (AST), allows an infinite variety of programs to be constructed out of a small, fixed number of constructs, and provides an unambiguous representation for programs which can be consumed by tools such as compilers and interpreters. The programmer, on the other hand, may or may not think of her program in this way, and in any case she must translate the idea in her head into an

AST, and then in a separate effort construct that AST within the memory of her computer.

Figure **??**(b) shows the simplest solution. The programmer describes the AST directly in a form that requires little or no interpretation. For example, she can write a Lisp program in s-expressions, using parentheses to explicitly indicate the nested structure of her program. In doing so, she actually engages in a process of action and feedback. She types a bit, perhaps attempts to run the program, reviews the program's source text (really only a thinly-veiled AST), and then elaborates on or revises the program. The programmer is able to inspect the AST almost directly, but has to do so by counting parentheses (dashed arrows in the figure show the path of feedback from the computer to the programmer).

These languages are easy to extend because introducing a new construct is accomplished by simply defining a new symbol, but the extra effort required to read them is off-putting to the overwhelming majority of programmers. This has lead to a decades-long cycle of proposals to "fix" the syntax and counter-argument [**?**].

Instead, the majority use the model diagrammed in Figure **??**(c), where another representation stands between the programmer and the AST. Now she enters her program as free-form text, which is interpreted by a parser as directions for constructing the AST. This allows for a more "natural"-appearing syntax, but it also inserts a level of indirection between the programmer and her program. The feedback loop of idea to program and back now goes from mind to *text* and back, without ever involving the AST. In fact, the AST is now a mere by-product of the source text.

The parser imposes both limits and costs. Because the textual syntax acts as both interface and specification, a tension is created between economy of expression (to serve the needs of the programmer) and precision (to allow a parser to construct the right AST). An economical syntax will often involve local ambiguity, because leaving some things unsaid which can be inferred from context is a hallmark of human communication. Furthermore, if language extension is to be an integral part of the construction of software, it is inevitable that different programmers working on different parts of a system will create language elements that are superficially similar, and that other programmers will need to combine these elements together.

In the presence of these ambiguities, parsers have to become context-sensitive to some extent. One solution is a custom parser—such as those commonly used to parse languages like Java and C—which is designed to handle a particular language and not easily extended. Therefore, these languages tend to provide a single fixed syntax and the cost to change it is very high—you may have to convince a standards body to incorporate your idea, and then wait for compiler vendors to implement it. The other is a more sophisticated parsing algorithm, which may be able handle local ambiguity, but the design of such parsers is an ongoing research problem, with significant performance costs [**?**].

A third approach is *structure editing*. In Figure **??**(d), the programmer operates on the AST, with both presentation and editing mediated by an interface which explicitly understands the structure of the language. The first advantage of this setup is that the program's source is represented in the form of an AST, is edited at the level of the AST, and is always presented back to the user in terms of meaningful AST nodes. It's no longer possible for the programmer to *read* the program one way while the parser disagrees. Secondly, the visual representation of the program—as well as the programmer's method of interacting with it—are not constrained to lines of text. The editor is free to supply more sophisticated layout, rendering, and forms of interaction.

Yet this power comes at a high cost. To construct a structure editor for a complete language is a large effort which may or may not justify its cost [**?**]. Worse still, structure editors are too specialized and too limiting to be embraced by real working programmers, who have accepted the weaknesses of the parser-driven approach and have grown dependent on a universe of text-based tools. Thus structure editors find use only in certain niches, where the benefits outweigh the perceived costs (e.g. end-user programming, CS education) [**?**].

1.2 A New Way

Each of the three approaches just described makes a fundamental trade-off between the conflicting goals of readability, expressive power, and extensibility. This thesis strikes a new balance, showing that by giving up textual source code, a new way of building languages can combine the

expressive power of Lisp with the best syntax you can devise. I present a unified framework for constructing languages, editors, and compilers based on two simple ideas: all source code and derived objects are represented from the start as ASTs, and source is transformed into derived trees via reductions written in a simple, extensible, functional, meta-language.

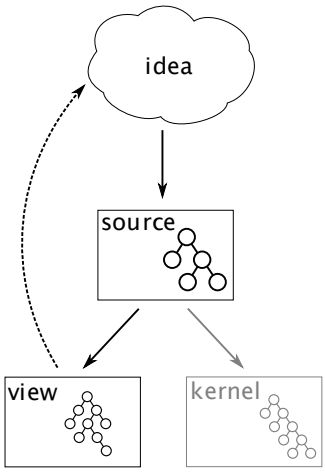


Figure 1.2: Lorax

Figure **??** shows the high-level structure of Lorax's editing and execution model. Both the executable form of a program and its visual representation are defined in terms of ASTs (labeled *kernel* and *view*). The translation from the source form to these derived forms is specified in a modular way, with the same tools used to construct core languages and extensions. The programmer operates on the source AST in a relatively direct way, but this interaction is mediated by the "view" transformation, which presents the program in a familiar and understandable form.

The benefits of this approach are:

(1) When constructed this way, a structure editor can be flexible enough to work with programs in multiple languages constructed in a dynamic process and freely embedded within one another.

(2) Defining new language constructs or even entirely new languages is as simple as writing a program in an existing language, and the resulting languages are fully supported by the

editing environment. Thus the barrier to entry for creating new languages is significantly reduced, and this approach may become a more routine part of the process of writing programs.

(3) Freed from the constraints of parsing, more sophisticated presentation becomes easy to achieve. The tools prototyped so far can achieve excellent readability for typical programming constructs, but also support rich, familiar, mathematical notation.

1.3 Related Work

Two recent projects have some of the same goals, and share the fundamental idea of moving to a tree-based representation for source code. Intentional Software is working on a tool based on Charles Simonyi's Intentional Programming work from the 1990s [**?**], but little detail has been made public [**?**]. The system appears to be targeted to large teams made up of *domain experts* who are knowledgable in some field, and *domain engineers* who are expert programmers, tasked with developing languages for the domain experts to use. Despite the difference in specifics, the Intentional Programming manifesto from 1995 aligns well with the assumptions and ideals of this thesis.

The Meta Programming System (MPS) [**?**], from JetBrains, is a semi-proprietary product, which seems to be aimed more at programmers building languages for their own use or for the use of other programmers. MPS is based on an object-oriented approach to nodes and generation of (Java) code from the tree-structured source. Its editor and presentation language seem to be limited to mimicking the appearance and interaction style of textual source.

Both these systems seek to provide a complete environment for producing software (a Language Workbench [**?**]), displacing the conventional tools, and consequently they are multi-person-year projects. This thesis aims to show that useful functionality is possible with just a small, compact system.

MetaEdit [**?**] is a commercial system aimed at (non-textual) modeling languages. Tools ad-

dressing the difficulty of implementing good tool support for textual DLS are many, often targeting the free Eclipse IDE platform: [?][?][?].

Structure editors, or syntax-directed editors, had their greatest flowering in the 1980s, but usually had very different goals, including improved compiler performance (via incremental compilation) [?][?]. Lang provides an interesting retrospective [?].

Barista [?], built on Citrus [?], is a more recent structure editor for Java exploring new user interface paradigms for editing code.

The Fortress language [?] has similar aspirations with regard to the faithful reproduction of mathematical notation. Indeed its creator cites improved notation as one of three key ideas for the language [?]. It supports the use of any Unicode [?] symbol as an operator. However, Fortress's primary source format is ordinary text; programs are typically written in text and then translated to a richer presentation (via TEX) for publishing. This approach harks all the way back to Algol 60's multiple languages, and even the never-implemented m-expressions of Lisp. The Fortress spec also includes a mechanism for syntax extension, but it does not appear to have been implemented.

I present a representation for ASTs and fundamental tools for manipulating them in Chapter **??**. Chapter **??** shows how languages are defined and extended. Chapter **??** describes the prototyped system which implements these ideas, and Chapter **??** presents two case studies of the effectiveness of the system for defining new syntax and programming constructs.

Chapter 2

Abstract Syntax Trees

To build programs directly out of nodes, we need a common representation for nodes. It must be flexible enough to represent many kinds of programs and support the kind of editing operations we will want to provide. It should provide a natural way to represent the primitive values which appear in programs, and the common ways of aggregating values into larger structures. It should be able to represent arbitrary programs, and allow nodes to be freely composed without imposing any fixed constraints but always maintaining a clearly defined structure.

2.1 Nodes, Values, and References

A *program* is a tree made up of *nodes*. Every node has a *type*, a unique *label*, and a *value*. A node's type identifies it as one of a class of related nodes, all of which have some common meaning (for example, the type **plus** might represent addition expressions). A node's label is an opaque identifier which gives it a distinct identity, and allows references to nodes to survive transformations of the program. A node's value may be: *empty*, if the node type alone carries all the node's meaning; an atomic value, which is a *boolean*, *integer*, or *string* (a sequence of characters which are treated as an indivisible value);[1] a *sequence* of n child nodes (indexed by integers 0 to $n-1$) or an (unordered) *map* of distinct *attribute names* to child nodes. A *reference* is a special type of node which has the type **ref**, and has as its value the label of another node.

[1] An alternative would be to treat single characters as a primitive value, and build strings out of sequences of character-valued nodes. For the current purpose, treating strings as atomic is more efficient and simpler to deal with during editing.

I define these four kinds of nodes because they seem to be sufficient to represent the elements of programs for the languages I experimented with. These elements seem to fall into four categories, corresponding to the four kinds of nodes. Of the four, map nodes are the most commonly used, and their ability to contain an arbitrary set of attributes is key to the extensibility of the system. Sequences are represented as first-class nodes (rather than as, say, multi-valued attributes) primarily for the convenience of the editor, discussed later. Note: sequences can often be viewed as map nodes where the attributes are identified by integers rather than names and again the distinction is made mostly for convenience of implementation.

The program is *well-formed* if its nodes satisfy the following constraints:

(1) No two nodes have the same label.

(2) No node appears as a child of more than one parent, or under more than one index/name of a sequence/map-valued node (so the nodes form a tree).

(3) The value of each reference node is the label of some node in the program.

Nodes are immutable values, so a program is a *persistent data structure* [**?**]. It is relatively efficient to construct a modified program by building a new tree sharing much of the structure of an existing tree.

Note that any node can be considered as the root node of a sub-tree consisting of its descendant nodes. A sub-tree of a well-formed tree will be well-formed unless it contains a reference to a node which is not part of the same sub-tree. Such a reference is analogous to a free variable, and can play a similar role.

For the sake of modularity, types and attribute names are associated with namespaces. Simple names can be used without fear of unintentional collisions.

These minimal constraints allow trees to be transformed into one another using familiar functional programming techniques, and ensure that operations on trees have well-defined results. Therefore, Lorax requires that trees be well-formed at all times, and its components are designed to enforce these constraints.

2.2 Specifications

A *specification* constrains the structure of nodes in a program. A program is *valid* with respect to a certain specification if the arrangement of nodes, values, and references satisfies the specification's constraints.

In practice a program may contain violations of a specification, and the user will be interested in the nature of each violation in order to be able to fix them (by correcting either the program or the specification). Therefore a specification will typically be implemented in the form of a *checker*, a function over programs that produces a set of *errors* each consisting of a description of the problem and the location where it occurs.

Depending on the nature of the properties being checked, a specification may be defined only in terms of *local* properties of individual nodes and their direct children, or may refer to *non-local* relationships between nodes more distantly connected. In general, local properties are easier to define, easier to check, and easier to understand, so for the most part they are to be preferred. In section **??**, we describe a particularly direct and convenient way of specifying these basic structural properties which is sufficient to define the abstract syntax of a programming language.

By providing a modular way of describing and checking properties of programs, specifications give much-needed structure to the open model of ASTs described in the preceding sections. They do not, however, restrict the user's ability to modify and extend her program and/or language, even if that means the program is invalid at times.

2.3 Reduction

The next step in making a useful system is providing a way to produce (multiple) *target* programs from a given *source* program. The central idea is to have a source program, consisting of nodes in some "user" programming language, which includes a node type for every important programming construct. In an extensible system, the user is able to add new types of nodes or even entire languages to a running system, even though the underlying system only understands a

fixed set of node types. The way to bridge this gap is via *reduction*.

Reduction is a restricted form of *graph reduction*, inspired by the macro expansion process of Lisp.[2] In fact the term "expansion" might be more appropriate because a typical reduction replaces a more abstract node with a larger number of simpler nodes.

A *source* program is reduced to a *target* program by applying a *reduction function* to the root node of the source program. If any reduction is possible, the reduction function returns a new, replacement root node, which typically repackages the children of the original root node under some new kind of parent. As long as some reduction is performed, the reduction function is applied repeatedly to the previous result. Eventually, the root node is fully-reduced (the function fails to return a new reduced node). At that point, each child node is reduced in the same way, and a new root node is constructed with the reduced children.

Because each reduction step constructs an arbitrary replacement node, it's possible at least in theory to write any arbitrary transformation as a reduction. This means, for one thing, that reduction may not terminate. This generality, and the potential errors it allows, are typical of meta-programming systems; the full power of the language is available at compile-time, including the ability to introduce bad behavior. Although a system with less expressive power might be capable of meeting the need with less potential for bad behavior, in practice the kinds of problems that are encountered are most often easily fixed, provided that the system gives reasonably good error reports and no permanent damage is done when reductions do not perform as expected.

During reduction a series of intermediate programs are produced which are partially reduced, and in general do not conform to the source or target specification. It might be interesting to investigate ways of defining specifications and reductions such that it could be statically shown that a valid source program always reduces to a valid target program. I have not investigated how this might work, but it certainly would require imposing restrictions on the results that could be produced by reductions, suggesting at the very least a static type system for the language of

[2] When I use the term "Lisp", Im referring to any of the many variants of Lisp, including Maclisp, Common Lisp, Emacs Lisp, Scheme.

reductions.

As a practical matter, it's convenient to define reduction functions only for properly-formed inputs. This can be facilitated by declaring both the specification (e.g. expected attributes for each node type) and the reduction (given a node with those attributes) at the same time. If the specification defines only local properties, then the reduction should assume only the presence of *some* node at each required location, but make no demands on the form of these child nodes.

2.4 Characteristics

This way of constructing program source has some implications for the way languages can be defined and the way programs can be worked with.

Node types, attributes, and specifications naturally fit with the concepts of *tree grammars*, which allows specifications to be defined in ways that are familiar to language designers (i.e. with a grammar). Because grammars so-defined will be used only for checking tree structure and not for parsing, they can be constructed in the natural way, without any need for tricks to work around parsing algorithm shortcomings (e.g. left-factoring).

Programs are *self-describing*. Each node carries an explicit declaration of its meaning, and each primitive value is manifestly of a certain type. This is in contrast to a textual language, where the same sequence of characters might represent a name, data, or a keyword, depending on the context in which they appear. This is a major advantage especially for tools that manipulate programs, because no parser is necessary to extract the structure of the program.

Labels provide robust *source locations*. The label provides each node with an identity that survives when the other parts of the program are changed, or when the program is serialized to non-volatile storage, etc. Thus labels provide a way for tools (e.g. compilers and debuggers) to refer to source locations. For example, a typical refactoring operation such as changing the name of a variable or moving some code from one place to another looks like several unrelated changes to a *diff* tool that has two versions of source text to look at, but a similar tool operating on labeled nodes could compare each *node*, even if the location, the content, and even the type of the node

have changed.

Reference nodes provide a way of referring to entities in a program which can never be ambiguous, and is independent of such language-specific notions as names and scope. Using labels, as opposed to, say, pointers to the referenced node, keeps the tree simple, makes the relationship explicit, and allows references to be inspected without recourse to low-level techniques. Reference nodes require special support from editors, which may also take advantage of the explicit reference structure to provide enhanced presentation of references (which in a textual setting would require knowledge of not only the lexical but also the semantic structure of the language).

Nodes may be serialized for storage, distributed processing, etc. Assuming a choice of character set and encoding, all the components of a node can be easily converted to a stream of characters. Node types, labels, and attribute names can be simple strings; boolean, integer, and string values are easily handled; map and sequence values pose no great challenge. Because the representation permits only trees, each node can be serialized when it first occurs; there's no need for an encoding of back-references. Moreover, the choice of serialized format is not so important because the structure of programs is defined at the level of nodes. Any serialized form that preserves the meaning in the terms defined above is equally good.

This representation of programs as nodes has some things in common with a handful of other tree-structured representations source code and/or intermediate representations within tools such as compilers.

2.4.1 Compared to XML

XML and other related *markup languages*, are designed to augment textual data with explicit structure for a variety of purposes. The structure of an XML document has much in common with the structure of nodes as defined here, and the *XML Infoset*[**?**] model for documents in particular has a similar flavor. However, there are some important differences:

(1) The child elements of each node in an XML document are always in an ordered sequence, while named attributes can contain only simple character data. In Lorax, a node's children

may be ordered or named, whichever makes sense, and there is no separate notion of attribute values vs. child nodes.

(2) XML is explicitly a format for character streams, onto which an abstract model can be imposed *post-hoc*. This leads to many awkward and ultimately uninteresting problems, such as when to ignore white-space and when it should be included in the data model. In Lorax, the source AST is defined in terms of the relevant types, and only nodes that actually contain character data are represented as characters.

(3) Because XML is meant to be human-readable in a weak sense, text-editors are still the dominant mode of interaction with it. Although there are WYSIWYG tools for editing and viewing certain kinds of XML documents (e.g. SVG[**?**], docx[**?**]), there are few general-purpose tools for working with XML per se, aside from text editors.

XML's undesirability as a concrete syntax for programs has been well-established, and stands as an example of the usability challenges posed by representing source code as structured data [**?**][**?**]. I believe this this failure largely results from the use of the textual representation as the editing interface. Although the XML syntax is particularly bad in this regard, any general-purpose, textual markup language will suffer in terms of readability for the flexibility it provides.

2.4.2 Compared to S-expressions

Most Lisp programs are written in *s-expressions*, a simple data structure offering only lists, symbols (i.e. names), and a handful of types of primitive values. This simplicity gives considerable generality and flexibility, because it allows any number of new constructs to be introduced simply by defining new forms (macros). However, the use of s-expressions directly in lieu of a concrete syntax has been a highly divisive choice, essentially separating the population of programmers into two "camps". Only those who are not put off by sequences of nested parentheses can appreciate the expressive power that Lisp offers.

It is a major goal of this thesis to establish that the expressive power of the Lisp model can

be realized in the context of a language that can be read by "the rest of us." Lorax's representation for nodes has the same generality and free extensibility as s-expressions, and its simple and unambiguous structure makes it ideal for defining and transforming programs.

Chapter 3

Languages

The preceding chapter described a representation for programs which is suitable for any language. To define a particular language in this framework involves defining three aspects of the language.

The *abstract syntax* of the language defines what nodes may appear in a program, and in what relationship to each other. In Lorax, abstract syntax is defined by a program in the **grammar** language. The grammar program is first and foremost a specification which defines what programs are (syntactically) valid.

Every language must have a *semantics* that gives its programs meaning. A particularly economical way to give semantics is to define a minimal *kernel* language and then define the remainder of the language in terms of reduction from an enriched syntax to the kernel syntax. Concretely, when a grammar defines a node type, it may also define the semantics of the construct by means of a reduction to a more primitive construct. This approach is commonly used to *define* languages, but not necessarily to implement them (see, e.g. Mozart [**?**]). In Lorax, as in Lisp, this economical approach is opened up to the programmer—the same extension mechanism used to define the core language is available in all programs. The example of Lisp (particularly, Scheme [**?**]) shows this to be a workable strategy, so the experimental question is how well it applies in this new setting.

Thus a Lorax grammar defines a language's syntax and gives a semantics for programs in that language in terms of the kernel language programs they reduce to. A second kind of semantics that

may be desired is *static semantics*, that is, specification of additional constraints on valid programs including anything from proper scoping of references to a type system. These constraints could also be specified in a grammar language, but Lorax does not currently support such declarations.

With abstract syntax and semantics taken care of, both the Lisper and the Programming Language Theorist have all they need and will go happily on their ways. However, we would like to go further than that, so one more element is needed. *Concrete syntax* is what the user reads and writes. The goal is to take advantage of the new approach to do things with concrete syntax that are impractical or impossible to do with textual source code. In Lorax, the concrete syntax of each type of node is given as a reduction to a *presentation language*.

3.1 Grammars

A grammar defines all three aspects of a language. The abstract syntax is defined *prescriptively*, by identifying the subset of all possible programs which are valid. Both the semantics and concrete syntax are defined *operationally*, as declarations of what reduced program and what visual representation will be derived from each source node.

3.1.1 Abstract Syntax

A program is *valid* with respect to a *grammar* if all of its nodes' types, values, and references comply with the constraints imposed by the grammar. For each node type, a grammar specifies:

(1) Zero or more *abstract types*, which are types for which the type being introduced is a *sub-type*.

(2) For a sequence node, the expected number and (abstract) type of child nodes.

(3) For a map node, the expected attributes and the (abstract) types of nodes that may inhabit each of them.

(4) When a child/attribute is a reference, the expected (abstract) type of the referenced node.

$$expr \leftarrow pow\ \{e{:}expr,\ n{:}\mathbf{int}\}$$

Figure 3.1: A simple node declaration.

Abstract types lend some modularity to the grammar. A new subtype of any abstract type can be introduced later without changing any existing definition, and nodes with the new type can appear as children of nodes which were declared using the abstract type. Figure **??** shows the declaration of a hypothetical node type **pow**, for expressions raising some runtime value to a fixed integer power. **pow** is an instance of the abstract type **expr**. To be valid, a **pow** node must have an attribute **e**, containing a node whose type is an instance of **expr**, and another attribute **n**, an integer. So far, this declaration mimics the way you might define a similar node as an algebraic data type in a functional language such as Haskell or ML.

Note that all the grammar constraints except the constraint on the types of referenced nodes are local in the sense described earlier—checking them requires inspecting only a single node and its immediate children. Checking the types of referenced nodes requires being able to inquire about the type of a node which may be anywhere in the program. However, this information is easily extracted up front to a map of types for all nodes in the tree, so this one non-local check seems to be worth including in the standard checker.

A grammar so-defined is not capable of specifying every interesting property of a language. That is, not every syntactically valid program is correct. Depending on the nature of the language being defined, additional, more ambitious specifications can define additional properties (e.g. static semantics). The abstract syntax is meant to be an easily-defined first step, and to provide the information needed by the editor to provide support for editing programs which use arbitrary new node types.

3.1.2 Reductions

In addition to defining what programs are syntactically valid, a grammar provides a definition of concrete syntax via a *display* reduction and of semantics via an *expand* reduction. Either or both

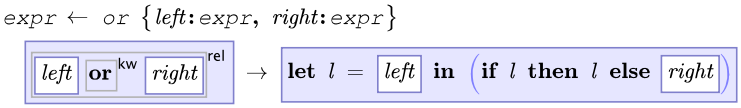


Figure 3.2: The **or** node of the core language, including reductions.

reductions may be present in a declaration. These reductions are code fragments which are used to construct reduction functions (see section **??**) which take a program in the source language defined by the grammar and reduce it to a derived program. Using a meta-programming approach, the reductions are economical and simple to define, but the full host language is available for use in reductions when needed.

A display reduction is always present and reduces a source node to a presentation language node (often the root of a sub-tree containing several presentation language nodes). On the lower left in Figure **??** is the display reduction for **or** nodes, which arranges the left and right children of the node into horizontal sequence, separated by the keyword "or."

An expand reduction is used to reduce a node in preparation for evaluation or execution, and therefore defines the semantics of the node. The result of this reduction is a node in the *target language*, or else a node that will itself be reduced, eventually to a target language node. The declaration of a node type does not provide a expand reduction if the node is part of a target (kernel) language. On the lower right in Figure **??** is the expand reduction, which defines the behavior of the **or** construct in terms of kernel language constructs: it evaluates the left argument and yields the value if it is not **false**, otherwise it evaluates the right argument and yields that value. This kind of short-circuiting evaluation returning the first "true" value is a common way of implementing the logical "or" operator in scripting languages [**?**][**?**][**?**].

It is easy to define a reduction that fails to terminate (for instance by producing a node of the same type), or which "blows up" (by producing a larger, but no more reduced, node). The usual approaches to proving termination of recursive algorithms apply. In any case, once language definition is part of the development process, it should no longer be so surprising that problems can arise at that level. Because you are your own language vendor, you can simply fix it; it's not

like finding a bug in your C++ compiler, which you may have no capability to fix.

Thus the abstract syntax, semantics, and concrete syntax of every derived language construct are defined together in a simple, declarative style. Each construct is self-documenting; any use of a new construct can always be expanded (manually or automatically) to the equivalent reduced program.

3.2 Kernel Languages

A *kernel language* is one whose nodes have some fixed, pre-defined semantics (typically, they can be interpreted, yielding some specified result with some specified side effects). A complete language is built by *extending* a kernel language with additional nodes whose semantics is defined in terms of reduction to (ultimately) the kernel language.

Any language can act as a kernel language in Lorax. If the kernel language is of a more conventional type, say Java, all the constructs of that language have to be defined as primitives. Additional constructs could then be defined via reduction to ordinary Java syntax. After reduction, the target program would be processed by a Java compiler to produce an executable. Thus a language like Java is a suitable kernel language, but hardly a convenient one.

I designed the Lorax's *host kernel language* to be much simpler to implement and use. The host kernel language is designed to be:

(1) Minimal, but sufficient to define a general-purpose language via extensions.

(2) A meta-language, able to consume and produce nodes.

(3) A "functional" language, operating naturally on immutable nodes and trees.

(4) Easily implemented and easily integrated with existing software.

These characteristics make the host kernel language—and the core language built on top of it—ideal for implementing Lorax itself. For example the reductions themselves are written in it. I describe the host language in detail in section **??**.

3.3 Presentation Languages

The nodes of a *presentation* language cast program elements in visual terms. A presentation language should be independent of the particular syntax of any one language, but might be suited to a particular family of related languages. Crucially, the presentation language must be flexible enough to represent any conceivable construct that might be added to the source language. This is achieved by providing composable elements in the presentation language that can be combined in new ways to create new concrete syntax. In Lorax, a single presentation language is currently implemented.

3.3.1 The *expr* Presentation Language

Lorax's **expr** presentation language is well-suited to representing the expressions that make up the declarative portion of a typical modern programming language in the usual way, except that it provides more typographically-rich elements. It can also reproduce much of the familiar notation of algebraic expressions.

An **expr** program is represented internally as a tree made up of *box* nodes. Each box node occupies a rectangular area of the rendering surface and always encloses the areas occupied by any child boxes, although this structure is not visible to the user, who sees a familiar layout of keywords, operators, and other textual elements. Several kinds of boxes are available in **expr**.

An *atom* is a box that renders a sequence of characters and/or symbols using the normal spacing for text. Several types of atoms are available, each conveying what kind of entity the characters are meant to represent. When atoms are reduced to lower-level nodes, a distinctive typographical treatment is applied to each, as shown in Table **??**. The set of atom types is meant to be easily expanded to serve the needs of any conceivable source language. The total number of types in any one language is probably limited to a dozen or so, and many are common across languages. Therefore I conjecture that the universe of useful atom types is not much larger than what **expr** currently provides. When a new type is needed, the effort to add it is typically small

Type	Treatment	Examples	Typical use
keyword	boldface	**true** **if**	fixed language syntax
var	italic	x g fib	names (user-provided or generated)
num	*none*	1 2.0 3,000	numerical literals
string	sans serif	abc Hello,␣world	character literals (note special treatment of the space character)
name	boldface, italic	***a*** ***foo***	name literals
mono	monospaced	`nil` `cons`	external references
prod	monospaced, italic	*`expr`* *`left`*	node types in grammars
symbol	*none*	$\rightarrow$ $\in$ $\sum$	mathematical symbols

Note: the actual character content of each type of atom is arbitrary, except **symbol**, which handles only a pre-defined set of available symbols.

Table 3.1: Types of atoms in the **expr** language.

(most simply specify a font and/or face),

Composite boxes contain child boxes which they arrange in a certain way. A *sequence* is a horizontal arrangement of nodes separated by a certain amount of space. There are a handful of sequence types, each implying a certain amount of inter-node space. The amount of space is a visual indication of how tightly the sub-expression represented by the sequence binds, as I'll discuss in the next subsection. A **scripted** node contains a *nucleus* and *sub-* and/or *super*-script nodes. *Special signs* are similar to composites but also draw a glyph surrounding the child node(s). Examples are **radical** and **fraction**, used in particular mathematical contexts.

A **group** node draws a pair of grouping symbols around a content node. Available symbols include parentheses, various kinds of brackets, $\lfloor floor \rfloor$, $\lceil ceiling \rceil$, and $|magnitude|$. All these symbols expand vertically to visually surround their contents; this variation in size is aesthetically pleasing, but can also provide a visual cue which helps the reader to match up the paired symbols.

An **embed** node encloses its contents in a visual indication of being a separate unit, contained within the surrounding context. A **disembed** node has the opposite effect, showing its contents as being logically part of the outer level of code. These are used in the rendering of quotations, and have been seen already in the example reductions in Figure **??**.

3.3.2 Expressive Power

The diversity of atom types provides a measure of visual sophistication for programs, with a very small effort on the part of the language designer. Simply by identifying a visual style for each piece of syntax, some information about the meaning of each node is conveyed to the programmer. The particular treatment is meant to match the readability and high aesthetic standards of the pseudocode that might be found in a journal article or a good computer science textbook.[1] The rendering of these fonts on screen at relatively low resolution of 100 dots per inch or less is not really optimal, but given the trend toward more dense displays (as of late 2010, laptop displays approaching 150 dpi are common and displays on handheld devices can be much higher than that), it is likely that acceptable rendering of multiple fonts even at small sizes will soon be quite achievable. In any case, the actual choice of how to present each type of atom is unrestricted; a more conventional approach would be to use just a few fonts and instead use color to distinguish different elements in the conventional "syntax-highlighting" manner [**?**]. These issues have been explored heavily in the field of integrated development environments (IDEs), but few published studies seem to exist. One reference is Harward, et al [**?**].

Because the **expr** language preserves the hierarchical structure of expressions and specifies the visual layout of each sub-expression, it's possible for the system to identify cases where the nesting of expressions could lead to confusion, and to insert parentheses automatically in these cases. This is done *after* the reduction of the program to the **expr** language, so it's applied consistently to any language construct, without special effort at the point of definition of a node.

[1] The reader can judge my success by inspecting any of the figures in this paper, most of which were captured directly from Lorax's editor as vector graphics.

The **expr** language provides enough levels of spacing (four) to handle a moderately complex expression without parentheses. For example, in the expression

$$\textbf{if } x < y + 4! \textbf{ then } ...$$

the factorial operator binds most tightly (!, no space), then addition is performed (+, with a thin space), then comparison (<, medium space), and finally the conditional (**if** ... **then** ..., thick space). Examples of how this works will be presented later.

A presentation language is concrete in that it represents the program as it is presented to the user, however it is somewhat abstracted from the details of rendering characters and pixels. The reduction from source to presentation language is therefore quite direct and simple. Once the program is reduced to the presentation language, lower-level processing takes care of the details of rendering. This lower-level processing is common to all languages using the same presentation language, and does not need to be extended in the typical course of using (and extending) a source language.

expr supports nested expressions well, and in combination with the lower-level presentation language (described in section **??**) it can produce good results for some moderately complex programs if used with care. However to handle larger constructs such as classes or modules would raise layout problems that **expr** is not really intended to address. For example, how to handle indentation and when and how to break lines. Additional nodes could be added to **expr** to address these issues.

Lorax's presentation language, plus this kind of extension, should be able to implement most general purpose programming languages, and allow their syntaxes to be extended in new ways. With some additional features, it could also support new kinds of visualization and interaction for source code. For example, embedded images as in Sikuli[**?**], or program elements that respond to clicks as in a typical structure editor.

Going further, to serve the needs of different kinds of languages, entirely different presentation languages could be defined, such as a "flowing text" language for documents, a "grid" language

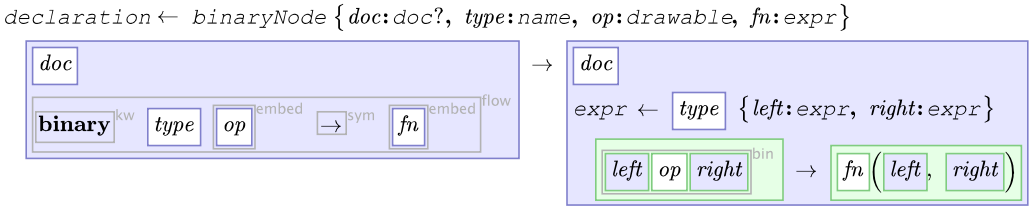


Figure 3.3: Definition of **binaryNode** as an extension of the grammar syntax.

for spreadsheet-like programs, a "flowchart" language for state machines, data flow programs, and regular expressions, and so forth. These styles might call for an entirely new low-level presentation language and renderer, but the underlying framework of Lorax would be able to handle the jobs of defining languages, and validating and transforming the programs.

3.4 Extending the Grammar Language

The grammar language (section **??**), host language (**??**), and presentation language (**??**) work together to support the definition of new language constructs. A grammar is a Lorax program, so the same tools for syntax extension are available for use in grammars. The **grammar** language should be viewed as a starting point. It is meant to be general enough to define typical languages, to handle most common needs for defining syntax, and to serve as a kernel language for grammars.

As an example, consider the variety of 2-argument, infix operators that we may wish to define for the core language. If defined in terms of the kernel grammar language, each declaration will echo the definition of **or** shown in Figure **??**, varying only in the symbol used to represent the operator, and the name of a function to invoke. This kind of duplication is a clear opportunity for language extension, which you can do in Lorax by adding to the grammar for the grammar language itself.

Figure **??** shows the declaration of a new grammar node, **binaryNode**, with just four attributes for the parts of the declaration which are unique in each case: an optional documentation string, the name for the node being declared, the symbol which will represent the new operator for display purposes, and a function in the host language which implements the operation.

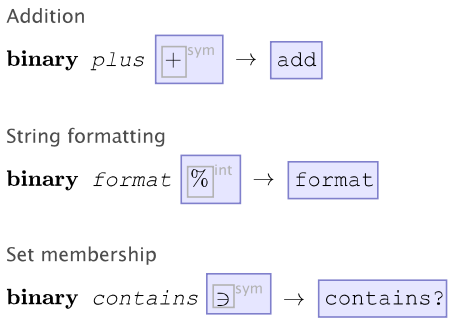


Figure 3.4: Some infix operators defined using **binaryNode**.

The display reduction (the blue, quoted box on the left) for these declarations mimics the regular node declaration syntax but omits the declaration of attributes and has just four editable elements. The expansion on the right is a quoted declaration which will be evaluated to produce an ordinary node declaration. Compared to the five nodes of the simplified declaration, the expanded declaration contains about 16 nodes. Like much syntax extension, the new node embodies the well-known goal of eliminating code duplication [**?**]. As seen in Figure **??**, node types declarations using this new syntax are significantly simplified.

What's happening here is the concrete syntax and semantics of the new construct are given in the form of two small meta-programs. However, each reduction is presented in the rich syntax of its target language (the **expr** presentation language itself on the left, and the **grammar** language on the right), and the visual representation of quoted and un-quoted nodes gives the reductions the look of a simple template with "holes" for elements to be inserted. Thus the grammar language combines the power of meta-programming with a simple, understandable visual representation.

Chapter 4

Implementation

The Lorax prototype implements nodes, reductions, and grammars as described in the previous chapters, and a compiler for the kernel language. I built a core language via syntax extension of the kernel language. The Lorax editor is capable of editing and executing both grammars and core language programs.

The prototype demonstrates the feasibility of implementation all of the previously-discussed concepts, and provided a test bed for experiments with defining languages and syntax, described in the next chapter.

4.1 The Host Platform

The prototype system is implemented in Clojure [**?**], a language in the Lisp family which runs on the Java Virtual Machine [**?**]. Clojure also fills the role of the host kernel language. Some characteristics of Clojure that make it a good choice include:

(1) Similar notion of a kernel language. The prototype simply adopts (a subset of) the special forms of Clojure as its kernel language.

(2) Functional orientation. Clojure is a non-pure functional language; it allows side-effects but (in contrast to most Lisps) all of its bindings and data structures are immutable.

(3) Compilation service available at runtime. The Clojure compiler is part of the runtime stack, so once a program is reduced to the kernel language, it can be evaluated (compiled)

and executed immediately.

(4) Access to the Java platform. The prototype takes advantage of Java's GUI toolkit for rendering and editing.

Nodes are represented as a Lisp-style abstract data type comprising a series of functions `make-node`, `node?`, `node-type`, etc., which allow Clojure programs to construct and manipulate nodes independent of the concrete representation. The prototype uses only these functions to work with nodes.

The implementation of the node ADT uses Clojure's `deftype` construct to represent nodes as instances of a `nodetype` class at the JVM level. This reduces the overhead for simple nodes to the minimum. Map- and sequence-valued nodes use Clojure's built-in persistent hash map and persistent vector data structures (based on Bagwell's array mapped tries [**?**]) to store their children.

The `make-node` function provides a convenient serialized form for nodes: s-expressions. A `print-node` function turns a node object back into text which can be read and evaluated to re-constitute the node. This low-level representation is never seen by the user of the system.

4.2 The Host Language

The host language is composed of a kernel based on the primitive constructs and values of Clojure together with a core language built via syntax extension on top of the kernel language. Lorax core language programs are reduced to the kernel language (via reductions defined in a **grammar** program), to Clojure's s-expressions (via the Lorax meta-compiler), to Java bytecode (via the Clojure compiler), and finally to a native executable (via the JVM's compiler/optimizer). Therefore a program written using newly-introduced syntax runs without any fixed overhead. This allows the editor to perform reductions during editing with good responsiveness, and allows for non-trivial programs to be executed (see the next chapter). At the same time, taking advantage of the Clojure compiler and platform allowed me to build a working system in a short time and with only a modest amount of code. This efficiency allowed me to bring two previous approaches to a

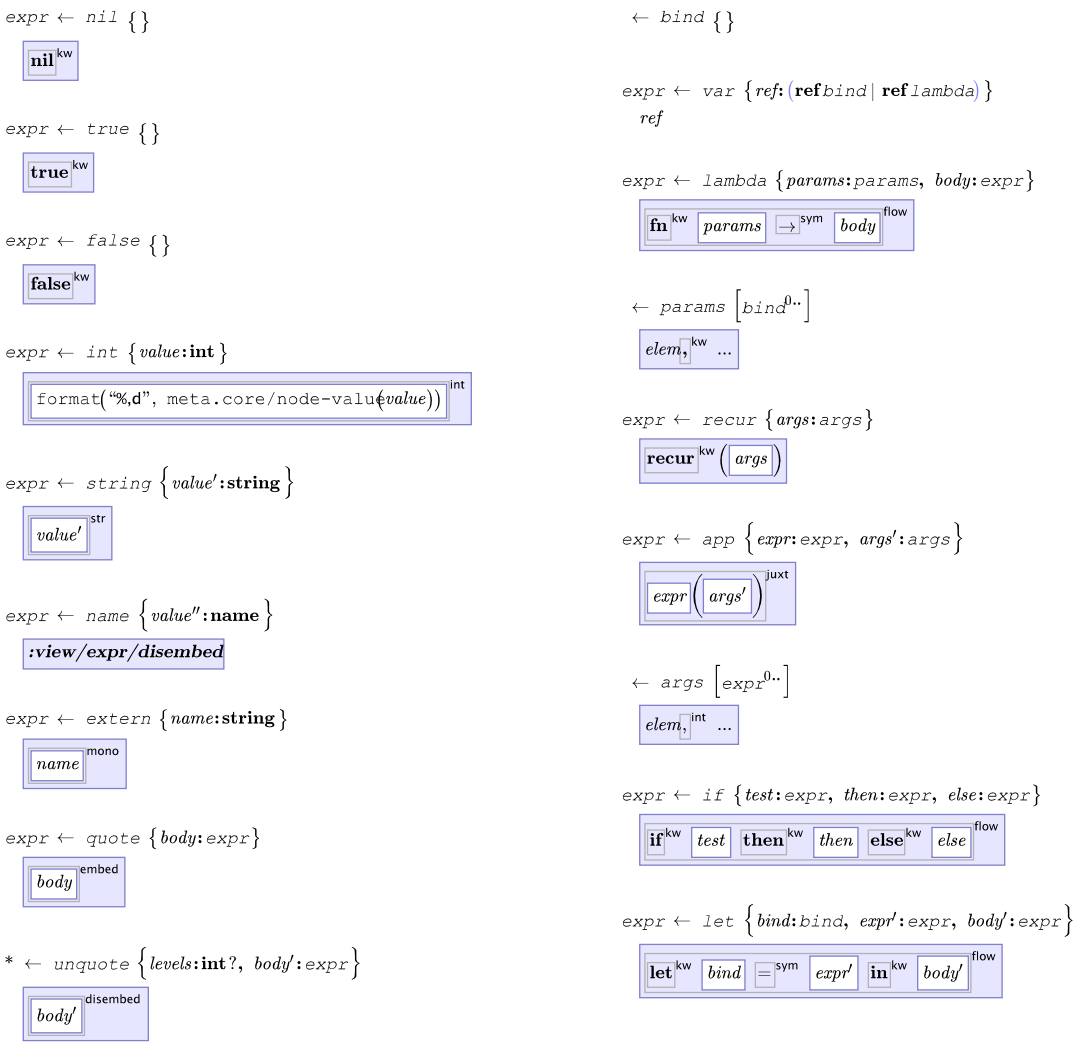


Figure 4.1: Grammar for the host kernel language.

mostly-working stage before I arrived at the node representation, reduction scheme, and grammar language described in this thesis.

4.2.1 The Kernel Language

The kernel language, shown in Figure **??**, consists of six kinds of primitive values, six node types corresponding to Clojure's special forms, two nodes for quotation, and a node for making external references to the platform. Virtually every node is an instance of the abstract **expr** type, yielding some kind of value as a function of its arguments; there are no statements and no mutating assignments.

The kernel language is extremely simple compared to a typical general-purpose language. Like Clojure, the kernel language is not statically typed, which helps keep Lorax's implementation simple—all checking of operands is done at runtime, by the Clojure/Java platform. All but a few kernel language node types are instances of **expr**, representing expressions yielding a single value. This simplicity makes the language easy to implement, and makes it an easy target for reductions.

The primitive values of the kernel language are the singular values **nil**, **true**, and **false**, plus integers, strings, and names.

A **lambda** node introduces a function taking a fixed number of arguments. A **params** node always represents the parameters of a function, and can appear only as the **params** attribute of a **lambda** node. Each parameter is a **bind** node, discussed later. A lambda node is "bound" within its body, supporting simple recursive calls. **recur** is a non-stack-consuming recursive call, and must appear in tail position.

n-ary functions are included in the kernel language as they are in Clojure to allow a simple mapping to Java methods for high performance without sophisticated compilation techniques. The kernel language does not support Clojure's variable-arity functions.

app is (call-by-value) function application. The *expr* is evaluated first (to a function), then the expressions in **args** are evaluated from left to right.

Simple conditionals are provided by **if**. All values except **false** and **nil** are treated as true when they appear as the *test* expression.

let evaluates an expression, binds a name to the resulting value and then evaluates its body with that binding in scope. Lorax provides only a single-binding, non-recursive primitive *let* form.

A **var** node is a reference to the value bound by the parameter of a lambda, by the lambda itself, or by a let expression. Variables are lexically-scoped, and lambdas capture all bindings in scope at their point of declaration. All these forms are strict in all their arguments, and have no side-effects.

An **extern** node refers to a variable at the Clojure level. This allows any function or value defined by the Clojure platform or Lorax runtime to be accessed. Use of these external facilities is

minimized in order to make the core language as self-contained as possible.

quote and **unquote** are given special treatment by the meta-compiler.

4.2.2 Meta-compilation

Given a program in the kernel language, a function `meta-compile` translates the nodes to Clojure s-expressions in a mostly straight-forward way, due to the close correspondence between kernel nodes and Clojure's special forms. Only a few types of nodes need special consideration.

A **bind** node is reduced to a symbol, and a **var** node to the symbol of the corresponding binding. The problem of producing suitable names and avoiding conflicts is easily solved due to the uniqueness of labels in the Lorax program. The meta-compiler simply generates a new name for each label and uses that name for both **bind** and **var** nodes.

When the meta-compiler encounters a **quote** node, it enters a separate mode where instead of translating kernel language nodes to s-expressions, it instead translates nodes (potentially in any language) to s-expressions that construct a new copy of the quoted nodes. When an **unquote** node is encountered, the meta-compiler reduces the contents in the normal way, and then inserts the resulting value into the expression building the quoted node. Quotations may be nested more than one level deep, so the reduction keeps track of the current level and only resumes ordinary translation when the level reaches zero.

When quoted nodes are compiled, they are relabeled (that is, each node is assigned a fresh, globally-unique label) and each non-free reference is updated to point to the new node. This is important because it means that no code outside of the newly-generated fragment can possibly contain a reference to anything defined in it. For example, reductions for syntax extensions commonly bind values to names (in order to control evaluation order, for example), and each name is defined by some **bind** node. So in a reduced program, each instance of the reduced fragment will contain a binding which originated in the same node, but each binding is completely distinct and independent in the reduced program due to the relabeling that is performed when the reduction is executed. Because all names are unique at all times, the problem of unintended variable capture

Examples using the core syntax for sequences:

```
$  1 : (2 : nil)
→  1, 2

$  1, 2, 3
→  1, 2, 3

$  let lst = "a", "b", "c" in first[lst], first[rest[lst]], lst[2]
→  "a", "b", "c"
```

Note: f*(x) produces a lazy sequence, and x[i] uses no stack:

```
$  let inc = (fn x → x + 1) in (inc*(1))[999,999]
→  1,000,000

$  1..10
→  1, 2, 3, 4, 5, 6, 7, 8, 9, 10

$  (−100..1,000,000,000)[150]
→  50

$  y^2 | y ← 1..10
→  1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

Figure 4.2: Example core language expressions and their values.

does not arise, and some of the complexities of providing "hygienic macros"[**?**] may be avoided.

Once a kernel language program has been meta-compiled, the standard Clojure function `eval` is invoked, which compiles the expression to Java bytecode, loads it into the running JVM, invokes it, and returns the result as a Clojure/Java value.

4.2.3 Core Language

The host core language provides common facilities built via syntax extension on top of the kernel language. A subset of Clojure's Core API [**?**] is implemented, plus primitives for constructing and manipulating AST nodes. Some examples using the syntax which exposes Clojure's cons-list primitive are shown in Figure **??**. Some of the definitions used in these examples are presented in Section **??**.

4.2.4 Compiling Reductions

To employ a *display* or *expand* reduction defined in a grammar, it needs to be transformed into a reduction function which can be applied to a source program node, yielding a reduced node (see Section **??**). Each reduction in the grammar is an expression which may contain variables referring to the attributes of the node. Therefore a reduction function can be constructed by simply wrapping the reduction expression in a lambda abstraction and a series of **let**s binding each of the attributes. For example, the reduction for **string** nodes looks like this:

$$\textbf{fn}\ \ n\ \rightarrow\ \textbf{let}\ \ \mathit{value}\ =\ \textbf{attr}\big[n,\ \ \textbf{\textit{value}}\big]\ \textbf{in}\ \ \boxed{\mathit{value}}^{\mathrm{str}}$$

where **attr**$[node, name]$ extracts the named attribute from a node. Once the **value** attribute is bound, the result node is constructed by evaluating the quoted **string** node (which appeared in the actual declaration in the grammar shown in Figure **??**).

A similar function is constructed and compiled for each node type, and then an overall reduction function is built which dispatches to the right reduction based on the type of each node. This overall function is built incrementally as the declarations of a grammar are processed, so that later reductions can make use of syntax defined in earlier declarations.

4.3 Rendering the Expression Language

After a program is reduced to the **expr** presentation language (via the **display** reductions specified in the particular grammar being used), it can be presented to the user using the common facilities of Lorax. To do so, the program is further reduced to a low-level presentation language.

4.3.1 The *view* Presentation Language

The **view** language is a lower-level presentation language. Some of the algorithms of TEX [**?**] are used to lay out nodes and construct glyphs, as in e.g. [**?**]. In contrast to the **expr** language, all the elements of the **view** language specify fonts, colors, and sizes in concrete terms, so they can be

rendered directly. The main tasks of the reduction from **expr** to **view** are to identify the correct font size for each atom based on the nesting of expressions, and to transform **embed**/**disembed** nodes into border/background colors. Both of these computations involve propagating some information about the structure of the tree down as the nodes are recursively processed.

The mode, a concept from TEX, identifies the depth of nesting of expressions, and controls the selection of font sizes for atoms. There are four modes, *Display* (D), *Text* (T), *Script* (S), and *Scriptscript* (SS). D and T call for normal-sized text, S is about 30% smaller, and SS is about 50% smaller. The handling of modes follows TEX's algorithm in most cases, but deviates from it when necessary for consistency. For example, typographical convention dictates that addition expressions in the smaller modes should be set without spaces, but that would clash with Lorax's handling of parentheses, so Lorax sets these expressions with proportionately smaller spaces.

Each of the atom types of the **expr** language is reduced to a **chars** node. For example, when a node n having the type **keyword** and *str* attribute s is reduced:

$$n: \ \textbf{keyword}\ \{str \mapsto s\} \quad \longrightarrow \quad \textbf{chars}\ \{str \mapsto s,\ font \mapsto \textbf{keywordfont}_{\textbf{mode}[n]}\}$$

where $\textbf{keywordfont}_D$ and $\textbf{keywordfont}_T$ are both names for the normal-size font for keywords, and $\textbf{keywordfont}_S$ and $\textbf{keywordfont}_{SS}$ name the two smaller versions. $\textbf{mode}[n]$ is an inherited attribute [**?**]; except where specified, the mode of each node is the mode of its parent. Except for a handful of types, most atoms are reduced in exactly the same way, with a different font used for each type.

Atoms representing string literals get special treatment. They're surrounded by quotes, and any white space characters in the string value are replaced by Unicode OPEN BOX characters (␣).

$$n: \ \textbf{string}\ \{str \mapsto s\} \quad \longrightarrow \quad \textbf{chars}\ \{str \mapsto \text{“} + \textbf{escapews}[s] + \text{”},\ font \mapsto \textbf{stringfont}_{\textbf{mode}[n]}\}$$

Each of **expr**'s four sequence types are reduced to a single type, and spaces are inserted

between the child nodes:

$$\textbf{juxt}\ [c_0, c_1, \ldots, c_m] \quad \longrightarrow \quad \textbf{sequence}\ [c_0, c_1, \ldots, c_m]$$

$$\textbf{binary}\ [c_0, c_1, \ldots, c_m] \quad \longrightarrow \quad \textbf{sequence}\ [c_0, \textbf{thinspace}, c_1, \textbf{thinspace}, \ldots, \textbf{thinspace}, c_m]$$

$$\textbf{relation}\ [c_0, c_1, \ldots, c_m] \quad \longrightarrow \quad \textbf{sequence}\ [c_0, \textbf{medspace}, c_1, \textbf{medspace}, \ldots, \textbf{medspace}, c_m]$$

$$\textbf{flow}\ [c_0, c_1, \ldots, c_m] \quad \longrightarrow \quad \textbf{sequence}\ [c_0, \textbf{thickspace}, c_1, \textbf{thickspace}, \ldots, \textbf{thickspace}, c_m]$$

A **fraction** node is reduced to **over**, which has a configurable line weight, and the mode of each child is reduced:

$$n : \textbf{fraction}\ \{\mathit{numer} \mapsto c_0,\ \mathit{denom} \mapsto c_1\} \quad \longrightarrow \quad \textbf{over}\ \{\mathit{top} \mapsto c_0,\ \mathit{bottom} \mapsto c_1,\ \mathit{weight} \mapsto 1\}$$

$$\textbf{mode}[c_0] \leftarrow \mathit{fractionmode}(\textbf{mode}[n])$$

$$\textbf{mode}[c_1] \leftarrow \mathit{fractionmode}(\textbf{mode}[n])$$

$$\mathit{fractionmode}(D) = T$$

$$\mathit{fractionmode}(T) = S$$

$$\mathit{fractionmode}(S) = SS$$

$$\mathit{fractionmode}(SS) = SS$$

A **scripted** node is unchanged except that the mode of the raised and lowered child nodes is reduced:

$$n : \textbf{scripted}\ \{\mathit{nucleus} \mapsto c_0,\ \mathit{super} \mapsto c_1,\ \mathit{sub} \mapsto c_2\}$$

$$\textbf{mode}[c_1] \leftarrow \mathit{scriptmode}(\textbf{mode}[n])$$

$$\textbf{mode}[c_2] \leftarrow \mathit{scriptmode}(\textbf{mode}[n])$$

$$scriptmode(D) = S$$

$$scriptmode(T) = S$$

$$scriptmode(S) = SS$$

$$scriptmode(SS) = SS$$

The **embed** and **disembed** nodes which represent quotations and other kinds of embedded code are reduced to **border** nodes, which draw a colored border around their contents and a background behind them. The color of each border is determined by the meta-level of the node. The level of the root node is always 0, and it is incremented when an **embed** node is encountered:

$$n: \ \textbf{embed}\ \{content \mapsto c\}$$

$$\rightarrow \textbf{border}\ \{weight \mapsto 1,\ margin \mapsto 3,\ border \mapsto b,\ fill \mapsto f,\ content \mapsto c\}$$

$$b = \textbf{bordercolor}_{\textbf{level}[c]}$$

$$f = \textbf{fillcolor}_{\textbf{level}[c]}$$

$$\textbf{level}[c] \leftarrow \textbf{level}[n] + 1$$

Two series of colors $\textbf{bordercolor}_i$ and $\textbf{fillcolor}_i$ are pre-defined. $\textbf{bordercolor}_1$ is dark blue, $\textbf{bordercolor}_2$ is dark green, etc. Six colors are defined to support several levels of embedding. $\textbf{fillcolor}_0$ is white (the background color of ordinary nodes). $\textbf{fillcolor}_i$ is a lighter shade of $\textbf{bordercolor}_i$.

For **disembed**, a similar border, but the border color corresponds to the *parent* node's meta-level, while the background indicates the level of the inner node:

$$n: \ \textbf{disembed}\ \{content \mapsto c, levels \mapsto l\}$$

$$\rightarrow \textbf{border}\ \{weight \mapsto 1,\ margin \mapsto 3,\ border \mapsto b,\ fill \mapsto f,\ content \mapsto c\}$$

$$b = \mathbf{bordercolor}_{\mathbf{level}[n]}$$

$$f = \mathbf{fillcolor}_{\mathbf{level}[c]}$$

$$\mathbf{level}[c] \leftarrow \mathbf{level}[n] - l$$

The remaining nodes of the **expr** language require no special handling, because they do not affect the mode or meta-level of their sub-expressions.

A handful of additional nodes are provided in **view** for handling some of the needs beyond what **expr** can do. A **section** arranges its children in a left-aligned vertical stack. A wider space node, **quad**, is useful for indentation.

The reduction described in this section is implemented directly in Clojure, not in a Lorax grammar, because the grammar language does not currently provide any way to declare attributes. That might be a useful extension; if both inherited and synthesized attributes [**?**] could be defined, it would be possible to write type systems for expressions. However, the burden of implementing this particular reduction manually is modest because the **expr** language is small and not meant to be extended.

The smaller modes quickly become hard to read for anything but very simple expressions (especially on relatively low-resolution displays), so wise language designers (and programmers) will use them sparingly. The TEXbook [**?**] gives much good advice about how to use them effectively. Some clear applications of the smaller modes are for constant ratios and exponents, and for decorating variable names with small integer subscripts and "prime" symbols, as in $x' = \frac{1}{2}x^2 + a_0$.

4.3.2 Rendering the *view* Presentation Language

The **view** language is consumed by a *renderer* written in Clojure using the graphics primitives of Java2D [**?**]. The renderer recursively inspects the tree, calculating node sizes and layout, and then actually drawing nodes. The renderer also provides hit-testing for nodes, identifying a list of **view** nodes which enclose a given point.

For each node, the renderer calculates a width w and height h in (floating-point) pixels. Depending on the node type, there may be a baseline height, b (between 0 and h), which defines the position of the node relative to a common baseline. When a node has children, the renderer calculates a horizontal and vertical position (x, y) for each child, relative to the upper-left corner of the parent node.

For **chars** nodes, w, h, and b values are calculated based on the string and font via Java2D. In general, other nodes' baselines are derived from their children, so that the **chars** nodes in nearby parts of the tree are baseline-aligned.

In a **sequence**, child nodes with baselines are vertically aligned to their baselines, and children without baselines are aligned to their centers (and to the collective center of the baseline-aligned nodes).

The baseline of a **scripted** node is that of its nucleus. An **over** node has no baseline.

The delimiters of a **group** node (e.g. parentheses or brackets) are vertically centered relative to the contents, and the actual glyph is chosen from a list of variously-sized alternatives. TEX's fonts are used for these glyphs, as well as for most symbols reduced from **expr/symbol**.

Any node can have a **color** attribute, which causes the single node to be drawn with a different foreground color.

These definitions are sufficient to get simple expressions to render well, but the current algorithm lacks TEX's second alignment step (the *axis*), so some expressions are not quite properly aligned (e.g. fraction lines are not aligned with the vertical centers of + and − operators).

The renderer makes several passes over the tree to draw different layers. The first pass draws only **border** nodes, which therefore appear "behind" all other elements. The nodes are drawn in pre-order, so that parent borders and their background fill are drawn behind any child borders. On the second pass, an indicator of the selected node is drawn, so that it will appear above the border rects, but behind the actual content. The final pass renders all non-border nodes.

4.3.3 Renderer Implementation

The renderer is written in a simple functional style, freely traversing the node tree as much as necessary. However because some of the necessary calculations are relatively expensive, the renderer initially performed poorly, taking on the order of 2-500ms to render a small program. This was easily improved by taking advantage of the fact that much of the renderer's time is spent in two kinds of referentially-transparent functions. The function that calculates the size of a text glyph given the characters and font is simply memo-ized, so that no more than one call is ever made for each unique glyph. The functions that calculate the position of each node are also cached, but only for the duration of a single rendering pass (mostly to avoid consuming arbitrary memory for no-longer-needed cached values when a succession of updated trees is rendered one after the other). That way the renderer performs well (10-20ms) despite being written in the most direct way, making no attempt to avoid redundant calculations. Clojure's functional nature and macro facility made it easy to implement this algorithmic change "after the fact," without requiring the bulk of the rendering code to be re-written.

The renderer currently violates good functional programming style by performing the actual drawing via side-effecting operations during the traversal of the node tree. In hindsight, it would have been better to have the traversal yield a list of drawing commands to be executed later. For instance the layering of different kinds of visual elements could have been handled declaratively, rather than as a side-effect of traversal order.

4.3.4 Meta-reduction of *expr*

When editing regular programs, the **expr** language is meant to be invisible. When editing a grammar, the **expr** nodes of each presentation reduction are the subject at hand, so it's necessary for understanding to make them visible. To accomplish this, another hand-written reduction is used when editing grammars. This reduction decorates each node with a visual indication of its type, location, and size. This reduction is somewhat idiosyncratic in that it never actually replaces

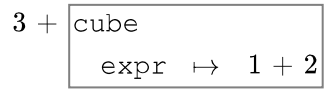


Figure 4.3: Program fragment using a **cube** node, which hasn't been defined.

a node, but rather it inserts additional nodes (super-scripted **border** nodes), which surround the original nodes. Also, this reduction needs to be restricted to operate only on **expr** nodes of the original program, and not any nodes that might have been introduced by a reduction that was applied earlier. The reduction function is wrapped in a higher-order function that tracks which nodes remain to be reduced, using the mechanism described in the previous section.

The result can be seen in any of the examples of presentation reductions presented earlier. The idea of using a higher-order function (combinator) to assemble multiple simple reductions into something more complex came up again and again in building the prototype.

4.3.5 Fallback Presentation Reduction

Another useful reduction is one that is able to reduce any arbitrary node, imposing no restrictions whatsoever on the type or value of the node. This reduction can be used to show the internal structure of a node, disregarding the presentation reduction, or for displaying nodes for which no presentation reduction is available. This might be because the node's type is not declared in any applicable grammar, or because the presentation reduction itself is missing or unusable. All of these scenarios represent situations that Lorax aims to handle gracefully, because all of them can arise in the process of editing a program and/or the grammar that defines its language.

The solution is a *fallback reduction*, which takes any node and reduces it to a node in the **view** language. A primitive-valued node is shown with a default style (similar to **expr**'s int, string, or name). A map- or sequence-valued node is surrounded by a border to set off the node as special, with the node type at the top and a series of lines with the attributes/indices and their values.

In Figure **??**, the programmer has attempted to write $3+(1+2)^3$, but has used a "cube" node type for which no reduction is present. The fallback reduction makes the structure of the program

clear in the absence of any specific knowledge of the "cube" node type, and gives the programmer the information she needs to figure out how to fix it. In this case she can either add a declaration for "cube" or change the program to use a different node type. This implementation does a good job of drawing attention to nodes that aren't handled properly and thus need attention, but it may be overly jarring when used within the context of an otherwise valid expression. Therefore, there may be a role for a less obtrusive reduction which shows any properly-defined children and gives an indication that something is amiss.

4.4 Editing

The Lorax editor assembles the components described in the previous sections, along with a GUI shell. Each program is displayed in a separate window within a single process. When a grammar is used in the rendering of a program, changes to the grammar are reflected in the program's display immediately. Grammars and other kind of programs are presented and edited in the same way, the only difference being that a different sequence of reductions is applied. Neither the UI nor the programs themselves currently provide any way to specify the reductions that are to be applied to a particular program; it's up to the user to do that when invoking the editor.

An editor window (see Figure **??**) shows a program in a scrollable panel, accompanied by some additional information about the selected node, if any. Actions to edit the program are available in the menus, or via keystrokes.

4.4.1 Editor Pipeline

To convert a program into **view** nodes for rendering, the editor applies a series of reductions in turn, as diagrammed in Figure **??**. Some of the reductions are baked into the platform, while others are compiled at runtime from the grammars that are specified when the editor is invoked. The first set of reductions takes different program elements from the source language to the **expr** language. A special reduction handles names, and another introduces a **missing** node wherever the grammar in effect defines an attribute (even if optional) but the program doesn't actually

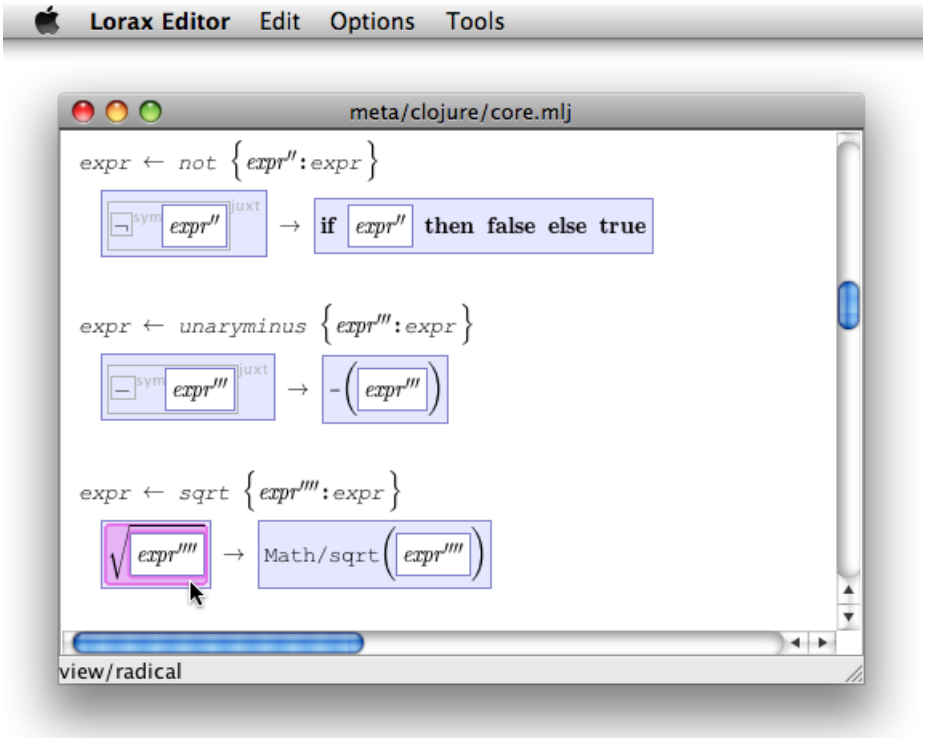


Figure 4.4: The Lorax editor, showing a portion of the grammar for the core language.

contain such a node. Then the reduction defined by the grammar is applied. At this point, all language-specific syntax should have been replaced by **expr** nodes, if the program and grammar are correct.

Next, a pair of reductions produce the final **expr** program, inserting parentheses where they are needed, and reducing any remaining non-**expr** nodes in the generic way. Next the **expr**-to-**view** reduction reduces the program to the **view** nodes that will be rendered, and finally that program is handed to the renderer for layout and drawing.

The editor itself takes charge of executing the reductions, keeping track of how the final **view** nodes were derived from the source program. Thus the end product of the rendering pipeline is a target program in the **view** language, plus a map identifying the target program node that arose from each source program node. Because a source program node often reduces to a sub-tree rather than a single node, not every target program node is identified with a source; only the root of each sub-tree needs to be tracked. The editor uses this tracking of nodes to let the user interact with the

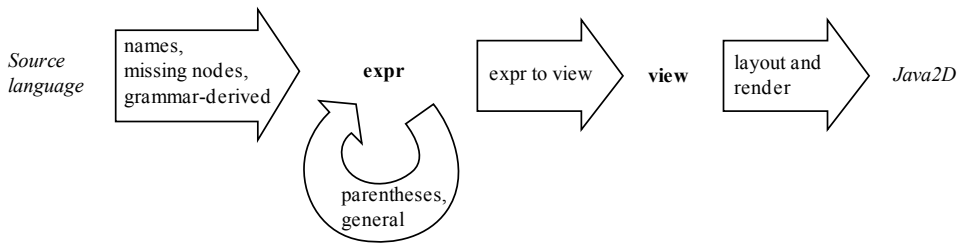


Figure 4.5: Reductions in the editor pipeline.

nodes of the *source* program, even though what's actually being rendered is the target program, many layers of reduction removed.

The need to map target nodes back to source nodes drove many decisions about how to represent programs. The presentation reduction must operate on one node at a time, and must produce a single target node. For example, the arguments of an **app** node are held by a characteristic **args** node mostly so that the editor can identify the resulting sequence of nodes and provide appropriate editing actions on them. For now, **expand** reductions are not so restricted, because Lorax does not try to trace nodes of reduced programs back to the corresponding source node. However, that might become desirable in the future, for instance in order to provide proper source locations for runtime errors.

4.4.2 Selection

One of the main tasks of the editor is to make the structure of the program apparent to the programmer. Rendering the nodes of the program in a rich, familiar notation enhances readability but may not always make the structure of nodes very clear. In order to make a change to the AST, the programmer needs to be aware of this structure. The editor's UI for selection facilitates this awareness by visually emphasizing the relationship between a single selected node and its children, and by providing a way to move between nodes that is defined in terms of the source AST. Thus the structure becomes visible once it is salient, but is left somewhat implicit or even obscured otherwise.

Unlike a text editor, where the typical unit of selection is a character position (i.e. the lowest level of the program text), in Lorax the selection identifies a single source-program node which can be at any level of the source tree. A new node can be selected by clicking anywhere in the view. The new selection node is the deepest node whose boundary encloses the clicked location. Once a selection has been established, a new node can be selected by invoking an action to move the selection to a neighboring node using the notion of relative position defined in the next section. Currently there are actions to select the parent, next or previous sibling, or the first visible child.

The single selected node is the target of all editing operations. Limiting selection to a single node supports many but not all editing operations that might be desired, so a more sophisticated model might support more notions of selection. For example, multiple sequential nodes could be selected and moved as a unit.

Identifying program elements this way effectively exposes the tree structure of the program to the programmer in a tangible way and encourages the programmer to think about the program in terms of this structure. Whether or not this feels natural or is easy to understand represents one test of the overall concept of editing the AST.

4.4.3 Identifying Nodes by Position

Selection operates in terms of source nodes and their relationships to each other, but it also reflects the concrete representation of the program, using the mapping of target nodes to source nodes. Because each target node is identified with a rectangular area of the view, and because each source node is identified with a root target node, the editor can identify the boundary of the visual representation of each source program node. When reduced to concrete syntax, the unordered children of a map-valued node are effectively ordered, and some children may be ignored.

The *parent* node is simply the parent in the usual sense. The editor assumes that every ancestor of every visible node is itself visible. Only the root node has no parent.

The *visible children* of a node are the subset of the node's children for which there is some **view** node, ordered by their visual position. That is, the visible children do not include any child

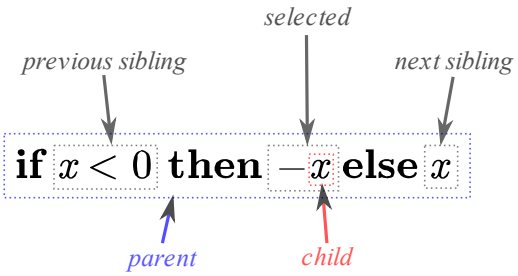


Figure 4.6: A simple expression in Lorax, annotated to show relative node positions.

nodes which were ignored by the presentation reduction, and the otherwise unordered children of a map-valued node are ordered according to how they appear in the visual representation of the node. The order is from left to right (sequences) and top to bottom (sections and fractions). The children of a **subscript** node are ordered as follows: nucleus, superscript, subscript.

The *visible siblings* of a node are the visible children of the node's parent. The *next* sibling is defined in the obvious way for all nodes which are not the last visible child of their parent. The *previous* sibling is similarly defined (except for the first visible child).

Figure **??** shows a simple expression, annotated to indicate the parent, sibling, and child nodes of the node labeled "selected".

4.4.4 Edit Actions

The editor provides a range of operations on source nodes, which the user can use to construct and modify node trees. The examples in the next chapter will show them in action.

There are just two primitive operations.

Delete removes the selected node from its parent. If the parent is a map, it will no longer have any value for the attribute that was occupied by the selected node. If a sequence, the node is removed and any following children move to smaller indices. Note that a simple delete removes an entire subtree, not just the selected node itself.

Insert adds a new node as a child of the selected node with a particular attribute name or

index. If the selected node is a map, any previous child with the same attribute name is replaced. If a sequence, the new child pushes any existing children to the right.

All other edits are composed out of those two primitives:

Replace deletes the selected node and inserts a new node in the same position.

Swap with previous exchanges the places of the selected node and the previous sibling, by a sequence of three replace operations. *Swap with next* is similar.

Insert parent Interposes a new node between the selected node and its parent, by deleting the selected node, constructing a new node with it as a child, and then inserting the new node in the same position.

Copy does not affect the selected node, but saves a reference to it as a separate piece of editor state (the *clipboard*).

Cut copies and then deletes the selected node.

Paste replaces the selected node with the contents of the clipboard.

When a new node is to be introduced, a simple interface is presented to allow the node's type to be specified. Another UI allows primitive values to be edited.

Before each edit operation, the current source AST and the identity of the selected node are pushed onto a stack of previous states. When the *undo* action is performed, the most recent state is popped from the stack and restored. Because each edit involves $O(\log n)$ nodes, this is efficient enough for extended use.

When any editing action changes the source AST, the editor re-reduces the entire program and then redisplays the new reduced target tree. When a grammar is edited, the grammar itself is re-displayed, and then any reductions that were compiled from it are regenerated and any programs using them are redisplayed as well. This environment allows languages to be redefined dynamically.

4.5 Example: Entering Expressions

Here's how the Lorax editor can be used to enter some simple expressions.

Suppose you want to evaluate the expression $(1 + 2) \times 3$. Begin with an empty core-language

program, which is a single **program** node, with no children. The core grammar requires at least one **doc** or **expr** node in the program, so the editor supplies an empty node to start you off:

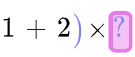
Select the node and type the character '`1`'. The editor infers you want to replace the missing node with a new **int** node in the core language, and does so, leaving the new node selected:

Now we want to add 2 to that, so first type '`+`'. The editor assumes you want to replace the selected node with a new **plus** node, and it adds the selected node as the first child of the new node (this is the *insert parent* action with the type **plus** for the new node). Now the missing right argument is selected:

Simply typing '`2`' completes the first sub-expression:

Now we want to multiply this expression by 3, but you can't simply add the **times** node yet, because that would make a new node with only 2 as the left argument. Instead, use the "select parent node" action (from now on, ↑) to move the selection to the **plus** node (or just click on the + sign):

Now type '`*`' to create a new node. The previous selection becomes the left child, and Lorax inserts parentheses to indicate that the actual grouping of sub-expressions is contrary to what would be suggested by the normal spacing of the operators alone:



Finally, type '`3`' to complete the expression:

$(1 + 2) \times 3$

For this kind of simple expression, the Lorax editor's efficiency is quite similar to that of entering text. In fact the sequence of characters is not much different: '`1+2↑*3`' as opposed to, say, '`(1+2)*3`'. Assuming you use only the keyboard, the actual number of keystrokes is fewer by one with Lorax.

Alternatively, you could have entered the same expression in "top-down" fashion using the sequence '`*+1→2↑→3`'. Interestingly, this seems to mimic the process of typing a prefix expression, with '→' taking the place of spaces between sibling nodes, and '↑' acting as a '`)`' to close one node and return to the parent. I suspect in practice this is more intuitive than it sounds when described in that way, but only experience will tell (just ask anyone who has learned to use an RPN calculator). The editor attempts to support both styles equally well.

Furthermore, Lorax's rendering of the expression includes appropriate spacing, which you might feel the need to add if you were entering text, maybe like this: '`(1+2) * 3`' or '`(1 + 2)*3`', depending on your preference. This kind of manual type-setting is extra work for the programmer (imagine how many hours each year world-wide are spent entering and adjusting white-space!), and it's not very effective anyway. None of the three alternatives is particularly readable.

The host language provides syntax for more specialized purposes also, and these nodes aren't quite so easily accessed. For example, to generate a list of the first 10 perfect squares, you can use a sequence comprehension. Again, starting with a blank program:

?

Use the *insert node* action to create new node, entering the type **for**. The new node includes a variable binding. The editor supplies the default name, x (which we won't change):

? | $x \leftarrow$?

For the expression, first enter a variable reference by typing '`x`'. This inserts a reference to a nearby variable, which happens to be x. Alternatively, hold down the *insert reference* modifier

keys and click on x.

$$x \mid x \leftarrow ?$$

Use the *insert parent* action, entering **square** for the type:

$$x^2 \mid x \leftarrow ?$$

Now move the selection to the next sibling twice (→, →, or just click on the last '?').

$$x^2 \mid x \leftarrow ?$$

Insert another node, this time a **range**.

$$x^2 \mid x \leftarrow ?..?$$

Enter 1 and 10 for the range's min and max:

$$x^2 \mid x \leftarrow 1..10$$

And that's it. Switching to Lorax's evaluation view:

$$\begin{aligned} \$ \quad & x^2 \mid x \leftarrow 1..10 \\ \rightarrow \quad & 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 \end{aligned}$$

In this case, there is some extra effort to enter the **for**, **square**, and **range** nodes. That's the price of being able to add arbitrary nodes to your language. Furthermore, the editor UI could and should be extended to make this penalty as small as possible, for example, since the keystroke '`f`' has no particular behavior, it could offer a choice of all known node types beginning with 'f'.

These examples hopefully give a sense of how an editor can offer both generality and reasonable usability, but the current Lorax editor is certainly only a starting point. Different UI interactions would be appropriate for different kinds of devices (say, emphasizing gestures over keyboard input for a device with a touch-sensitive screen), or for different users (say, something more like a mouse-driven structure editor for end-user programming applications).

Chapter 5

Case Studies

To evaluate the success of Lorax at supporting the addition of new language constructs, I implemented two of them, using the grammar language to extend the Lorax core language. The first is a simple extension of the syntax for constructing lists (one of the basic data types of the core language), and the second introduces a completely new kind of value.

5.1 Defining the Core Language via Syntax Extension

The core language is built via syntax extension on top of the kernel language, so its definitions can serve as a test of the suitability of Lorax's **grammar** facilities for building these kinds of extensions. This section compares one of the declarations from Lorax's core language with the corresponding elements from Lisp and free-form languages in terms the effort invested and the benefit accrued.

Several core language nodes provide support for using the primitive cons-list values of the Clojure platform, which are one of the basic tools for organizing data in any Lisp. These extensions employ a handful of Clojure primitives to expose the native list values of the platform, and the rest of the syntax is built around them.

5.1.1 List Comprehension in Lorax

Figure **??** shows the declarations of the two primitive constructs for working with with lists (**cons** and **match-cons**), and a list comprehension (**for**) node constructed from them. For the

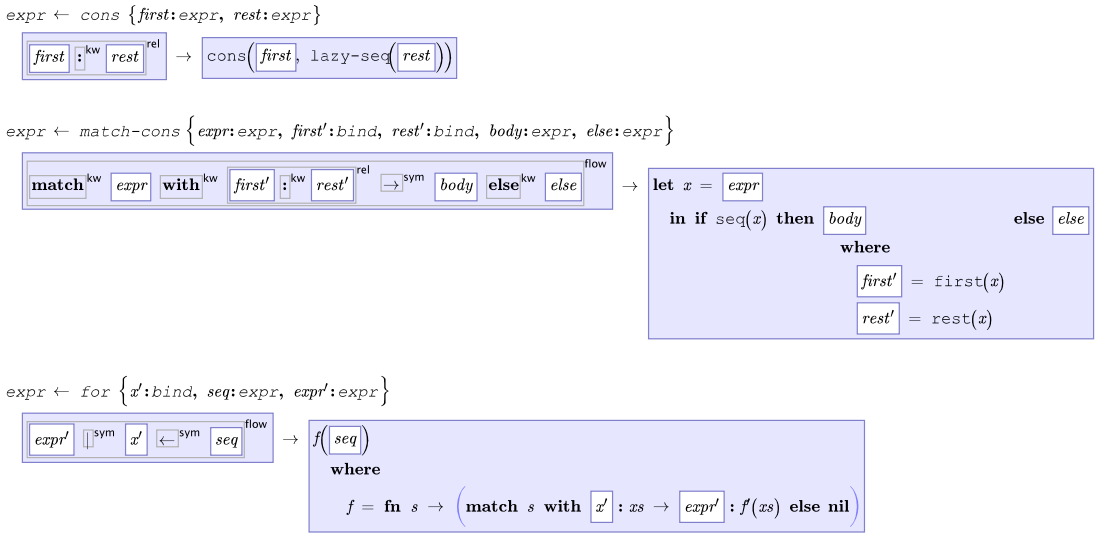


Figure 5.1: Declaration of the **cons**, **match-cons**, and **for** node of the core language.

present purpose, the first two nodes can be regarded as part of the base language, and the list comprehension syntax as an extension to be added to that language.

The *expand* reduction for the **for** node evaluates its **seq** child, and then applies a recursive function to it. This function attempts to match its argument as a non-empty list (using **match-cons**). If so, the **x** child is bound to the first value, and **expr** is evaluated and then the function is applied to the rest of the list. Note that one of the bindings (**x**) was supplied by the programmer, and the other (xs) is local to the reduction.

5.1.2 List Comprehension in Lisp

For comparison, the definition of a Clojure macro with equivalent capability is shown in Figure **??**. The macro operates in a very similar way to the Lorax syntax declaration, so a fairly direct comparison of the two is revealing. The two declarations, and the resulting syntax, differ in several ways.

Both declarations make use of quasi-quoted syntax to construct a reduced program fragment. In Lorax, the rendering of quoted nodes using a contrasting background color provides a clear visual cue of the embedded structure. It's clear at a glance where the three components of the syntax

```
=> (defmacro simple-for
     [[x xs] expr]
     `((fn f# [s#]
         (if (seq s#)
           (let [[~x & r#] s#]
             (cons ~expr
               (lazy-seq (f# r#))))))
         ~xs))

=> (simple-for [x (range 1 11)] (* x x))
(1 4 9 16 25 36 49 64 81 100)
```

Figure 5.2: List comprehension macro in Clojure

are substituted into the reduced program. In Clojure, quotation and un-quotation are indicated with lexical signifiers (as in '`(...)' and '`~x`'), and it's up to the reader to count parentheses to understand what is evaluated when. To avoid unintended variable capture, names introduced in the Clojure reduction are marked with another signifier (as in '`x#`') which causes a new name to be generated each time the quotation is evaluated. In Lorax, no such signifiers are needed, because variable references are unambiguous—the meta-compiler takes care of generating new, unique labels when the quotation is expanded.

Thus the reduction/macro is of roughly equivalent complexity in the either system, but Lorax's handling of quotations and variable references makes the reduction both easier to read (via visual cues) and easier to write (by avoiding subtle issues of variable capture).

5.1.3 Concrete Syntax

In addition to this definition of semantics, Lorax's declaration defines a new syntax for the comprehension, which is designed to be familiar from both the mathematical notation of set theory and the comparable construct in Haskell. In Lorax, the reduction for this syntax is entirely trivial, simply giving the arrangement of the three child nodes and the symbols to be placed between them. The effort involved to provide such a simple reduction is essentially nil. For the programmer using the extended language, this syntax gives list comprehensions a distinct appearance, aiding

comprehension.

Returning to Clojure, the new syntax is identical to any other element of the language, with little more than the name "simple-for" distinguishing it from any other expression. A more illustrative comparison is with Haskell's similar construct. In Haskell, one can write the same example as follows:

```
[ x^2 | x <- [1..10] ]
```

which has the same basic structure as the Lorax version, except that it suffers slightly for being limited to the ASCII character set.

However, Haskell's list comprehension syntax is baked into the compiler, and the programmer has no ability to add a new syntax of this kind without modifying the Haskell compiler's front-end. This is typical of languages with free-form syntax; what the language provides may work well, but the specification of syntax is inaccessible to the user (i.e. it's outside the reach of what can be accomplished with a reasonable effort).

5.1.4 Evaluation

Lorax's facilities for declaring new syntax and specifying semantics and concrete syntax allow new syntax to be introduced with an effort that compares well to Lisp's macro facility. However, both the declaration of the new syntax and its use are significantly more readable than the corresponding Lisp programs, or even the syntax from Haskell which is designed for just this purpose. Meanwhile, the effort involved is much less than the effort would be to add such a construct to a language such as Haskell.

5.2 Introducing a New Runtime Value

The preceding section showed how the facilities of the platform can be exposed and wrapped in a novel syntax. This syntax helps the programmer to understand the program, but as soon as the program is reduced to the kernel language for evaluation, the syntax is gone and only the

primitive values remain. A more ambitious goal for syntax extension is to augment the language with a new kind of runtime value. This allows programs to operate on a new kind of data, and allows the programmer to see results of computation in the natural form. The next example shows how a new kind of value can be introduced, and how it supports writing a program in a much more natural and comprehensible way.

5.2.1 Enumerating the Positive Rationals

In a delightful Functional Pearl [**?**], Gibbons et al. present a series of Haskell programs which generate the infinite series of all positive rational numbers. They begin with the idea of traversing the infinite matrix $a_{ij} = i/j$ which contains every positive ratio, but also contains many equivalent, unreduced ratios (e.g. $1/2$, $2/4$, ...).

The authors show that a series containing all positive rationals in reduced form, without duplicates, is obtained by iterating the function $x' = 1/(\lfloor x \rfloor + 1 - \{x\})$,[1] beginning with $x = 1$. This is a surprisingly simple expression, but it is somewhat computationally undesirable in that calculating $\lfloor x \rfloor$ and $\{x\}$ involves division.

Interestingly, this formula can be implemented using only "a constant number of arbitrary-precision integer additions and subtractions, but no divisions or multiplications" by choosing a different representation for ratios. It happens that the five necessary operations—reciprocal, floor, addition of an integer, negate, and fractional part—can all be efficiently performed on ratios represented as *regular continued fractions*. A continued fraction has the form[2]

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{\cdots + \cfrac{1}{a_n}}}$$

A regular continued fraction is one in which all the coefficients except a_0 are positive, and $a_n > 1$ (except for the special case 1). Every rational has a unique representation as a regular continued

[1] In the authors' notation, $\lfloor x \rfloor$ is the floor, or whole-number part of x, and $\{x\}$ is the fractional part: $\{x\} = x - \lfloor x \rfloor$, $x > 0$.

[2] Incidentally, this expression is a frequently-cited exception to TEX's rules for formatting fractions—all the nested expressions are best typeset at the same size, to emphasize the recursive structure. Lorax does not provide a way to override that behavior, so continued fractions do not look quite this nice in Lorax!

fraction.

Having arrived at this elegant result, the authors proceed to reduce their formulas to the notation of Haskell for implementation, using lists of integer coefficients to represent continued fractions. In the process, the origins of the code are completely obscured by the loss of the original notation. For example, one of four cases for negation of a regular continued fraction looks like this:[3]

$$negatecf\ [n_0, 2] = [-n_0 - 1, 2]$$

It's up to the reader (of the paper or of the code) to decode the representation of fractions being used here and work out how this corresponds to the algebra that motivated it. However, in the proper notation, the same definition reads as simple algebraic equation which is easily understood and checked:

$$-\left(n_0 + \frac{1}{2}\right) = (-n_0 - 1) + \frac{1}{2}$$

The awkwardness of Haskell's notation is an impediment to understanding the program as an artifact, but it also obscures the program's meaning in a more subtle way. The choice of integer lists as a representation, as opposed to defining a new algrebraic data type with a similar recursive structure, was probably driven by the relative economy of the notation for lists (which is provided by the Haskell parser as a special case). As a result, accurate type information is lost, which makes the program harder to understand both in the writing and at runtime.

5.2.2 Continued Fractions in Lorax

In Lorax, one can extend the language with a new kind of value for these fractions. I did this by defining a **continuedFraction** node which defines a recursive data type. It is displayed in the obvious way, except that when the continuation is the "null" value, it is displayed in a slightly simplified form. The *expand* reduction is new—it reduces to an expression which evaluates the component expressions and then constructs a node. Therefore the node itself becomes a runtime

[3] Actually, what's shown in the paper has been pretty-printed for publication [**?**]. In the actual source code, it must have looked something like this: `negatecf [n_0, 2] = [-n_0-1, 2]`.

value. Several **match** nodes provide pattern matching on the runtime shape of the argument, and are used to identify the cases in each operation. Figure **??** shows the declarations of the five operations, and some simple examples of their use are shown in Figure **??**.

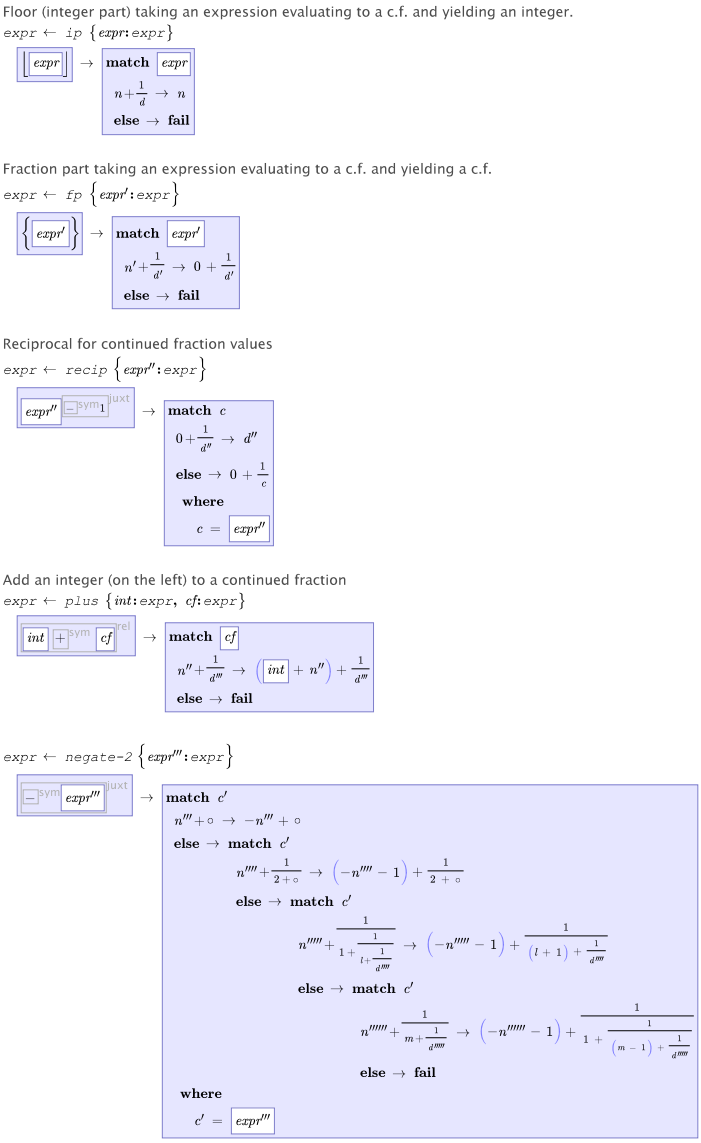


Figure 5.3: Grammar for operations on continued fractions as runtime values.

Note that these declarations are somewhat straining the current capabilities of Lorax. Ideally the construction of a runtime node would be as simple as adding a second level of quotation to the **expand** reduction, but the current prototype does not handle that properly, so a bit of extra ceremony is required. In fact, in order to achieve the relatively understandable reductions shown in

$$\$ \quad \textbf{let } c = 2 + \circ \textbf{ in } \lfloor c \rfloor,\ \{c\},\ c^{-1},\ 1 + c,\ -c$$

$$\rightarrow \quad 2,\ 0 + \circ,\ 0 + \frac{1}{2 + \circ},\ 3 + \circ,\ \text{-2} + \circ$$

$$\$ \quad \textbf{let } d = 0 + \frac{1}{3 + \circ} \textbf{ in } \lfloor d \rfloor,\ \{d\},\ d^{-1},\ 1 + d,\ -d$$

$$\rightarrow \quad 0,\ 0 + \frac{1}{3 + \circ},\ 3 + \circ,\ 1 + \frac{1}{3 + \circ},\ \text{-1} + \frac{1}{1 + \frac{1}{2 + \circ}}$$

$$\$ \quad \textbf{let } e = 2 + \frac{1}{3 + \circ} \textbf{ in } \lfloor e \rfloor,\ \{e\},\ e^{-1},\ 1 + e,\ -e$$

$$\rightarrow \quad 2,\ 0 + \frac{1}{3 + \circ},\ 0 + \frac{1}{2 + \frac{1}{3 + \circ}},\ 3 + \frac{1}{3 + \circ},\ \text{-3} + \frac{1}{1 + \frac{1}{2 + \circ}}$$

$$\$ \quad \textbf{ratio}\lceil 3 + \circ \rceil,\ \textbf{ratio}\left[2 + \frac{1}{3 + \circ}\right],\ \textbf{ratio}\left[1 + \frac{1}{2 + \frac{1}{3 + \circ}}\right]$$

$$\rightarrow \quad 3,\ \frac{7}{3},\ \frac{10}{7}$$

Figure 5.4: Operations on continued fractions.

Figure **??**, I had to manually define a total of seven different *match* nodes, out of a possible 16 for patterns up to three levels deep. It would be much more convenient to have a general pattern-match construct supporting multiple patterns, each a node with bindings substituted for some of the child nodes, but Lorax's current approach to reductions cannot handle that. Nevertheless, with the hard work of defining syntax and semantics out of the way, the actual algorithm can be expressed quite naturally.

The expression which produces the next fraction in the series is wrapped in a function:

$$\mathit{next} = \textbf{fn } c \rightarrow \left(\left(\lfloor c \rfloor + 1\right) + -\{c\}\right)^{-1}$$

Note that in this form the expression does not exactly match what was shown earlier, because some algebraic manipulation is necessary to put it into a form that uses only the operations that have been defined. The resulting expression contains one node (operator) for each operation to be performed. For instance, the original expression hid a negation and an addition operation behind a single $-$ symbol, but my version makes the two operations explicit. Also, Lorax inserts a pair of parentheses to clarify the order of evaluation of the two addition operations, another point which

```
$  ratios[i] | i ← 0..14
     where
       ratios = ratio[r] | r ← rationals
       rationals = next*(1 + ∘)
       next = fn c → (((⌊c⌋ + 1) + −{c})^−1
→  1, 1/2, 2, 1/3, 3/2, 2/3, 3, 1/4, 4/3, 3/5, 5/2, 2/5, 5/3, 3/4, 4
```

Figure 5.5: Enumerating the rationals, using continued fractions.

is left to algebraic convention in the original. Other than that, my choice of notation resembles the original precisely.

Now generating the infinite series in continued fraction form is as simple as applying the *iterate* operator (*) to the *next* function, using 1 as the initial value:

$$rationals \;=\; next^*\big(1 + \circ\big)$$

The complete expression and the first 15 fractions (converted to simple ratios) appear in Figure **??**.

5.2.3 Evaluation

Lorax allows a novel runtime value to be defined entirely in terms of nodes and reductions. Once a constructor and some syntax for pattern matching on the new values are defined, it's quite straightforward to implement operations on the new values, and to write programs which make use of the values and operations on them.

In the case of continued fractions, the notation is clearly superior to what can be done in a textual language, in terms of aesthetics and ease of understanding. Furthermore, the improved notation encourages the use of a proper data type, as opposed to the awkwardness of Haskell which pushes the programmer towards using a generic data type which is slightly more convenient.

One thing to note is the use of identical + symbols for two distinct addition operations (first addition of integers, and then addition of an integer to a continued fraction). It might be preferable to use a different symbol for the new kind of addition. Because that symbol is specified in one place—the presentation reduction for that node—it's trivial to make the switch. On the other

hand, this may be a case where some ambiguity in the visual representation can actually enhance readability. If that freedom leads to an incorrect program, it's easy to click on either node and the editor will indicate which + is of which type. Alternatively, one might want to use the same kind of "plus" node for either operation and have the correct implementation determined automatically (either by analyzing the types or by checking the values to runtime). The latter approach is in fact what Clojure's built-in `+` operator does for integer, floating-point, and rational values, but Lorax does not currently attempt to provide such a mechanism.

However, the current limitations of Lorax's approach to reductions make the job of defining these operations more arduous than it should be. A more general mechanism for implementing pattern-matching is needed to make this attractive.

5.3 Final Notes

Online syntax checking is a significant boon to developer productivity (as evidenced by the general adoption of syntax-highlighting editors and interactive-compiling IDEs), and Lorax provides much of the benefit of this technique through its simple grammar language. However there is one place where this checking is not effective: the contents of quoted nodes may have any type whatsoever, and an un-quote node must be allowed to appear anywhere at all. This is a consequence of the fact that Lorax grammars specify only local structure. For example, to properly constrain the nodes of a presentation reduction, it would be necessary to require that the value produced by the reduction is a node in the presentation language. As it is, this kind of error is not discovered until runtime (i.e. when the editor attempts to apply the reduction). And lacking this kind of information, the editor is not able to provide reasonable suggestions when editing quote nodes. This is a significant gap, but adding what amounts to a type system to Lorax for this purpose alone seemed ill-advised. Of course, many of the languages people want to use are statically typed, so a more complete system will probably need to solve that problem anyway.

The most significant limitation of Lorax as a practical tool is its lack of integration with existing tools. The prototype editor could be developed into a stand-alone tool, and it could

perhaps be used to generate traditional source or object code for use with another system. A more ambitious goal would be to integrate this style of editing into an existing IDE. Ultimately, many other integration points would need to be addressed, including most obviously source code management, but also any and all tools that aim to present or analyze source code.

Chapter 6

Conclusion

In this thesis, I present tools for defining a language in terms of a grammar for ASTs and reduction to a general-purpose presentation language, as well as a structure editor for any such language. The reductions are written in a new meta-language which is defined in the same way, from syntax extension of a minimal kernel language. The Lorax editor provides superior readability of simple programs, plus accurate rendering of mathematical notation. It also eliminates some of the work of entering and modifying programs, offsetting the potential downsides of the structure editing approach.

Moreover, by reducing the difficulty of implementing syntax extensions, and extending the scope of what extensions can do, the system makes the creation of a new language as an extension of a kernel language much simpler and more powerful than before. Much of the work to render and provide editing for language elements is done in a general way, via the presentation language. New languages and new constructs can be specified quite economically as reductions to this language, which provides primitives for the commonly-used visual elements, and can easily be extended with additional symbols.

I believe this expressiveness has the potential to change the way programmers think about their languages. There currently exists a large gulf between what it is possible to do with an "internal" DSL (defined in terms of the syntax of the host language) as compared to a "external" DSL (for which you have to write a parser, etc.) The former is heavily constrained by the host language's syntax, while the latter is a relatively intimidating project. In Lorax, un-constricted syntax exten-

sion allows for much more significant additions the the host language, while its flexibility makes defining a new, separate language much less of a chore. Both options would take advantage of the same tools and methods, making the decision of which way to go much less fraught.

By representing source code in a way that reflects its structure, Lorax also makes it much more available for analysis and inspection. Therefore, Lorax programs can take the place of the algebraic data types or other data structures commonly used to represent programs inside compilers, static analyzers, and other tools. In many cases where it previously wouldn't have been worth the effort to construct a parser, Lorax can provide a rich, readable syntax with very little work. This could be a great help in programming language research and teaching, whenever small languages are defined in the course of exploring some point of language design or implementation.

One intriguing consequence of Lorax's approach is that the visualization of the program no longer has to be unambiguous. Instead, it can take whatever form is most appropriate to the task at hand. In some cases, as in the continued fraction example, this might mean having symbols on the page which look identical but mean different things. This isn't a problem because the editor knows which is which and can clarify when needed, but it does have one serious implication: when the program is printed out (or captured as a figure for publication), that extra information is thrown away, and *the printout does not actually specify a unique program.* To turn a printout back into source code would involve parsing, which is exactly what Lorax scrupulously avoids. However, in Lorax you can always address this problem by simply writing a second presentation reduction which is better suited to your current purpose.

As the old joke goes, text is the worst way of representing source code, except for all the others. My hope is that in time we'll get tools that handle trees well enough that we can finally leave text behind.