RICE UNIVERSITY

Linguistic Reuse

by

Shriram Krishnamurthi
Department of Computer Science
Rice University
Houston, TX 77005-1892, USA

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Matthias Felleisen, Chairman
Professor of Computer Science

Keith D. Cooper
Professor of Computer Science

John E. Dennis, Jr.
Noah Harding Professor of
Computational and Applied Mathematics

Houston, Texas

May, 2001

Linguistic Reuse

Shriram Krishnamurthi
Department of Computer Science
Rice University
Houston, TX 77005-1892, USA

Abstract

Programmers employ a multitude of languages to build systems. Some are general-purpose languages. Others are specific to individual domains. These assist programmers with at least three different tasks: domain modeling, system validation and representing the stucture of their general purpose program. As a result, programming languages have become key factors in the software engineering process. They are, however, rarely codified into the process and treated systematically.

My dissertation develops a framework to treat programming languages as software engineering artifacts. In this framework, languages are identifiable, reusable entities that programmers can compose and link to produce larger languages; furthermore, languages themselves meet the properties of software components. Programmers can augment this lateral growth of languages with vertical growth, by producing languages that synthesize languages. Thus, software construction becomes a multi-phase process. In later phases, programmers use languages to build programs; in earlier phases, they employ languages to construct languages. This treatment of languages as artifacts addresses several open questions.

Acknowledgments

At the chronological end of this document, it is my privilege to offer thanks.

To my parents, for their love, support and sacrifice.

To the teachers who helped me get here, most especially David Chatterjie, John Zachariah, Jeff Nunemacher and Conrad Kent.

To Alejandro Schäffer and Bruce Duba, for tutelage at critical junctures.

To the many past and present members of my research group, PLT. Matthew Flatt, Robby Findler and Cormac Flanagan were teachers as much as collaborators, and their influence pervades my work.

To the people who enrich the department at Rice. Corky Cartwright, Keith Cooper, Peter Druschel and Moshe Vardi contributed immensely to my intellectual growth, while Ann, Darnell, Iva Jean, Rhonda and the others kept the tasks in check.

To the Sunday Morning Brunch crowd: Ian, John, Heidi, Paul, Anika, Gene and Kevin. To others who made the Houston years memorable: Gerald, Phil, Geraldine, Debbie, Margaret and Helga. And to friends from other times and places: Al, Mals, Mihir, Greg and Ram.

To Dan Friedman, Olin Shivers and Phil Wadler, for guidance and perspective.

To David Applegate, Keith Cooper and John Dennis, for serving on my candidacy and dissertation examination committees.

To John Lacey, for kindling flames.

To Kathi Fisler, for patience, indulgence, travels, and the joy of her company.

And to my advisor Matthias Felleisen, who clarified early on the nature of research, periodically corrected my wayward trajectory, assembled a superb group, led by example, and made sure it was always a labor of love. Thank you, amigo.

Contents

Chapter 1

Language: Meaning and Intent

Language is froth on the surface of thought.
—John McCarthy

1.1 The Uses of Language

Programming languages are software engineering artifacts. That is, we can treat languages just like other software engineering artifacts. We can create, link, extend and compose independent fragments of languages. These powers are both valuable and critical. They are valuable because our linguistic needs constantly grow and evolve; they are critical because languages are one of our most effective ways of accomplishing several important programming tasks. As a result, we can produce *linguistic frameworks*, just as programmers currently produce programming frameworks. Programmers can instantiate and specialize these frameworks to produce specific languages, which in turn serve to create actual programs.

The features available in a programming language typically reflect the ends for which the language was designed. Put differently, these features delineate boundaries. They support programs that fall within the boundaries, and complicate those that fall without. These original ends often also control the path of evolution of the language. In short, programming languages mirror and preserve the worldview's of their designers and users.

Programming problems present their own worldviews. These derive from application domains, however, and are thus often different from those of programming languages, and may therefore make demands that cut across languages. As a result, there is often a mismatch between the available languages and a given problem. In these cases, it is usually beneficial to adapt a language to suit the current problem rather than design and implement a complete programming language from scratch.

1.1.1 Genesis: Domain Modeling

Domain modeling is the act of capturing, as closely as possible, information about the problem domain(s). Languages that model domains closely can reduce the potential for programmers to commit errors in the process of transliterating information into data. In the process, they simplify the task of the domain expert—who is not necessarily an accomplished programmer—in capturing domain information. Consequently, they can also present the results of the computation in terms that would be more easily understood by domain experts. In short, domain modeling simplifies the interface between the user and the system.

Historically, many programming languages were designed to be domain-specific, as their etymologies reflect: ALGOL was an algorithmic notation; FORTRAN a notation for describing formulae; COBOL was intended for business applications; and LISP for processing complex symbolic data. Languages evolved to represent broader themes, as programming linguists sought to understand the principles that underlie this diverse collection of languages. This lead to a period of consoliation and generalization.

The benefits of this period are evident: a greater understanding of the unity in the diverse notations, constructs and approaches in these domain-specific languages. But this consolidation has also robbed programmers of the specificity that earlier languages attempted to provide. As programming expands into an increasing number of fields, there is a greater need for languages to more closely reflect the concerns of specific fields.

1.1.2 Analysis: System Validation

The domains of application, and the social demands on software, are growing increasingly complex. Software is expected to protect the privacy of users' identities, the secrecy of their information, and the robustness of their computing environments. These demands have mushroomed as virtually all personal computers have become part of a single distributed (and therefore potentially hostile) network, and the number of such computers grows daily (with new mobile and embedded systems appearing almost continuously). How can a programmer provide such guarantees?

The problem of validating computer systems is essentially two-fold: establishing policies and enforcing them. The creation of policies is a broad, social undertaking, but their enforcement is often a technical problem. Broadly, there are two forms of enforcement: by post-construction analysis or by construction-time restriction.

Analysis is the act of validating that an existing artifact satisfies a given specification. It is typically an *axiomatic* approach that requires the system builder to construct appropriate proof methods that prove that the artifact is faithful (sound, complete, and so on) to the requirements. While some forms of this are relatively simple and in common use—the most frequent likely being type checking—in general, this problem is extremely difficult and, in practice, often intractable. From a linguistic point of view, these checks can be treated as part of the language's specification, and the static and dynamic checks can each be implemented by the static and dynamic semantic portions of the language. For example, a restrictive type system can prevent the construction of programs that do not terminate.

Validation by construction, in contrast, is *constructive*: the means of construction are limited so that only valid artifacts can be built. This requires the system builder to identify such a construction process, prove properties about its validity, and finally using the process to construct the software. A simple example of such a technique is the use of a structure editor, which allows the construction of nothing but syntactically correct programs. A more interesting example is the use of a programming language without arbitrary looping constructs to prevent non-terminating programs.

1.1.3 Synthesis: Representing Solution Structure

Once the programmer constructs an implementation language, accounting for a combination of needs in domain analysis and software analysis, he is ready to build the actual system. The process of doing this may reveal several patterns of use of the language—as distinct from patterns that lead to the description of the language itself—that the programmer may wish to abstract over or even enforce.

In short, this is similar to the language design process, except the resulting tasks must be performed entirely within the language. To describe these abstractions and restrictions, the programmer may construct a linguistic layer atop the chosen implementation language that simplifies and enhances the use of that language.

1.1.4 Multi-Lingual Programming

The preceding discussion mentions a single language, as if building a system is a mono-linguistic activity. In fact, since the number of domains is numerous and each contributes one or more modeling languages, building any sufficiently large system requires the use of several languages, not one. The programmer's task is then not only to use the mix of languages necessary to implement the system, but to coordinate the fragments written in each language. As we shall see, this task can itself induce the design of additional languages.

1.2 Deploying Languages

To specify a language, a programmer must, at the minimum, describe its syntax and its semantics. This is necessary, but not sufficient for deployment: the semantics needs to be in executable form before the language can be used. Equally importantly, the programmer needs the ability to *delimit* the scope of a language's use. A typical and natural delimiter is a boundary of separate compilation, such as a module (while smaller granularities are meaingful, broader ones are difficult to deploy).

A mechanism to deploy languages must necessarily be extensible with respect to the set of characteristics that define a language. In this dissertation, languages consist solely of compilers and run-time systems. In a production system, however, programmers may also want to provide a more formal syntax specification; delcare type rules and other verification conditions; describe language-specific compiler optimizations; and so forth. The mechanism described in this dissertation scales in a natural manner to allow such specifications.

Authors have described domain-specific or "little" languages for decades. Bentley [4] was especially influential in presenting the notion of language as an abstraction that programmers can use to effectively construct software. Unfortunately, his work and much of the literature that follows it has never established precise mechanisms for describing and reusing such languages. This severely limits the use of such techniques in large-scale software development.

Bentley and others have described the use of syntax rewriting tools (such as Awk) to implement little languages. They crucially ignore the relationship between the syntax of languages and their semantics. In particular, they are silent on the difficult question of state in the run-time system. The problem is that, given a source component C, its translated equivalent in some existing language, $T(C)$, must be linked to a run-time system, R. The user must explicitly manage details of whether R must be linked singly or in a shared manner, depending on whether or not there is shared state in R. The implementation of these policies is left to the user, and the implementation technique offers no assistance in the process. This also makes it difficult, if not impossible, to compose languages, due to the potential for interference between the run-time systems.

Beyond these problems, programmers also need tool support to make effective use of a language. The implementation of the language is only one of many such tools, such as syntax checkers, program steppers, type checkers, and so forth. An effective linguistic reuse strategy must make it easy to harness existing tools for related but new languages.

In short, the successful use of language depends on:

- a precise description of the syntax;
- an implemtation of the semantics;
- support for tools;
- facilities for growing and combining languages; and
- the ability to embed one language in another.

1.3 Examples

1.3.1 Automata

Suppose a programmer wants to distribute code that creates and executes finite-state automata. The programmer might wish to define a library of types and procedures, or of classes with methods, that hides the actual representation used for the automata. The information hiding matters since there are at least two significantly different ways of representing automata: as data structures and as executable values. Data structures are useful for manipulating automata as data (e.g., to minimize them). They can also be run via an interpreter, though this can be relatively inefficient. If the automata are being created solely to be run, then it is typically much more efficient to translate them directly into executable code (such as functions or labeled statements).

Hence, the library should offer a way to specify automata in a representation-independent, yet intuitive manner. This would allow the library's client code to contain expressions such as that shown in figure 1.1. The example uses a new construct, **automaton**, to define an automaton that, starting in *1-state*, checks whether

```
(automaton 1-state
  (0-state ((0 ⟶ error-state)
            (1 ⟶ 1-state))
  (1-state ((0 ⟶ 0-state)
            (1 ⟶ error-state)))
  (error-state)))
```

Figure 1.1 : An Automaton Description

```
(lambda (input-stream)
  (letrec
      ((0-state (lambda ()
                  (case (next-token input-stream)
                    ((0) (error-state))
                    ((1) (1-state))
                    (else (raise (new unexpected-input-exn))))))
       (1-state (lambda ()
                  (case (next-token input-stream)
                    ((0) (0-state))
                    ((1) (error-state))
                    (else (raise (new unexpected-input-exn))))))
       (error-state (lambda ()
                      (case (next-token input-stream)
                        (else (raise (new unexpected-input-exn)))))))
    (1-state)))
```

Figure 1.2 : Compiled Automaton Representation

a stream of `0`s and `1`s begins with `0` and then alternates strictly between the two values.

The specification in figure 1.1 might translate into the Scheme [31] code shown in figure 1.2. It represents automata as procedures that consume an input stream. The states are nested, mutually-recursive procedures. Each procedure represents one state's transition relation. All unexpected inputs generate an exception (*unexpected-input-exn*), so the transition relation with no transitions represents an error state. If the automaton attempts to inspect past the end of a finite input stream, a different exception is raised, which a client can handle to observe successful completion. Since programming languages like Scheme optimize calls in tail position to jumps or "goto"s, the state transitions in the example are quick and accumulate no evaluation context (colloquially, "stack space").

Unfortunately, the library programmer cannot define **automaton**, because it is not a procedure. The sub-terms of **automaton** are not expressions in Scheme; rather they are terms in a distinct, domain-specific language. Furthermore, **automaton** is a *binding* construct (as clarified by figure 1.2), which cannot be defined procedurally. Therefore, **automaton** cannot be defined in terms of traditional programming language constructs.

The automaton library might contain other operations of this form. For instance, it may provide a means for interleaving the execution of two automata. This would enable the client programmer to write

(**run/alternating** (*M1 stream-1*) (*M2 stream-2*))

which runs automaton *M1* on stream *stream-1* and *M2* on *stream-2* in strict alternation. In this case, even though both sub-expressions of **run/alternating** look like legal Scheme expressions, Scheme's call-by-value evaluation order would first run *M1* on *stream-1* until termination (which may never occur) before it begins to run *M2*. Thus **run/alternating** also cannot be a procedure in a call-by-value language.

Both constructs are illustrative of a class of useful and important abstractions that

(1) help programmers write software effectively, and (2) are not *procedural*. Sometimes they define notations that are not part of the language's syntax. In other cases, they require behavior that is different from what the language's semantics specifies.

1.3.2 Temporal Programming

Most programming languages provide the programmer with means to modify the current state. Programmers often use mutations as a low level surrogate for a high level notion of *events*. That is, state mutation is used to represent the occurrence of an event, and event observers inspect a location to determine whether the event has occurred. One such event is the progression of time.

Rather than using state for this purpose, a programmer might wish to make time an explicit notion in the language (that is, to build time into the *model* of the language). This work presents a simple example of such a language, in which time is discrete and periodic. Specifically, we add a keyword to the language, **now**, whose value increases by 1 every second. This language has the interesting property that a program such as this

```
(= now (begin
         (sleep 2) ;; pauses for two seconds
         now))
```

evaluates to false, even though the program is sequential and single-threaded!

There are numerous design decisions that arise in implementing such a language. Mundane questions include when time should begin and how it should change. More interesting are whether there should be one clock or many; if several, whether and how they can synchronize; who gets charged for the computational resources consumed by the clock(s); and so forth. We will return to discuss these issues in section 6.1.

```
interface OutputRoutines {
   void drawLine ();
   void printText (); }
class VendorGraphics {
   void drawSolidLine () {};
   void renderText () {}; }
// GraphicalOutput adapts VendorGraphics to OutputRoutines
class GraphicalOutput implements OutputRoutines {
   VendorGraphics lowLevelDriver;
   public void drawLine () { lowLevelDriver.drawSolidLine (); }
   public void printText () { lowLevelDriver.renderText (); } }
```

Figure 1.3 : Adapter Pattern: Java Version

1.3.3 Software Patterns

Languages are also useful to enforce structural properties of programs. This section discusses some samples of structural properties, and how to integrate them into the programming process.

Applied Design Patterns

Design patterns [25] sketch a solution to a class of related problems. By customizing a pattern to a given context, a programmer can reuse the solution. Some patterns represent general "architectures" that can be instantiated effectively in different contexts. Researchers call these architectures *software patterns*.

Figure 1.3 illustrates a sample use of a software pattern in Java. The code in the figure uses the Adapter pattern, which enables the reuse of an existing class with a different interface. More specifically, suppose an existing class implements the desired functionality but does not implement the desired interface. An Adapter acts as a surrogate for this class by implementing the desired interface, forwarding requests to instances of the existing class and tailoring responses to its interface.

In our example, *OutputRoutines* is an interface that represents output generators,

```
(interface OutputRoutines
  (methods (void drawLine ())
           (void printText ())))
(class VendorGraphics
  (methods (void drawSolidLine ())
           (void renderText ())))
(Adapter GraphicalOutput adapts VendorGraphics
                         to OutputRoutines
                         as lowLevelDriver
  (fields)
  (methods
    (public void drawLine () (lowLevelDriver . drawSolidLine ()))
    (public void printText () (lowLevelDriver . renderText ()))))
```

Figure 1.4 : Adapter Pattern: Specification-Based Version

and is implemented by actual generators on various devices. The vendor-provided *VendorGraphics* performs graphical output, but *VendorGraphics* does not implement the interface *OutputRoutines*. In figure 1.3, the programmer creates an Adapter class, *GraphicalOutput*, that forwards requests to an instance of *VendorGraphics*. (For brevity, we elide the details of the actual methods.) The relationship is documented informally through the comment.

The programmer could now make the mistake of defining the class *Client* with the type of some variable *var* to be *VendorGraphics* and of assigning an instance of *GraphicalOutput*:

```
class Client {
    VendorGraphics var = new GraphicalOutput ();
}
```

Unfortunately, *VendorGraphics* is the class being adapted rather than the interface it is being adapted to. The type-checker flags this assignment with the following error message:

```
test.java:13: Incompatible type for new. Can't convert
  GraphicalOutput to VendorGraphics.
   VendorGraphics var = new GraphicalOutput ();
                            ^
```

This example illustrates several effects of programming directly in terms of a pattern's constituent constructs:

- The implementation of the Adapter pattern in terms of plain classes and interfaces obscures the pattern's identity, and thus decreases the clarity of the program. This makes it more difficult for readers to understand the structure and intent of the code.

- It increases the potential for errors due to the volume of code. Pattern code handles administrative tasks such as maintaining invariants, which a programmer must correctly implement during the initial development and remember to update during maintenance. Even in this simple example, for instance, the programmer must remember to make *GraphicalOutput* implement the interface *OutputRoutines*, and declare the type of *lowLevelDriver* as *VendorGraphics*.

- It becomes difficult to replace the code for the pattern instance with an improved version, e.g., one that is more extensible or offers better protection against errors.

- All feedback is reported in terms of the individual units of code, and the programmer must then manually extrapolate from the error message to the original pattern-based design.

In short, programming directly with the constituents of patterns can potentially affect programmers at every stage: implementation, debugging and maintenance. Programmers can reduce the complexity while harnessing the power of patterns by designing languages that express such relationships.

Composition Invariants

Smaragdakis and Batory [49] present several approaches to validating the correctness of module compositions. They express the compositional requirements through the type system. For instance, suppose the implementation of a virtual machine could include a module that tags the memory representations of data (*Tag*) and another that implements garbage collection (*GC*). Furthermore, suppose that the *GC* module requires *Tag* to be present in the module composition. To ensure that the *Tag* module has already been added, *GC* declares a dummy variable of type *TagIncludedProperty*, which is the name of an empty class declared only by the *Tag* module. Thus, if *Tag* is not included in the composition hierarchy before *GC*, the dummy variable declaration will raise a type error, which indicates an erroneous composition.

An alternative is to implement this approach via explicit constructs for validating compositions. This offers many advantages:

1. The class and variable declarations play no role in the execution of the program. Therefore, they obscure its behavior, potentially making it more difficult to maintain.

2. Some of these invariants are tricky to implement. It is thus safer to have them implemented automatically by a program than manually by the programmer. This is especially important since some negative properties require a quadratic number (in the size of the program and the number of properties) of dummy variable declarations. Also, when a new property is introduced, the programmer has to add it to each class by hand. With a properly designed language extension, the programmer can instead establish a single point of control and greatly reduce the work required to specify the program's properties.

3. When an invariant is violated, the programmer should get an error in terms of the actual invariant, not just in terms of the (often mangled) names used to represent it in the source. Smaragdakis and Batory identify this issue as one of

```
(slide-show
  ...
  (slide "The Setup"
   (p ()
      "A role-playing game with three kinds of characters: "
      "princes, princesses and frogs")
   (p ()
      "All are subjects in a mythical kingdom")
   (p ()
      "Both the population and actions will grow over time"))
  ...)
```

Figure 1.5 : A Presentation Description

the major problems in their approach [49, pg. 562].

4. While adding the dummy variable declarations, a programmer might accidentally choose a name that is already in use for a different purpose.

Other Specifications

Several other specification techniques fall under this rubric. For instance, abstract structure operators [47] are used to simplify the process of describing traversals over recursively-defined types. They are generated automatically from the type specification. Instead of defining recursive procedures over these types, programmers use and compose these high-level operators, which are then translated into regular recursive procedures. The translation again raises the issues of name-clash management and feedback-reporting, which can be handled by this framework.

1.3.4 Presentation Software

Presentations are programs. They contain data, namely the content of each slide, and they require control: the transitions between slides, and the animations on each of them. Most commercial presentation software buffers users from the linguistic nature of a talk. The most powerful presentation tool would, however, be one that presents the user with direct, programmatic control over the content of the talk.

A talk program consists primarily of two constructs, one each for the data and the control. The data take the form of slides, so we model each slide as

(**slide** ⟨*title*⟩ ⟨*content*⟩)

For simplicity, we use a stylized form [38] of HTML to represent the content of each slide. A slide show is a sequence of such slides:

(**slide-show** ⟨**slide**⟩ $\cdots$)

The **slide-show** construct creates a new presentation window and renders each slide in turn to the window. Figure 1.5 contains an excerpt from a slide show.

A presenter might wish to combine various existing talks to create a new presentation. To do this, the presenter can employ the module system of the underlying language to organize the fragments. A naïve implementation of this system can lead to unintended results, however. If each **slide-show** constructs a new presentation window, then a presentation composed of several existing shows would be disjointed and unwieldy. Therefore, the programmer who constructs the presentation language needs to create a single shared state—in this case, the presentation window—that all modules containing slide shows can reference. Arranging such sharing is sometimes tricky and often ad hoc, yet it remains a crucial need for many language constructs.

1.4 Summary of Criteria

The many constructs described above, such as **automaton** and **run/alternating**, are all simple examples of *linguistic* abstractions. These typically have the following main uses:

- to affect the order of evaluation. For instance, **run/alternating** requires delayed evaluation in a call-by-value language. Its implementation (in Scheme) can wrap the expressions in procedures that are invoked to control stepping.

- to create new binding constructs. The translation of **automaton** into procedures turns the names of states into binding and bound occurrences of variables.

- to mask the creation of a data structure. It is straightforward to implement **automaton** so that it produces a traditional data structure representation of automata.

As the **automaton** example illustrates, a single abstraction may require many of these capabilities simultaneously. This is especially likely to happen in the case of *data languages*, whose constructs are supposed to mask the representation of the data from the client programmer. Linguistic abstractions are especially important in this context, since they are often the only way to mask the concrete representation, thus ensuring *representation independence*. In most languages, for instance, it is impossible to write an expression that dynamically generates a procedure. This makes it extremely difficult, if not impossible, to write the **automaton** abstraction through any other means, without resorting, at least in a limited sense, to interpretation.

1.5 Dissertation Overview

Having introduced and motivated linguistic abstractions, we can now ask:

- How can we describe broad classes of linguistic abstractions?

- How do we group linguistic abstractions and delimit their use?

- How can we compose and reuse these groups of abstractions?

- How can we reuse existing tools for programming languages—such as compilers and type-checkers—for language extensions?

The following chapters address these questions.

Chapter 2

Linguistic Abstractions and their Implementation

Epigrams are macros, since they are executed at read time.
—Alan Perlis, *Epigrams on Programming* [43]

2.1 Principles

Linguistic abstractions take many forms and are an established part of programmers' toolkits in diverse domains [1, 8, 12, 26, 36, 39, 48, 55]. We shall first consider a particularly useful special case of linguistic abstractions called *macros*. Macros in Scheme are tree transformers that rewrite concrete syntax trees. In traditional Scheme programs, program text is interspersed with macro definitions. A macro preprocessor gathers these definitions, elides them from the source, and expands all uses of the macros until the source reaches a canonical form. In short, the macros serve as a language (extension) specification; the macro processor acts as a compiler from the macro-enriched language to the base language. Macros can express linguistic abstractions such as **automaton** and **run/alternating**, described in section 1.3.

2.2 McMicMac

McMicMac is a framework for creating languages such as that of automata. We explain McMicMac through a series of examples reflecting increasingly complex protocols. The examples are intentionally simplistic to illustrate the principles of McMicMac independently of the details of the actual systems that can be built with these techniques.

```
(define-macro (automaton ⟶)
  (automaton start-state
             (state-name (input ⟶ new-state) ... ) ... )
  ⟹ (syntax
       (lambda (input-stream)
         (letrec ((state-name
                    (lambda ()
                      (case (remove-token input-stream)
                        ((input) (new-state)) ...
                        (else (raise (make-object unexpected-input-exn)))))) ... )
           (start-state)))))
```

Figure 2.1 : Automaton Macro

2.2.1 Macros

Examples such as **automaton** and **run/alternating** are expressible as McMicMac *macros*. The macros of McMicMac are descendants of those in Lisp and Scheme [31, 51]. They transform tree-shaped data, rather than manipulating flat data like the string-processing macros in the C pre-processor [32]. In short, macros implement a simple form of extensible parsing.

A parser is conceptually a table of rules that map syntactic shapes to code. When the input matches a shape, called a *trigger*,* the parser looks up the trigger's transformation rule, called the *elaborator*, and uses it to produce abstract syntax. The parsing table is traditionally fixed, thus limiting the input language a parser can recognize. Macro definitions add rules to a parser's table. Unlike traditional parse rules, though, macro rules do not directly generate abstract syntax. Instead, they generate terms in the source language. Therefore, the parser must analyze the generated term. This process continues until the program is in a canonical form.

In McMicMac, the programmer defines triggers using a pattern-matching notation,

*A trigger is really just a pattern in the sense of pattern-matching, but we use a different term to avoid confusion with design patterns.

(**define-macro** (**automaton** $\longrightarrow$)
  (**automaton** *start-state*
          (*state-name* (*input* $\longrightarrow$ *new-state*) ...) ...)
  $\Longrightarrow$ (**syntax**
      (*make-automaton-rep start-state*
        (*list* (*make-state-rep state-name*
              (*make-transition-rep input new-state*) ...)
            ...))))

Figure 2.2 : Alternate Automaton Macro

originally due to Kohlbecker and Wand [34]. When an input term matches a trigger, the matcher generates a pattern environment that maps pattern variables to the corresponding source terms in the input. It then invokes the elaborator to generate a source term. This term can be parameterized over sub-terms in the input. The elaborator extracts these input sub-terms from the pattern environment.

Figure 2.1 presents a concrete example: the macro for the **automaton** construct of section 1.3. The keyword **define-macro** is followed by a set of literals (here, **automaton** and $\longrightarrow$) that may appear in the input. The literals are followed by the trigger. All symbols in the trigger that do not appear in the literal set are *pattern variables*. A pattern followed by ellipses (...) matches zero or more instances of the pattern. It binds each pattern variable to the sequence of sub-terms that correspond to the pattern variable's position in the matching instances. Ellipses can be nested arbitrarily deep.

The macro definition specifies an elaborator following the $\Longrightarrow$ keyword. The elaborator uses **syntax** to construct a new source term. The **syntax** form consumes a template and converts the template into a term in the source language by replacing all pattern variables with their bindings from the (implicit) pattern environment. Thus, in the output term, *start-state*, *state-name*, *input* and *new-state* are replaced with the corresponding source text in the input expression, while all other names are inserted literally.[†]

Macros of this sort have traditionally been put to four main uses:

- to create new binding constructs. The procedural translation of **automaton** turns the state-names into binding and bound occurrences of variables.
- to mask the creation of a data-structure. Figure 2.2 shows an alternate implementation of **automaton** that represents automata as data structures.
- to represent structural program properties, such as uses of design patterns.
- to affect the order of evaluation. For instance, **run/alternating** requires delayed evaluation in a call-by-value language. The macro can wrap the expressions in procedures that are invoked to control stepping.

Different applications can choose alternate expansions (using the mechanism described in section 2.2.3) without any intervention from the user who specifies the automata.

2.2.2 Beyond Conventional Macros

Though conventional macros can describe many interesting linguistic abstractions, they are not powerful enough for many other programming tasks. McMicMac therefore generalizes macros in several ways. The key innovation in McMicMac is to equip these generalized macros with attributes to propagate information about the program, and to allow programmers to use these attributes to guide elaboration.

From Expansion to Parsing

Parsers have the type

$$source \longrightarrow \text{IR}$$

[†]This document elides many McMicMac features that are useful in practice; e.g., the macro writer can specify guards on the structure of sub-terms.

```
(define-micro (if)                    ;; for if's
  (if test then else) ⟹ (lambda ()
                           (make-if-IR
                             ((dispatch (syntax test)))
                             ((dispatch (syntax then)))
                             ((dispatch (syntax else))))))
```

Figure 2.3 : A Micro Specification

where *source* is the type of source expressions and IR that of the intermediate representation. Macros, which denote source-to-source rewriting functions, have the type

$$source \longrightarrow source\ .$$

In many cases, McMicMac programmers writing language extensions need the power to generate terms of type IR directly. McMicMac thus allows programmers to create elaborators, called *micros*, that have the type

$$source \longrightarrow \text{IR}\ .$$

In practice, the parsing process described in section 2.2.1 has two parts: the elaborators that create IR and the dispatcher that does pattern matching and invokes an elaborator. If this elaborator is a macro, the dispatcher applies it and parses the output again, repeating this process until the elaborator is a micro. In effect, therefore, the dispatcher, called *dispatch*, has type

$$source \longrightarrow micro\ ,$$

i.e., given a source term, it returns the corresponding micro. Micros are elaborators represented as procedures of no arguments that create IR, whose type we denote as

$$() \longrightarrow \text{IR}\ .$$

(The reason for this seemingly needless level of indirection will become clear in the following discussion.) The output of micros, unlike that of macros, is not expanded

```
(define-micro (if)                ;; for if's
  (if test then else) ⟹ (lambda (env)
                          (make-if-IR
                            ((dispatch (syntax test)) env)
                            ((dispatch (syntax then)) env)
                            ((dispatch (syntax else)) env))))

(define-micro (lambda)            ;; for lambda's
  (lambda vars body) ⟹ (lambda (env)
                          (make-lambda-IR vars
                            ((dispatch (syntax body)) (append vars env)))))
```

Figure 2.4 : Micros with Attributes

again, because it is already of type IR. Put differently, a micro must invoke *dispatch* to reduce source terms to IR.

Figure 2.3 shows the micro definition for a simple conditional construct. Like **define-macro**, *define-micro* is followed by a list of literals and a trigger pattern. To the right of the ⟹ keyword is the specification of the micro's elaborator, a procedure of no arguments. The elaborator uses *make-if-IR* to construct the IR representation of **if** expressions. The invocation

((*dispatch* (**syntax** *test*)))

extracts the source term corresponding to *test* from the pattern environment, uses *dispatch* to obtain the corresponding micro, and invokes the micro—a procedure of no arguments—to generate the IR value for *test*.

Attributes

Suppose a programmer wants a simple value inspection facility. Specifically, the expression (*dump*) should print the names and values of all the variables bound in the lexical scope. Provided we have access to the names of all the variables in that lexical context, the transformation associated with (*dump*) is quite straightforward. McMic-

Mac allows programmers to make this contextual information explicit by associating *attributes* with the dispatcher. Thus a programmer may wish to define the type of a micro to be

$$env \longrightarrow \text{IR}$$

where *env* is the type of the lexical environment. The micro can inspect this environment to determine the names of the bound variables.

This type generalizes to

$$attr \cdots \longrightarrow \text{IR}$$

to indicate that there can be several attributes. Every micro must accept all the attributes, and must propagate them to micro dispatches on sub-terms. Figure 2.4 presents the definition for **lambda** (which affects the set of lexical variables listed in *env*) and a revised definition of **if** (which doesn't).

To use McMicMac, an application must invoke *dispatch* on the source program while supplying appropriate values for all the attributes. The result of invoking *dispatch* is an IR value, which the program can use for subsequent processing. Some applications use the same type for the source and IR, i.e., they only exploit attributes, not the ability to transform representations.

Threaded Attributes

One possible IR to choose may be the set of values in the language. In that case, the "parser" may convert programs to their final answers, i.e., it may really be an interpreter. We would expect McMicMac to deal with such transformations too. They are useful for prototyping small embedded domain-specific languages, or for optimizing code-generators. Attributes can represent various aspects of the language's evaluation. In figure 2.4, for example, the environment maintains only a list of names bound in each context, but in an interpreter, the environment could map names to locations or values.

Non-trivial languages, though, have two kinds of attributes. Some attributes, e.g., environments, are *functional*, meaning they do not represent computational effects. Other attributes, however, are *threaded*. They are affected in the processing of sub-terms, and their order of propagation from the processing of one sub-term to another matters. A canonical example of such an attribute is the store. If the same store were passed to all sub-terms, then side-effects in the evaluation of one would not be visible in the other. Micros therefore return the updated values of threaded attributes along with the IR. Thus the type of a micro in the interpreter implementation can be

$$env \times store \longrightarrow \text{IR} \times store$$

or, in general, micros can have a type with the shape

$$funattr \cdots \times threadattr \cdots \longrightarrow \text{IR} \times threadattr \cdots \ .$$

Once again, it is the micro programmer's responsibility to invoke McMicMac, provide arguments for the attributes, accept the final IR value and the values of the threaded attributes, and process them.

We illustrate this new form of micro with two definitions in figure 2.5 that implement a stateful language in a purely functional manner using the store-passing style technique from denotational semantics. (**set!** is Scheme's assignment statement.) The code uses Scheme's multiple-value facility to return the actual value and the potentially modified store.

2.2.3 Modular Specifications

We have thus far discussed the kinds of transformations that programmers can express. In this section, we discuss how programmers can group these transformations into reusable units.

```
(define-micro (set!)              ;; for set!'s
  (set! var val) ==> (lambda (env store)
                       (let/values ((val-value val-store)
                                    ((dispatch val) env store))
                         (values ;; the value:
                                 (void-value)
                                 ;; the new store:
                                 (extend store var val-value)))))

(define-micro ()                  ;; for function applications—no literals
  (fun arg) ==> (lambda (env store)
                  (let/values ((fun-value fun-store)
                               ((dispatch fun) env store))
                    (let/values ((arg-value arg-store)
                                 ((dispatch arg) env fun-store))
                      ;; functions must return value/store pairs
                      (fun-value arg-value arg-store)))))
```

Figure 2.5 : Threaded Attributes

```
(define scheme-exprs
  (make-vocabulary))

(define-micro scheme-exprs (set!)
  (set! var val) ==>
    (lambda (this-vocab env store)
      ···))

(define-micro scheme-exprs ()
  (fun arg) ==>
    (lambda (this-vocab env store)
      ···))
```

```
(define automata
  (make-vocabulary))

(define-micro automata
    (automaton -->)
  (automaton ···) ==>
    (lambda (this-vocab) ···))

(define-micro automata
    (run/alternating)
  (run/alternating ···) ==>
    (lambda (this-vocab) ···))
```

Figure 2.6 : Vocabulary Specifications

```
(define compiler (make-vocabulary))
(define-micro compiler (let)
  (let ((var val) ...) body) ⟹
    ···)
(define-micro compiler (letrec)
  (letrec ((var val) ...) body) ⟹
    ···)

(define compiler-language
  (extend-vocabulary scheme-exprs
                     compiler))
```

```
(define analysis
  (make-vocabulary))
(define-micro analysis (let) ···)
(define-micro analysis (letrec) ···)

(define analysis-language
  (extend-vocabulary scheme-exprs
                     analysis))
```

Figure 2.7 : Tool-Dependent Expansions

Vocabularies

Most programming languages consist of several sub-languages, e.g., expressions, statements, types, argument lists, data, and so on. The programmer must therefore specify which sub-language a micro extends. McMicMac provides *vocabularies* for this purpose. A vocabulary is a grouping of related micros. All micros in a vocabulary must satisfy the same type signature. Figure 2.6 illustrates the revised declarations from earlier examples. Each micro declares membership in a vocabulary just before specifying its literal set.

The sum of declarations in a vocabulary specifies the syntax and the elaboration rules of a language. Put differently, a vocabulary describes the syntax table that is used by *dispatch*, and must therefore be a parameter to *dispatch*. We update the type of *dispatch* from section 2.2.2 to reflect this:

$$\mathit{source} \times \mathit{vocab} \longrightarrow \mathit{micro}\ .$$

The change in the type of *dispatch* forces us to update the programming pattern for micros. Each recursive call to *dispatch* must pass along a vocabulary, which the invoked micro must accept.

As we describe below, however, a micro may not always know which vocabulary it is in. Micros therefore take the vocabulary from which they were selected as an argument—in figure 2.6, each micro accepts a vocabulary, *this-vocab*, as its first argument—which they use to process sub-terms in the same language; alternatively, they can choose a different vocabulary for sub-terms in other languages. For instance, a function declaration may have some sub-terms in the expression language and others in the language of types. Micros therefore have the type scheme

$$vocab \times funattr \cdots \times threadattr \cdots \longrightarrow \textsc{ir} \times threadattr \cdots \ .$$

Composing Vocabularies

McMicMac allows programmers to extend and compose vocabularies to construct new ones. Thus programmers can divide a language into sets of related features, and compose these features to build a processor for the complete language. A programmer can also create an extension vocabulary that overrides some definitions in a base vocabulary with tool-specific constraints. This explains why a micro may not know which vocabulary it is in; after all, the non-overridden micros of the base are also in the new, composite vocabulary. This scenario is analogous to instance variables in an object-oriented language that reside both in a class and in its extensions.

With vocabularies, we can generate programs in various interesting ways:

- While traditional transformation techniques are limited in where they can be applied—macros and templates are usually restricted to the expression or statement languages—the McMicMac programmer can write transformations for any sub-language. For instance, we have used it to define abbreviations over types and to extend the language of procedural parameters.

- In realistic programming environments, different program-processing tools often have differing views of the underlying language. For example, a compiler might translate the binding construct **let** (which creates non-recursive local bindings) into a local function application while treating **letrec** (which introduces mutually-referential local bindings) as a primitive. In contrast, a polymorphic type inference engine might treat **let** as a core form, while it will transform **letrec** into a more primitive term. These distinctions are easy to express through vocabularies, as shown in figure 2.7. The function *extend-vocabulary* extends the language of its first argument with the triggers and elaborators of the second, overriding clashes in favor of the second.

- Some languages allow programmers to write lexically-scoped macros [31]. This is easy to define in McMicMac. The micro for a lexical macro construct creates a temporary vocabulary, populates it with the local macro, and extends the current language with the new macro. Because these are local, not global, changes, the language extension disappears when the body has been parsed, so terms outside this lexical context are unaffected.

- A programmer can use vocabularies to organize several traversals over the program. Typically, earlier passes synthesize information for later passes. For example, a programmer may want to add first-class closures to an object-oriented language like Java 1.0. The translator that implements this transformation would need (1) to determine the free variables of the closure's body; (2) to create a class to represent the closure and move its definition to the top-level, as required in many languages; and, (3) to rewrite the creation and uses of the closure.

  A series of vocabularies solves this problem elegantly. The first maintains the lexical environment while traversing code (figure 2.4); when it encounters an instance of the closure construct, it traverses the body with a vocabulary that computes the set of free variables. This list of free variables is the IR of this vocabulary. It can then generate the class definition. A threaded attribute accumulates top-level definitions created in internal contexts and propagates them outward. Finally, another vocabulary rewrites the creation and use expressions.

Determining uses can be done either from type information or through a dynamic check, depending on the target language.

- If a larger language is constructed by composing smaller language layers, programmers can define restricted versions of the larger language by leaving out some layers. We have found this ability especially useful in the DrScheme programming environment [17], which presents the Scheme programming language as a sequence of increasingly complex sub-languages. This hides the complexity of the complete language from the student; in particular, it flags terms that are errors in the linguistic subset but that might be legal—though often not what the student expected—in the full language. This provides much better feedback than an environment for just the complete language, and considerably improves the learning experience. In the same manner, a system builder can provide "safe" subsets of a language for security purposes, such as a language sans file-system and system call primitives in which a Web server can evaluate potentially malicious foreign code without fear of destroying local files.

2.3 Experience Summary

McMicMac has been successfully implemented and deployed in the DrScheme programming environment [16]. It implements two features, discussed in section 7, which are exploited by the run-time system. The vocabularies provide teaching languages for students, and a full Scheme language for advanced programmers. DrScheme is in use at over 100 universities and 100 high schools worldwide, and is routinely distributed on several CD-ROMs. Even though the student vocabularies compile programs into the advanced fragments of the language, the source-preservation facilities ensure that beginners never confront these advanced features in their interactions. McMicMac is also employed by other tools in the DrScheme suite, including the compiler, syntax checker, static analysis engine [19], and others.

Chapter 3

Software Components

As Will Rogers would have said,
"There is no such thing as a free variable."
—Alan Perlis, *Epigrams on Programming* [43]

3.1 Principles

Modules are widely used to organize and abstract program fragments. Modules come in many forms, but we are especially interested in those with the following properties:

1. Each module can be compiled separately from all the other modules. Its object code can be distributed without its source code.

2. Each module explicitly states its import dependencies, but does not designate which modules will satisfy the imports. The modules that satisfy these dependencies are selected only at link time.

3. Module linkage is hierarchical.

These criteria call for modules as units of *semantic* reuse, and correspond to the properties of *components* [54].* We will use the term *component programming* to refer to the construction of programs by linking together modules with the above properties.

*Other linguistic constructs such as class extensions can also be components. We discuss these briefly in section section 4.4.

```
(define database
  (unit (import) (export new add lookup all-keys)
    (define (new) null)
    (define (add key value db) (cons (cons key value) db))
    (define (lookup key db) (cdr (assq key db)))
    (define (all-keys db) (map car db))))

(define customers
  (unit (import make-new-db add-to-db) (export mailing-list)
    (define mailing-list
      (add-to-db 'hunoz "root"
        (add-to-db 'hukairs "bubba" (make-new-db))))))

(define spammer
  (unit (import db-keys db-lookup mailing-list from-address) (export)
    (define (spam key)
      (printf "Sending mail from ~s to ~s about ~s~n"
              from-address (db-lookup key mailing-list)
              "MAKE FUNNY MAST!!!"))
    (map spam (db-keys mailing-list))))

(define spam-operation
  (compound-unit (import from-addr)
    (link (DB (database))
          (ADDR (customers (DB new) (DB add)))
          (SPAM (spammer (DB all-keys) (DB lookup) (ADDR mailing-list)
                         from-addr)))
    (export)))

(define sender "spammer")
(invoke-unit spam-operation sender)
```

Figure 3.1 : Units in MzScheme

The combination of macros and components is potent, since they address different and important programming needs. Understanding and specifying their interaction is, however, an open problem. Combining components with macros is not trivial because macros can affect the component properties of modules. Specifically, macros can insert unbound names into the component's source, rendering it impossible to assign meaning to the component. We will explore these problems in detail in the next section.

This document uses MzScheme's [20] **unit** construct [21] as the principal vehicle for *component programming*. Each unit is a separately-compilable code container. Units are closed with respect to their imports, so their meaning is fixed relative to the meanings of their imports. They export values, and exports can be connected to the imports of other units. This process of wiring exports to imports creates *compound units*, which, in turn, are also units. The unit implementation *compiles* each unit upon definition. *Evaluation* is triggered by satisfying the unit's imports and explicitly *invoking* it. The result is the value of the last expression in a unit's body (extended to compound units by the order of linkage).

Figure 3.1 provides an example of the syntax and behavior of units. The unit bound to *database* defines and exports four operations. The *customers* unit has two imports, which it uses to construct a mailing list. The *spammer* unit has four imports, which it uses to generate messages. It exports nothing; instead, when invoked, it prints messages about sending out mail and returns a list of the printing operation's return values.

The compound unit bound to *spam-operation* links these units together. Each constituent unit of a compound unit is assigned an internal label: in this case, DB, ADDR and SPAM. The labels refer to their respective linked units. The compound unit links DB's exports to the other two linked units. A combination of exports from DB and ADDR, along with the compound unit's import (*from-addr*), satisfy the SPAM unit's imports. The link to ADDR uses the external, not the internal, name of the list. Invoking the compound unit evaluates the code in each of the constituent units.

3.2 Interaction with Linguistic Abstractions

In traditional macro systems for languages like Lisp and C, program text is interspersed with macro definitions. A macro pre-processor gathers these definitions, elides them from the source, and expands all uses of the macros in the source. In short, the macros serve as a language specification; the macro processor serves as a compiler from an enriched language to the base language. The resulting program is processed by the standard suite of tools for the base language, such as compilers and program analysis engines.

This phase separation implies that the language of macro definitions is distinct from the base programming language. The macro definition language needs only the ability to deconstruct and construct terms in the base language. The separation is illustrated well in C's macro system, where the macro definition language obeys different lexical conventions than the base language, and the pre-processor is sometimes a separate application, traditionally called `cpp`. Likewise, the language used by C++ templates is essentially a functional language, which is quite different from C++ itself. Even when the macro definition language is the same as the underlying language, as in traditional Lisp implementations, these should really be thought of as two distinct instances of the language running in different phases.

The phase separation does not, unfortunately, fix the semantics of the macro. Many interesting macros expand into uses of language primitives, whose meaning depends on the abstract machine that executes the program, and into user-defined names, whose meaning is not typically specified in the macro. In general, any name that is free in the macro-generated source needs to be given a meaning.

Consider the following macros, which define a stack abstraction in terms of lists:

$$(\textbf{empty-stack}\ S) \quad \Longrightarrow \quad (\textbf{set!}\ S\ (\mathit{cons}\ \text{'}\mathsf{empty\text{-}stack}\ \mathit{null}))$$

represents the empty stack as a list with a token,

$$(\textbf{push}\ V\ S) \quad \Longrightarrow \quad (\textbf{set!}\ S\ (\mathit{cons}\ V\ S))$$

mutates the variable representing the stack, and

$$(\textbf{pop}\ S) \quad \Longrightarrow \quad \begin{array}{l}(\textbf{begin}\ (\mathit{verify\text{-}stack\text{-}non\text{-}empty}\ S)\\ \quad\quad (\textbf{let}\ ((v\ (\mathit{car}\ S)))\\ \quad\quad\quad (\textbf{set!}\ S\ (\mathit{cdr}\ S))\\ \quad\quad\quad v))\end{array}$$

uses an auxiliary procedure, *verify-stack-not-empty*, to check whether the stack is empty and report an error if it is. Encapsulating these macro definitions inside a component highlights several subtle interactions, which give rise to three classes of problems:

Syntax Resolution: Since components are closed, all macros must either be defined within a component or explicitly imported. If the macros are defined within a component, they must be exported so authors of other components can reuse the same macros. Macros cannot be imported or exported, however, because (1) they are syntax transformers, not ordinary values, and (2) they must be available at compile time, whereas exports and imports are resolved at link time.

Name Resolution: Even if the macros can be resolved before compilation, the names they introduce cause difficulties. The expansion of **pop** introduces names like *verify-stack-non-empty*. Since this procedure is not built into Scheme, the expanded code is not closed unless *verify-stack-non-empty* is added to the unit's imports.

Meaning Resolution: The expansion of **pop** also relies on several names built into Scheme, including **let** and **set!**, without associating an explicit meaning for any of those names. If a programmer re-defines or even locally re-binds any of these, the macros can exhibit behavior that differs in arbitrary ways from that intended by their designer. The mutations may be non-local and inadvertent, making programs extremely difficult to debug. This situation is exacerbated in languages like Scheme where even primitive procedures and syntax can be re-defined.

In recognition of these problems, some macro systems implement a property called *referential transparency* [10], which expects free names to be resolved in the scope of a macro's definition, not use. This property does not, however, address many of our problems. First, it cannot guard against mutations. Second, it covers only names, but in our framework the meaning of every part of the syntax, not just identifiers, can be altered. Most importantly, the property is not defined for a modular programming context; indeed, the authors leave the extension for future work. The work on *syntactic closures* [3] has similar goals and similarly fails to address the interaction with components.

An apparent solution to the problem of name resolution is to add all the required names to the component's imports. This closes them syntactically, but:

- the user of the macros must then know all the names the macros will introduce;
- clients of the components, who should not be burdened by its implementation details, must link the names to their implementations;
- if the macros change to introduce a different set of names, both the author and all the clients must be notified of this change; and,
- the implementation component would be instantiated once per client, which is sometimes undesirable for reasons explained in section 6.7.

Worse still, since third-parties control linkage, the macro designer can assume nothing about the meaning of the linked implementations. Linking to the wrong unit would hinder meaning resolution.

Some of these steps could be automated, but the automation fails if components can be first-class values, as units are in MzScheme [20]. In such languages, the programmer can write arbitrary programs that choose which component to use at run-time, and statically determining which one will be used is undecidable. Therefore, it is impossible to statically associate the component with the implementation of the names used by its macros.

Chapter 4

Linguistic Abstractions for Components

There will always be things we wish to say in our programs
that in all known languages can only be said poorly.
—Alan Perlis, *Epigrams on Programming* [43]

This chapter addresses the problems raised in the previous chapter. It presents a model for combining linguistic abstractions in the presence of component programming constructs.

4.1 The Solution

This dissertation addresses the problem of combining linguistic abstractions with components by introducing a new language construct: **unit/lang**. The new construct conservatively extends MzScheme's **unit**s.

Syntax

The syntax of a **unit/lang** declaration is

$$\begin{array}{l}(\textbf{unit/lang}\ L\ (\textbf{import}\ I\ \ldots)\ (\textbf{export}\ E\ \ldots) \\ \quad B\ \ldots)\end{array}$$

where the meta-variables L I and E range over identifiers, and B over expressions. The I ... are the imports, the E ... are the exports and the B ... constitute the body of the unit. The name L specifies the *language* in which the unit is written. A language specifies both the syntax and semantics of the unit's body. A **unit/lang** expression can be used in any position where **unit** expressions are allowed.

Basic Model

We call the administrator for **unit/lang** LINGO. It performs three actions:

Language Registration: LINGO has one *native* language, Scheme, which implements Scheme with unit extensions. Programmers must register all other languages with LINGO.

Compilation: LINGO compiles every non-native language unit it encounters. LINGO examines the output of compilation, so it repeatedly compiles every unit until it obtains a native language unit.

Execution: LINGO can invoke *programs*, which are units without imports. It knows how to invoke programs in native languages. Because of compilation, these are the only units it encounters.

These are the only actions that LINGO performs. All other actions must be defined in some language, and executed inside a unit written in that language. LINGO is therefore much like a rudimentary command shell interface.

Languages

When a user initiates a LINGO session, LINGO loads the languages that programmers have registered to use for unit definitions. For simplicity, we elide the details of how a programmer registers languages with LINGO and assume they are located in a fixed directory. During registration, LINGO could check the language definitions for properties such as type correctness.

Every non-native language definition consists of two units: a compiler unit and a run-time system unit. These units have no imports or exports. When invoked, they each return a single procedural value: the compiler and linker, respectively. LINGO invokes each unit during the language registration phase and stores the resulting compiler and linker in a table with the type

$$LanguageName \longmapsto (Compiler + False, Linker + False)$$

where $LanguageName$ is the name of a registered language, and the types of compilers and linkers are described below. A field of the table is $False$ if the corresponding unit was not provided during registration. Such languages are still useful, because compilation does not rely on the presence of the run-time system, and execution of compiled code can proceed in the absence of the compiler.

> **Rationale.** We implement the compiler and run-time system as units for two reasons:
>
> 1. the definitions of the compiler and linker rely on several definitions, which can be encapsulated conveniently within a unit;
> 2. they can be distributed easily as components.

Compilation

When LINGO encounters

$$\begin{array}{l}(\textbf{unit/lang}\ L\ (\textbf{import}\ I\ \ldots)\ (\textbf{export}\ E\ \ldots)\\ \quad B\ \ldots)\end{array}$$

and L is not a native language, LINGO uses the L compiler to transform this declaration into

$$\begin{array}{l}((\textit{linker-for-language}\ \ L)\\ \ (\textbf{unit/lang}\ \mathsf{Scheme}\ (\textbf{import}\ P_C\ \ldots)\ (\textbf{export})\\ \quad \boxed{\begin{array}{l}(\textbf{unit/lang}\ T_C\ (\textbf{import}\ I_C\ \ldots)\ (\textbf{export}\ E_C\ \ldots)\\ \quad B_C)\end{array}}))\end{array}$$

where the subscript C indicates that the term is generated by the L compiler. The imports $(P_C\ \ldots)$ name the primitives of L.

> **Rationale.** The layer in the expansion that binds the primitives is a unit rather than a procedure because units import references, not values. This permits macro designers to expose side-effects across units in the same language.

The type of an L compiler is

$$Expr_L \ list \times Id \ list \times Id \ list \longrightarrow$$

$$LanguageName \times Expr_{Target} \ list \ \times Id \ list \times Id \ list \times Id \ list\,.$$

The three arguments represent the source component's body, its imports and its exports. LINGO applies the compiler to these three arguments. It returns the result of compilation, which is these five values:

1. the target language of compilation, T;
2. the body of the unit, compiled into T;
3. the names of the imports;
4. the names of the exports; and,
5. the names of the primitive values that L's run-time unit provides.

Compilers are therefore more general than macros, but a collection of macros can be turned into a compiler through a convenient interface; see section 4.3.

Linking and Execution

A **unit/lang** unit participates in two kinds of link operations. The first, which the programmer specifies manually, is traditional unit linking via **compound-unit**, as shown in figure 3.1. In contrast, LINGO automatically links units to implementations of the primitives that macro expansion introduces into the unit. This second type of linking is transparent to the programmer.

The elaboration of a **unit/lang** declaration first extracts the L linker component of LINGO's table: (*linker-for-language* L). The linker is a procedure with the type

$$Unit \longrightarrow Unit\,.$$

It consumes the unit outside the box and links it to the primitives from L's run-time unit. The result satisfies the expected interface and is closed over the values of the primitives.

Automatically linking the values from the run-time unit fixes the bindings of names in the elaborated code. Since these names are transparent to the programmer of the original unit, the programmer's code cannot alter the values bound to these names. The language designer therefore determines the semantics of the language, and the unit programmer cannot inadvertently alter this semantics.

> **Rationale.** Procedures are a natural way to represent both the compiler and the linker. In languages where procedures cannot accept or return units as values, the outer term would have to be replaced directly with code for dynamic linking.

The outermost expressions in the transformation, which represent the curried application of *linker-for-language*, are all Scheme expressions. Since LINGO is a unit manager, not a Scheme evaluator, these expressions fall outside the language it understands. We must therefore make their language explicit in the transformation:

(**invoke-unit**
 (**unit/lang** Scheme (**import**) (**export**)
  ((*linker-for-language* L)
   (**unit/lang** Scheme (**import** P_C ...) (**export**)
    (**unit/lang** T_C (**import** I_C ...) (**export** E_C ...)
     B_C)))))

The outermost unit exists solely to assign meaning to the linkage expressions. Since this unit is transparently invoked, client programmers do not notice its presence. The *linker-for-language* procedure is built into the Scheme language in our implementation. We call this transformed expression the *object form* of the original unit.

4.2 Pragmatics

The **unit/lang** construct has several desirable benefits:

Program Distribution: Programmers can distribute **unit/lang** units in two forms:

Source Code: A source program will compile and run on all implementations that offer the compilers and run-time systems of the languages used in the source.

Object Code: The object form is a closed program that can be compiled and distributed to clients as a black box in some object code format. The instance of LINGO that will run the object code needs only the run-time system units, not the compiler, because the object form contains the result of applying the source language compiler. These run-time systems can be distributed like any other component.

Separate Compilation: Since this design is a conservative extension of units, it inherits all the properties of units such as external linkage specifications, graph-based linking, multiple instantiability and dynamic linking [21]. In particular, because the original units can be compiled separately from each other before linking, so can units defined with **unit/lang**.

Hygiene: Compilers are free to rename the primitives, imports and exports. They may do this because the chosen names conflict with the target language, e.g., an import's name is a reserved word. More commonly, a compiler may α-rename binding and bound instances of variables to avoid inadvertent capture of variables between the programmer's source and that inserted by macros.

This is called hygienic expansion [33]. Supporting hygiene makes the object form slightly more elaborate to ensure that clients cannot detect the renaming:

(**invoke-unit**
  (**unit/lang** Scheme (**import**) (**export**)
    ((*linker-for-language* L)
     (**unit/lang** Scheme (**import** P_C ...) (**export**)
       (**unit/lang** T_C (**import** I_C ...) (**export** (E_C E) ...)
         B_C))))

The syntax (E_C E) in an export specification of a unit renames the internal name E_C to the programmer-specified external name E. Because units export by name, this renaming is necessary to ensure that external clients do not need to handle the internal renaming. Since units import by position, however, a similar renaming at imports is not necessary. This particular choice of renaming exports but not imports is an artifact of the definition of units, and would vary depending on the component construct in use.

Variable Target Languages: A compiler can choose the target language based on the source and generated code. Suppose an optimizing compiler can prove that some of the primitives in a unit are never applied to arguments of the wrong kind. It can then generate code for a language that provides versions of primitives that do not perform safety checks on their arguments. The two different implementations of the target language—one with checked primitives, the other with both checked and unchecked ones—are distinguished by having different names, and presumably also distinct implementations.

Staging Run-Time System State: The **unit/lang** framework intrinsically supports two levels for the run-time system's state: language-wide and unit-wide. Application writers may need to create an intermediate level, whereby some state is shared by some, but not all, units written in a language. They can

```
(unit/lang Scheme+automata (import) (export 0/1-alternator)
           (define 0/1-alternator
             (automaton 1-state
               (0-state ((0 ⟶ error-state)
                         (1 ⟶ 1-state)))
               (1-state ((0 ⟶ 0-state)
                         (1 ⟶ error-state)))
               (error-state error))))
```

Figure 4.1 : Scheme+automata Use

easily express this idea in this framework by creating a new language which trivially maps into the original language, and whose run-time system holds only the state common to the subset of units. This illustrates the broad applicability of languages, and the utility of **unit/lang** even when programmers employ only one general-purpose language.

4.3 Programming with unit/lang

In this section, we present an illustrative example of **unit/lang** in use. We first show how users can program with **unit/lang**, then describe how a language implementor would support these users. We also present the transformed version of our example.

Language Use

The **automaton** form translates its sub-expressions into a group of mutually-nested procedures representing states. The generated code raises an exception on illegal inputs. The type of these exceptions is represented by a class, *unexpected-input-exn*.

Figure 4.1 shows a use of the language Scheme+automata. In figure 4.2, two units in Scheme+automata share values: the one named Producer defines two automata which Consumer consumes and runs in alternation. Whether or not units can share values across unit boundaries in this fashion depends on the semantics that the

```
(compound-unit (import)
               (link (Producer
                        ((unit/lang Scheme+automata
                           (import)
                           (export 0/1-alternator all-0s)
                           (define 0/1-alternator
                             (automaton 1-state
                               (0-state ((0 ⟶ error-state)
                                         (1 ⟶ 1-state)))
                               (1-state ((0 ⟶ 0-state)
                                         (1 ⟶ error-state)))
                               (error-state error)))
                           (define all-0s
                             (automaton 0-state
                               (0-state ((0 ⟶ 0-state))))))))
                     (Consumer
                        ((unit/lang Scheme+automata
                           (import alternator stable)
                           (export)
                           (lambda (stream)
                             (run/alternating (alternator stream)
                                              (stable stream))))
                          (Producer 0/1-alternator) (Producer all-0s))))
               (export))
```

Figure 4.2 : Compounding Scheme+automata Units

```
(unit/lang Scheme+def-syn (import) (export)
           (define primitives '(unexpected-input-exn))

            (define-syntax automaton
              (syntax-rules (⟶)
                ((_ start-state (state-name (input ⟶ new-state) ...) ...)
                 (lambda (input-stream)
                   (letrec ((state-name
                             (lambda ()
                               (case (remove-token input-stream)
                                 ((input) (new-state)) ...
                                 (else (raise (new unexpected-input-exn))))))
                            ...)
                     (start-state))))))

            (define compiler (generate-compiler primitives))

            compiler)
```

Figure 4.3 : Scheme+automata Compiler

```
(unit/lang Scheme (import) (export)
           (define unexpected-input-exn
             (class ···))
           (lambda (u)
             (invoke-unit u unexpected-input-exn)))
```

Figure 4.4 : Scheme+automata Run-Time System

```
((loader-for-language 'Scheme+automata)
 (unit/lang Scheme (import unexpected-input-exn) (export)
            (unit/lang Scheme (import) (export 0/1-alternator)
              (define 0/1-alternator
                (lambda (input-stream) ;; expanded code
                  (letrec
                      ;; same as in figure 1.2
                      (else (raise (new unexpected-input-exn)))
                    (1-state))))                                  ))
```

Figure 4.5 : Scheme+automata Expansion

run-time system gives to the primitive values in the language; we revisit this issue in section 6.7.

Language Specification and Elaboration

Figure 4.3 presents the compiler unit for Scheme+automata. The compiler unit returns the actual compiler. The name of the primitive value, *unexpected-input-exn*, is provided to **generate-compiler**, which generates the compiler proper. Figure 4.4 shows the implementation of the run-time system, which includes the definition of *unexpected-input-exn*, which is eventually linked in as the implementation of the class required by the macro.* The last expression in the run-time system is the linker for Scheme+automata units. It links its argument, a unit, to the implementation of the primitive demanded by the Scheme+automata compiler.

When Lingo encounters a unit written in Scheme+automata, as in figure 4.1, it feeds the unit's source to Scheme+automata's compiler and links the result to the implementation of the primitives. The core of the compiler-generated object form is shown in figure 4.5.

The compiler for Scheme+automata is written in the language Scheme+def-syn, which is Scheme enriched with a macro definition construct. Specifically, Scheme+def-syn provides two new syntactic forms: **define-syntax** [10] and **generate-compiler**. As the Scheme+def-syn compiler parses the Scheme+automata compiler unit's body, it accumulates all **define-syntax** macro definitions in a table. It then uses this table to replace the invocation of **generate-compiler** with an expression that evaluates to a compiler that expands the macros while leaving other code unaffected. This compiler uses Scheme as its target language. The Scheme+def-syn language therefore helps users ignore the difference between macros and compilation by allowing traditional macro definitions to serve as the specification of a compiler.

*The language implementer is responsible for ensuring that the run time system provides the correct values for the primitives that the compiler expects, and in the proper order.

4.4 The Component Menagerie

At its most general, a component is a code container with additional properties. Any component can be written in a multitude of languages, provided it carries a suitable language annotation. As a result, one could imagine a stable of constructs similar to **unit/lang**, one for each component constructor. The general elaboration of such constructs should be similar in flavor to the model presented above, with adjustments for differences between units and these other constructs.

The definer of **lambda/lang**, for instance, would use function application in place of unit invocation to link the implementations of primitives. Since functions in most languages follow a call-by-value discipline, however, this fails to provide the implicit aliasing of values that unit import provides. To address this shortcoming, the implementor of **lambda/lang** would have to implicitly

- box the values exported by the run-time system implementations,
- rewrite the source unit's body to automatically unbox all references to primitives, and
- disallow all mutations to the primitives.

With this transformation, languages whose implementations depend on the aliasing would not need to implement it themselves.

Constructs such as **lambda/lang** can be added to a suitable base language. Consider the following program fragment:

(**unit/lang** *L1* (**import**) (**export**)
  (**lambda/lang** *L2* (x) x))

With the body but not the outer unit expanded, this results in

(**unit/lang** *L1* (**import**) (**export**)
  ((*loader-for-language* 'L2)
   (**unit/lang** Scheme (**import** *L2prims* …) (**export**)

(**lambda** (x) x))))

(assuming the compiler for `L1` preserves the informal meaning of **lambda/lang** and `L2` does not alter the body of the function). Thus, the host language for **lambda/lang** must provide function application, the *loader-for-language* primitive, and **unit/lang** (or use **lambda/lang** with the box semantics described above). A different base language, built around functions rather than units as the core component model, could be similarly implemented.

Chapter 5

Language Reuse, Extension and Composition

Once you understand how to write a program
get someone else to write it.
—Alan Perlis, *Epigrams on Programming* [43]

The description of **unit/lang** in chapter 4 treats language descriptions as monolithic entities. In practice, language implementors will want to describe languages in a modular fashion. They can then compose and reuse existing languages to build new ones, thus performing component-based *language* construction analogous to component-based program construction [54].

Section 2.2.3 described a construct called the *vocabulary*. Vocabularies are modular language descriptions with well-defined composition interfaces. Vocabularies let programmers divide a language into sets of related features, and compose these features to build a processor for the complete language. A programmer can also create an extension vocabulary that overrides some definitions in a base vocabulary with tool-specific constraints.

To combine languages, programmers must combine both their compilers and their run-time systems. For arbitrary languages, the definition of this combination is ambiguous. We thus entrust the task of defining composition to the language designer, extending **unit/lang** as a mechanism for expressing these compositions. In other words, **unit/lang** provides mechanisms atop which language implementors can implement the policies appropriate to their languages.

5.1 Autonomous Languages

The simplest way to reuse a language is to make it the target of compilation of some other language. The new language, which we call *autonomous*, inherits the implementation, and also some of the tool support, of the target language, which can greatly reduce the implementation burden. More importantly, it allows correctness to be guaranteed in a modular manner, since the implementor of the new language can exploit the proven properties of the target language.

This case corresponds to traditional language implementation strategies, e.g., the translation of C++ to assembler or of Scheme to C. An autonomous language does not need to have any syntactic similarity to the target it will compile into. This typically requires the specification of a complete compiler. The only dependency might be in the implementation of the run-time system. Suppose A is compiled to B. Then the run-time system of A is linked to a unit in the language B. As a result, the primitives of A's run-time system must be computationally meaningful when used in a B unit. Therefore, A's run-time system must be implemented in a language whose semantics is compatible with that of B.

5.2 Extension Languages

A second technique for reusing languages is by making the new language an extension of the existing language, whose extended features map into those of the host language.

An extension language shares, and possibly extends, the syntax of an existing base language. Say A extends B. Then A is responsible for remaining faithful to the syntax of B. As a result, the compiler for A may often be implemented using one of two strategies:

- directly, as an extension of A's compiler; or,
- indirectly, employing a language to describe the extensions, delegating the responsibility to be faithful to B's syntax to the implementation language, which

provides a guarantee of preserving the syntax of B.

The run-time system must provide values compatible with linking to and use in an A unit.

In the extreme case, A may not add any new syntactic features at all. What use would such an "extension" be? Not adding syntactic features is not equivalent to adding nothing: A's compiler may list additional primitives, which are defined in A's run-time system. At its most basic, this demonstrates how **unit/lang** encompasses the most rudimentary form of linguistic extension—the creation of libraries—within the framework.

Creating such libraries provides programmers with an unexpected power: libraries can exploit the property that the run-time system's implementation unit is evaluated exactly once to specify initialization code across all uses. This code can contain mutational operations, since the programmer has a clear guarantee of the number of times it will be evaluated. In contrast, traditional unit linkage makes it difficult to preserve this property: every client linking to the property forces its evaluation, and the only way to avoid this is to know of all the clients in advance, and invert the traditional design by allowing number of evaluations, rather than the level of abstraction, determine the linking structure.

5.3 Language Composition

The model presented in section 4 inhibits certain types of extensibility. Specifically, the compiler unit (and linker unit) returns one procedural value: the compiler (and linker). While this is sufficient for compiling a unit, it does not provide enough information to a program attempting to combine this compiler with that of another language to produce a compiler for the compound language. The compiler unit therefore returns two values: the compiler and a language-specific datum that represents the composition information. The linker unit similarly returns the linker and some composition information. This is exploited by the compilers and linkers of compound languages. Correspondingly, the compiler and linker units also accept one import, a primitive to look up the composition information of a language.

Supporting language combination requires other changes to **unit/lang**. Compilers and run-time systems compose in slightly different ways:

Compilers Compilation must take place in a single step. Since each language may have different syntactic properties, the serial application of compilers is not the same as composing them before applying them to a source program. For instance, consider two extensions to a simple lambda calculus language (with *cons* but without **quote**): the first converts **quote** into a nested sequence of *cons*, while the latter interprets the stack primitives. Consider the program

(**push** (**quote** (push V S)) *stack*)

Applying the stack compiler (which knows nothing about **quote**) transforms this into

(**push** (**quote** (set! S (cons V S))) *stack*)

which is transcribed into the (incorrect) final expression

(**push** (*list* 'set! 'S (*list* 'cons 'V 'S)) *stack*)

whereas the composite compiler should generate

(**push** (*cons* 'push (*cons* 'V (*cons* 'S *empty*))) *stack*)

Therefore, compilers cannot be composed like traditional functions, wherein one compiler processes the output of another. Instead, the code should be compiled once, by a compiler that represents the composition of the individual compilers.

Run-Time Systems Run-time systems differ from compilers in two important ways:

- Since they only provide values for names, they can be applied serialized unlike a compiler.

- Because the run-time system may contain state to be shared between all components written in that language (or composite languages derived therefrom), the run-time system cannot be modified or copied in any manner. Furthermore, each language can have only one implementation of its run-time system.

Therefore, run-time systems can be serialized, but they must also be treated as atomic and unique for each language.

As a result, whereas compilers previously returned a list of identifiers, they now return a list of list of identifiers. Each sub-list of identifiers corresponds to those that will be provided by one run-time system. The new types are

$$LanguageName \longmapsto$$

$$(\langle Compiler, CompilerExt\rangle + False, \langle Linker, LinkerExt\rangle + False)$$

and

$$Expr_L\ list\ \times Id\ list \times Id\ list \longrightarrow$$

$$LanguageName \times Expr_{Target}\ list\ \times Id\ list \times Id\ list \times Id\ list\ list\,,$$

We consider two special cases for language composition: those defined using macros, and those that follow a particular compiler composition policy.

5.3.1 Macros

A macro expander can be viewed as a program with a finite table of rewriting rules which it applies to a program's source until no further rules apply. Under this view, combining macro languages is easy: it corresponds to merging the tables, taking care to detect and resolve any conflicts due to overlap in the triggers of the rules. Combining the run-time systems is also easy, assuming no conflict of names (or that these names are automatically renamed). Therefore, the composition of two macro languages can easily be described using the (revised) **unit/lang** framework itself.

```
;; Exp = ∅

;; interp-1 : Exp ⟶ Num
(define (interp-1 expr)
  (error 'interp "no semantics for ~s" expr))
```

Figure 5.1 : *EditFun*$_1$

```
;; Exp = num (val) | plus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))

;; interp-2 : Exp ⟶ Num
(define (interp-2 expr)
  (cond
    ((num? expr) (num-val expr))
    ((plus? expr) (+ (interp-2 (plus-lhs expr))
                     (interp-2 (plus-rhs expr))))
    (else (error 'interp "no semantics for ~s" expr))))
```

Figure 5.2 : *EditFun*$_2$

5.3.2 Compilers

Writing composable compilers is, in general, an intractable problem. We justify this claim by appealing to the long line of research in composing denotational semantics [9]. With a little discipline, however, programmers can write compilers that are composable with other compatible compilers. We illustrate the programming pattern by presenting a series of *interpreters* for a simple arithmetic language. While the domain-specific details—those of computing an answer as opposed to generating code that computes it—are obviously different for interpreters and denotational semantics, the solutions are similar. This is because the problem with composition arises from the protocol used to assume and yield control between the components, and the same protocol applies to both interpreters and semantics.

```
;; Exp = num (val) | plus (lhs rhs) | minus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))
(define-struct minus (lhs rhs))

;; interp-3 : Exp ⟶ Num
(define (interp-3 expr)
  (cond
    ((num? expr) (num-val expr))
    ((plus? expr) (+ (interp-3 (plus-lhs expr))
                     (interp-3 (plus-rhs expr))))
    ((minus? expr) (− (interp-3 (minus-lhs expr))
                      (interp-3 (minus-rhs expr))))
    (else (error 'interp "no semantics for ~s" expr))))
```

Figure 5.3 : $EditFun_3$

```
;; Exp = ∅

;; interp-1 : Exp ⟶ Num
(define (interp-1 expr)
  (error 'interp "no semantics for ~s" expr))
```

Figure 5.4 : $MidFun_1$

```
;; Exp = ... | num (val) | plus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))

;; interp-2 : Exp ⟶ Num
(define (interp-2 e)
  ((make-interp-2 interp-1) e))

;; Interp = Exp ⟶ Num

;; make-interp-2 : Interp ⟶ Interp
(define (make-interp-2 next)
  ;; the-interp-2 : Interp
  (define (the-interp-2 expr)
    (cond
      ((num? expr) (num-val expr))
      ((plus? expr) (+ (interp-2 (plus-lhs expr))
                       (interp-2 (plus-rhs expr))))
      (else (next expr))))
  the-interp-2)
```

Figure 5.5 : $MidFun_2$ Extension

```
;; Exp = ... | minus (lhs rhs)

(define-struct minus (lhs rhs))

;; interp-3 : Exp ⟶ Num
(define (interp-3 e)
  ((make-interp-3 interp-2) e))

;; Interp = Exp ⟶ Num

;; make-interp-3 : Interp ⟶ Interp
(define (make-interp-3 next)
  ;; the-interp-3 : Interp
  (define (the-interp-3 expr)
    (cond
      ((minus? expr) (- (interp-3 (minus-lhs expr))
                        (interp-3 (minus-rhs expr))))
      (else (next expr))))
  the-interp-3)
```

Figure 5.6 : *MidFun*$_3$ Extension

```
;; Exp = ∅

;; interp-1 : Exp ⟶ Num
(define (interp-1 e)
  (pre-interp-1 e pre-interp-1))

;; Interp = Exp × Interp ⟶ Num

;; pre-interp-1 : Interp
(define (pre-interp-1 expr int)
  (error 'interp "no semantics for ~s" expr))
```

Figure 5.7 : *Fun*$_1$

```
;; Exp = ... | num (val) | plus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))

;; pre-interp-2 : Interp
(define pre-interp-2
  (make-interp-2 pre-interp-1))

;; interp-2 : Exp ⟶ Num
(define (interp-2 e)
  (pre-interp-2 e pre-interp-2))

;; Interp = Exp × Interp ⟶ Num

;; make-interp-2 : Interp ⟶ Interp
(define (make-interp-2 next)
  ;; the-interp-2 : Interp
  (define (the-interp-2 expr int)
    (cond
      ((num? expr) (num-val expr))
      ((plus? expr) (+ (int (plus-lhs expr) int)
                       (int (plus-rhs expr) int)))
      (else (next expr int))))
  the-interp-2)
```

Figure 5.8 : Fun_2 Extension

```
;; Exp = ... | minus (lhs rhs)

(define-struct minus (lhs rhs))

;; pre-interp-3 : Interp
(define pre-interp-3
  (make-interp-3 pre-interp-2))

;; interp-3 : Exp ⟶ Num
(define (interp-3 e)
  (pre-interp-3 e pre-interp-3))

;; Interp = Exp × Interp ⟶ Num

;; make-interp-3 : Interp ⟶ Interp
(define (make-interp-3 next)
  ;; the-interp-3 : Interp
  (define (the-interp-3 expr int)
    (cond
      ((minus? expr) (− (int (minus-lhs expr) int)
                        (int (minus-rhs expr) int)))
      (else (next expr int))))
  the-interp-3)
```

Figure 5.9 : *Fun*$_3$ Extension

Specifically, we present a series of interpreters for a series of three languages. The first language in the sequence is the empty language; for all inputs, it flags an error. The second language introduces numbers and addition. The third adds subtraction to the set of operations. We present interpreters written in Scheme extended with a mechanism called **define-struct** for defining structures [20].

The first sequence is called *EditFun*. The language of $\mathit{EditFun}_1$ (figure 5.1), the first interpreter, is empty. Therefore, it raises an error for all inputs. The second interpreter, $\mathit{EditFun}_2$, shown in figure 5.2, understands numbers and an addition operation. Finally, $\mathit{EditFun}_3$ (figure 5.3) also processes subtraction.

The *EditFun* sequence does not reuse code, even though much of it is repeated from one stage to the next. To remedy this we create a second sequence, *MidFun*, which reuses existing interpreters and only adds the implementation of new operations. The first interpreter (which is identical to $\mathit{EditFun}_1$) is shown in figure 5.4. Figure 5.5 and 5.6 present extensions which handle the added language features. In each extension, the outer function—*make-interp-2* in figure 5.5 and *make-interp-3* in figure 5.6—accepts an argument that represents the next interpreter to invoke for unrecognized terms. The inner function defines the interpreter proper.

Unfortunately, though the *MidFun* sequence of interpreters is better from the perspective of reuse, it is not equivalent to *EditFun*. Specifically, the term

```
(interp-3 (make-plus (make-minus (make-num 1)
                                 (make-num 2))
                     (make-num 3)))
```

results in an error in MidFun_3 but not in $\mathit{EditFun}_3$ because the recursive calls which evaluate the arguments to the addition operator in MidFun_2 invoke *interp-2*, which cannot handle subtraction. Therefore, MidFun_2 is not an extensible version of the $\mathit{EditFun}_2$ program with respect to $\mathit{EditFun}_3$.

The solution to this problem is presented in the *Fun* sequence (figure 5.7, 5.8 and 5.9). The code in this sequence is similar to that in *MidFun* but each inner interpreter accepts *two* arguments. The first argument is the expression, as before. The second argument is the interpreter that should be used for all recursive calls. It is also passed to the interpreter invoked for unrecognized terms. *pre-interp-2* and *pre-interp-3* are the values corresponding to the inner function declarations.*

This programming pattern emulates two properties:

inheritance The *next* argument of each interpreter dictates which interpreter's behavior it inherits. Technically, this construction implements an "extensible conditional", which object-oriented languages provide automatically by means of inheritance and dispatching.

modularity The interpreter provided as a second argument to each interpreter is expected to handle the entire language. Thus, interpreters do not need to be aware of the rest of the language, so long as they are given an extended interpreter and pass it appropriately when they make calls.

We can now show that the *Fun* sequence is extensible with respect to the members of *EditFun*. This validates the design pattern for extensible functional interpreters.

The preceding discussion has discussed the problem in the context of Scheme as a canonical functional programming language. These problems arise equally in the context of other languages and paradigms, such as object-oriented programming. The same problem arises in Java, for instance, when using the Visitor pattern [25]. To avoid the casts required to circumvent Java's lack of type genericity, we present a non-numeric scenario; in this instance, a simple fantasy game that adds players and actions over time. Figure 5.10 presents the initial game (in place of the initial interpreter); figure 5.12 shows the addition of a kind of player (a type extension equivalent to adding a new abstract syntax form); and figure 5.11 shows the addition

*The successful extensibility of Emacs [50], which is one of the most widely used extensible products, uses "hooks" to enable extensions. One can understand hooks as a weak form of the *next* protocol in the functional pattern.

```
interface SubjectV {
    abstract Object forPrince (Subject s) ;
    abstract Object forPrincess (Subject s) ;
    abstract Object forSpellbound (Subject s) ;
}

abstract class Subject {
    public String name ;
    abstract public Object dispatch (SubjectV v) ;
}

class Prince extends Subject {
    Prince (String s) { name = s ; }
    public Object dispatch (SubjectV v) { return v.forPrince (this) ; }
}

class Princess extends Subject {
    Princess (String s) { name = s ; }
    public Object dispatch (SubjectV v) { return v.forPrincess (this) ; }
}

class Spellbound extends Subject {
    Subject sub1, sub2 ;
    Spellbound (String s, Subject s1, Subject s2) {
        name = s ;
        sub1 = s1 ;
        sub2 = s2 ; }
    public Object dispatch (SubjectV v) { return v.forSpellbound (this) ; }
}

class Speaker implements SubjectV {
    public Object forPrince (Subject s) { return s.name ; }
    public Object forPrincess (Subject s) { return s.name ; }
    public Object forSpellbound (Subject s) { return s.name ; }
}
```

Figure 5.10 : Initial Game

```
class Kissing implements SubjectV {
    Subject kissee ;
    Kissing (Subject k) { kissee = k ; }
    String kiss (Subject kisser)
    { return (kisser.name + " kissing " + kissee.name) ; }
    public Object forPrince (Subject s) { return kiss (s) ; }
    public Object forPrincess (Subject s) { return kiss (s) ; }
    public Object forSpellbound (Subject s)
    {
        Subject kisser1 = ((Spellbound) s).sub1 ;
        Subject kisser2 = ((Spellbound) s).sub2 ;
        return
            (String) (kisser1.dispatch (new Kissing (kissee))) + " and " +
            (String) (kisser2.dispatch (new Kissing (kissee))) ;
    }
}
```

Figure 5.11 : Game Extension: New Behavior

of a new operation (similar to adding a new language-processing behavior). This program contains an error analogous to that in the Scheme fragment. When invoked with the following code,

```
Prince pr = new Prince ("Charming") ;
Princess ps = new Princess ("Leia") ;
Shielded sh = new Shielded ("Valiant", pr) ;
Spellbound sb = new Spellbound ("Perfect", sh, ps) ;
ExtKissing ks = new ExtKissing (sh) ;
sb.dispatch (ks) ;
```

the program aborts with this error:

```
Exception in thread "main" java.lang.ClassCastException: Kissing
at Shielded.dispatch(interp.java:83)
at Kissing.forSpellbound(interp.java:67)
```

```
interface ExtSubjectV extends SubjectV {
      abstract Object forShielded (Subject s);
}

class Shielded extends Subject {
      Subject sub ;
      Shielded (String s, Subject su) {
            name = s ;
            sub = su ; }
      public Object dispatch (SubjectV v) {
            return ((ExtSubjectV) v).forShielded (this) ; }
}

class ExtKissing extends Kissing implements ExtSubjectV {
      ExtKissing (Subject k) { super(k) ; }
      public Object forShielded (Subject s)
      {
            Subject newKisser = ((Shielded) s).sub ;
            return newKisser.dispatch (new Kissing (kissee)) ;
      }
}
```

Figure 5.12 : Game Extension: New Player

```
at Spellbound.dispatch(interp.java:26)
at Main2.main(interp.java:105)
```

The problem underlying both programs is in the interfaces that describe the kinds of information employed by the compiler of each language. This information is fundamentally denotational, or monadic [42, 58], in nature, corresponding to the semantics of the source and target languages of compilation. The problems associated with composing monads [30] apply equally to vocabulary compositions due to their monadic nature.

5.4 Discussion

Studying the composable fragments of languages in light of the properties of components (section 3) reveals an interesting property: the languages are themselves components. Thus **unit/lang** can be viewed as a component constructor for languages, just as **unit** is a component constructor for programs. As a result, we can handle language growth and extension using the familiar principles and techniques that we have already developed for managing programs.

The development of autonomous and embedded languages demonstrates that while the run-time systems have similar characteristics in both cases, the compilers differ greatly in their implementation strategy and approach. Most language development is autonomous. Yet extension languages are deeply valuable because, beside their obvious implementation reuse, they engender a more profound practice: *design* reuse. That is, the experiences, constraints and lessons that inform the design of the base language are automatically inherited by the extending language, and hopefully guide the design of the extensions. (The implementor of an autonomous language, in contrast, views the target language primarily as a source of implementation, not design.) The extension language further benefits by not having to redesign—perhaps poorly, in an ad hoc fashion—the linguistic core that the extended language provides.

An age old question asked of language design is whether or not library design is language design. Because interfaces are themselves little languages, and libraries implement interfaces, library design must be a form of linguistic extension, even if an extremely rudimentary one. In practice, library design is difficult, and the hard problems that library designers encounter—how to balance hiding and exposing functionality, how to preserve invariants, what types to create, export and make opaque, what static and dynamic scope operators to provide, and so forth—are really linguistic ones. So it is heartening to note that the design of **unit/lang** not only encompasses this form of language extension, but blurs the distinction between these two forms in a way traditional language extension mechanisms (such as macro systems, extensible grammars or operator overloading) do not.

Chapter 6

The Expressive Power of unit/lang

> The pun is subversive, as Shakespeare's Fools know, because it reminds power that it rests on the instabilities of language, and language is unstable in part because it is so continuously, so democratically in use, like the Vauxhall Gardens.
>
> —James Wood, *The New Republic*

This chapter presents a series of examples illustrating uses of **unit/lang** that highlight different aspects of its semantics and use.

6.1 Temporal Programming

In section 1.3.2 we encountered a language in which time was discrete and periodic, and the current time was bound to the variable **now**. In this section, we explore some tradeoffs in its implementation.

One simple way to implement the language Time is to use a state variable that is updated every second by a timer thread:

```
(define now 0)

(thread
  (lambda ()
    (let loop ()
      (set! now (+ now 1))
      (sleep 1)
      (loop))))
```

The compiler for Time could simply introduce this code into the unit's body, which would bind **now** in the unit's body. This approach has the following shortcomings:

- The base language that Time extends may not have threads, or mutation, or even definitions, in which case this code would fail to function.
- This approach inserts a copy of this code in each Time unit, which is unacceptable if we want all these units to share a single, universal clock.

It is clear from these problems that the implementation of the clock must reside in the Time language's run-time system.

To export the clock's value to the user's code, we might slighly adapt the implementation of the clock:

```
(define now-box (box 0))

(thread
  (lambda ()
    (let loop ()
      (set-box! now-box (+ (unbox now-box) 1))
      (sleep 1)
      (loop))))
```

(Equivalently, the value could be accessed by invoking a thunk.) The language now lists a primitive, *now-box*, and the compiler rewrites instances of **now** in the user's code to (*unbox now-box*). This guarantees that updates to the clock's value are detected by the user, and it does not burden the user with having to know how the clock is implemented (whether with a box, a thunk, or other means). In short, this appears to be a satisfactory solution.

This second solution has a subtle problem, too. The problem is that it converts the user's intent of dereferencing a variable, **now**, into a function application. If the program is running atop an operating system that accounts for resource consumption, it would charge the user's code with function calls that the programmer never

```
(unit (import lookup-language) (export)
  (define now 0)
  (define (start-timer)
    (thread
      (lambda ()
        (let loop ()
          (set! now (+ now 1))
          (sleep 1)
          (loop)))))
  (start-timer)
  (values
    (lambda (u)
      (invoke-unit u now))
    'dummy-rts-info))
```

Figure 6.1 : The Time Language Run-Time System

intended. Under a pricing model in which function calls (quite reasonably) cost more than variable accesses, the user's program may prematurely exhaust its allocated time resource.

Fortunately, the solution to this problem is easy to encode using units, because units export references [21]. Therefore we can restore the original implementation of the clock, eliminate the rewriting performed by the compiler, add *now* to the set of primitives of Time, and obtain the correct implementation of the language. The resulting run-time system is shown in figure 6.1.

6.2 Interaction Languages

The Lingo framework assumes that all code will reside within a unit (section 4.1). In a modern programming environment, this is true of most code that the user writes, which resides within some sort of compilation unit. The one exception is code entered interactively at a read-eval-print loop (REPL). In this section, we address how such code can be treated within the **unit/lang** framework.

It is tempting to wrap every interactive expression the user types into a Scheme unit without imports or exports, invoke it immediately, and print the resulting value (if any). For instance, if the user types in the expression

(+ 1 2)

the interaction manager can translate this into

(**invoke-unit** (**unit/lang** Scheme (**import**) (**export**) (+ 1 2)))

which, like the original expression, evaluates to 3.

This simple solution is insufficient. Scheme units must be closed with respect to their imports and definitions; the expression

(**define** (*sum-of-squares a b*)
  (+ (*square a*) (*square b*)))

fails this criterion (since *square*) is not bound, so the following expression, which would be generated by the REPL, would be syntactically illegal:

(**invoke-unit**
  (**unit/lang** Scheme (**import**) (**export**)
    (**define** (*sum-of-squares a b*)
      (+ (*square a*) (*square b*)))))

Yet a user might enter the above code, followed by

(**define** (*square x*) (* *x x*))

and then legally use *sum-of-squares* in the REPL. Thus, the above transformation fails to capture the semantics of the REPL.

Since some of the individual expressions are not closed but their aggregate often is, the REPL implementor might enclose the entire collection of expressions in a Scheme unit:

(**invoke-unit**
  (**unit/lang** Scheme (**import**) (**export**)

```
    (define (sum-of-squares a b)
      (+ (square a) (square b)))
    (define (square x) (* x x))
    (print (square 13))))
```

This attempt still fails to accommodate any remaining free variables. A bigger concern is what happens when the user types in another expression. If the entire aggregate of expressions is enclosed and invoked again, this repeats all the computation:

```
(invoke-unit
  (unit/lang Scheme (import) (export)
    (define (sum-of-squares a b)
      (+ (square a) (square b)))
    (define (square x) (* x x))
    (print (square 13))
    (sum-of-squares 12 5)))
```

Not only is this wasteful of resources, it also repeats all effects. In the above example, the REPL prints the value of the call to *square* once per expression entered thereafter, instead of only once. Therefore, this approach also fails to capture the semantics of the REPL.

The underlying problem is the faulty assumption that REPL code behaves identically to code within Scheme units. In fact, roughly speaking, Scheme compilers automatically transform all references to unbound identifiers into dereferences in the global environment. This corresponds to a different semantics, which we can capture with a different language: Scheme-interaction.

All Scheme-interaction units must import one value, which corresponds to the top-level environment. The compiler for Scheme-interaction translates code into Scheme with the following changes:

- All references to (non-primitive) global variables are rewritten into dereferences (by name) into the top-level environment.

- All definitions of global variables are rewritten into augmentations of the top-level environment.

This ensures closure of the resulting component, since the unbound identifier names are replaced by calls to a primitive with the *quoted* identifier name supplied as an argument. Definitions, likewise, are translated into an invocation of a primitive. The Scheme-interaction run-time system is responsible for providing these primitives. They can be implemented simply over association lists. The run-time system can also provide primitives to initialize and reset the association list, so the programming environment has the ability to configure the user's interaction space. This not only accurately mirrors the REPL's semantics, it encapsulates precisely the difference between compiled and interactive Scheme by demonstrating a reduction from the latter to the former.

The actual implementation, shown in figure 6.2, is somewhat different because it exploits the built-in namespace facility in MzScheme [20]. When *eval* is given a namespace as its second argument, it performs its evaluation relative to that namespace. Therefore, this language is able to simulate the use of an interactive namespace by exploiting this language feature. All effects automatically register in the namespace parameter, so this conveys side-effects across Scheme-interaction units.

6.3 Languages to Implement Language-Extension Languages

Section 5.2 discusses extension languages, and mentions the potential to use specialized languages to implement compilers for these extension languages. Section 4.3 presented a use of such a language, Scheme+def-syn. This section explains the creation of a similar, but simpler, language, Scheme+defmac. Whereas Scheme+def-syn permits pattern-based macro specifications, Scheme+defmac permits only Lisp-style macros [51], which enables us to focus on the macro facilities and elide the details of

```
(unit (import lookup-language) (export)

    (define (compiler body imports exports)
      (unless (= (length imports) 1)
        (error 'scheme-interaction
               "units must have exactly the namespace as import"))
      (let ([interaction-namespace-name (car imports)])
        (values (map (compile-one interaction-namespace-name) body)
                #f
                imports
                exports
                '(()))))

    (define (compile-one interaction-ns)
      (lambda (e)
        `(parameterize ([current-namespace ,interaction-ns])
           (eval ',e))))

    (values compiler
            'dummy-elab-info)

    )
```

Figure 6.2 : The Scheme-interaction Language Compiler

```
(unit (import lookup-language)
      (export)
      (values
       (lambda (u)
         (invoke-unit u))
       'dummy-rts-info))
```

Figure 6.3 : The Scheme-interaction Language Run-Time System

```
(unit/lang ("defmac-compiler-impl") (import lookup-language) (export)

  ;; ?body is the source containing the macro definitions
  ;; *body is the source containing the macro uses

  (define defmac-elab-info%
    (class* object% (defmac-elab-info<%>) (-body-text)
      (public [body-text (lambda () -body-text)])
      (sequence (super-init))))

  (define *prim-accumulator (gensym "primitive-accumulator:"))

  (define (compiler ?body ?imports ?exports)
    (values (let ([*body (gensym "body:")]
                  [*imports (gensym "imports:")]
                  [*exports (gensym "exports:")]
                  [*namespace (gensym "namespace:")])
              `( see figure 6.5 ))
            #f
            ?imports
            ?exports
            `((,*prim-accumulator))))

  (values compiler
          'dummy-elab-info))
```

Figure 6.4 : The Scheme+defmac Language Compiler (Scaffold)

```
(values (lambda (,*body ,*imports ,*exports)
          (let ([,*namespace (make-namespace 'all-syntax)])
           (parameterize ([current-namespace ,*namespace])
            (send ,*prim-accumulator clear)
            (eval `(define-macro declare-primitives
                     (lambda (new-prims)
                      `(send ,,,*prim-accumulator adjoin ',new-prims))))
            ,@(map (lambda (e)
                      `(eval ',e))
                   ?body)
            (values (map expand-defmacro ,*body)
                    #f
                    ,*imports
                    ,*exports
                    (list (send ,*prim-accumulator get))))))
        ,(make-object defmac-elab-info% ?body))
```

Figure 6.5 : The Scheme+defmac Language Compiler (Core)

```
(unit (import lookup-language)
      (export)

  (define prim-accumulator%
    (class object% ()
      (private [prims '()])
      (public
        [adjoin
         (lambda (more-prims)
           (set! prims (append prims more-prims)))]
        [get (lambda () prims)]
        [clear (lambda () (set! prims '()))])
      (sequence (super-init))))

  (values (lambda (u)
            (let ([prim-accumulator (make-object prim-accumulator%)])
              (invoke-unit u prim-accumulator)))
          'dummy-rts-info))
```

Figure 6.6 : The Scheme+defmac Language Run-Time System

```
(unit (import lookup-language)
      (export)

      (define (compiler body imports exports)
        (values body
                #f
                imports
                exports
                '((defmac-elab-info<%>))))

      (values compiler 'dummy-compiler-info))
```

Figure 6.7 : The defmac-compiler-impl Language Compiler

```
(compound-unit
  (import lookup-language)
  (link [TYPE ((require-library-unit "compiler-core-types.scm"
                                     "unitlang"))]
        [RTS ((unit (import elab-info<%>)
                    (export)
                    (define defmac-elab-info<%>
                      (interface (elab-info<%>) body-text))
                    (values
                     (lambda (u)
                       (invoke-unit u defmac-elab-info<%>))
                     'dummy-rts-info))
              (TYPE elab-info<%>))])
  (export))
```

Figure 6.8 : The defmac-compiler-impl Language Run-Time System

pattern matching.

A unit in Scheme+defmac is written in Scheme extended with the following two constructs:

define-macro describes a new macro form, providing a function that serves as the expander.

declare-primitives names a list of primitives that must then be provided by the run-time system.

The compiler for Scheme+defmac traverses the body, which consists of Scheme code as well as the above two extensions. When it encounters a macro definition, it evaluates the expander expression and places the result in a table. When it encounters a declaration of primitives, it records these in a table. The table is implemented by invoking a primitive provided by the Scheme+defmac run-time system.

The goal of the Scheme+defmac compiler is to generate a compiler from this extended Scheme language to ordinary Scheme (without macros). Having traversed the entire body and accumulated macro and primitive declarations, the compiler replaces the body with an expression that has the type of a compiler. This expression is in fact a compiler that understands the syntactic structure of Scheme, and is also aware of the extensions declared by the language. This expression is the actual compiler for the extended language. Thus, when a program written in this extended language must by evaluated, **unit/lang** invokes its compiler on the program, which translates it into a Scheme program without the extensions. This is then ready to be linked to the primitives of the extended language.

Figure 6.4 presents the compiler for this language, and figure 6.6 presents its run-time system. The compiler is not written in Scheme; rather, it is written in a special-purpose language designed for the purpose of describing this compiler, called defmac-compiler-impl. The purpose of this language is to make available to the Scheme+defmac compiler the *defmac-elab-info%* value, which is intended to be shared between all macro definition units and is used to combine macro languages. The defmac-compiler-impl run-time system loads a system library to obtain the value of *defmac-elab-info%* only once; therefore, this representation is shared across all macro collections.

6.4 Macro Language Composition Languages

Programmers can reuse macro definitions by composing them to build richer languages. They accomplish this by using two languages in concert: one to define the primitive collections of macros, and the other to define the composition languages. These two languages must be defined together because language composition requires a protocol between the primitive and compound languages.

In this section we present a simple macro composition mechanism in the form of two languages, defmac-compound-compiler and defmac-compound-rts. The defmac-compound-compiler language creates *compilers* for composite languages defined using Scheme+defmac. The defmac-compound-rts similarly creates run-time systems.

The body of a defmac-compound-compiler unit is stark: it simply lists a sequence of languages to compose, where each language's name is specified in an implementation-specific way. (The only constraint on the "name" is that it be a valid argument to *lookup-language*.) In particular, the defmac-compound-rts language permits no expressions or computation in its bodies.

The defmac-compound-compiler language's compiler is shown in figure 6.9. Each Scheme+defmac unit generates a table of the macros and primitive names as its elaborator information. The defmac-compound-compiler compiler extracts these tables and composes them to create the compiler for the compound language. Its run-time system (figure 6.11) contains routines to accumulate names of primitives.

The defmac-compound-rts language defines the combined run-time system. Because run-time systems may close over state, they cannot be re-invoked in the process of compounding languages; doing this may, for instance, destroy the ability

```
(unit/lang ("defmac-compiler-impl") (import lookup-language) (export)

  ;; ?body is the list of languages to compose
  ;; *body is the source containing the macro uses

  (define *prim-accumulator (gensym "primitive-accumulator:"))

  (define (compiler ?body ?imports ?exports)
    (values (let ([*body (gensym "body:")]
                  [*imports (gensym "imports:")]
                  [*exports (gensym "exports:")]
                  [*namespace (gensym "namespace:")]
                  [*prims-list (gensym "prims-list:")])
              (let ([compiler-bodies
                     (map
                      (lambda (l)
                        (let ([e (send (lookup-language l) compiler-ext)])
                          (if (is-a? e defmac-elab-info<%>)
                              (send e body-text)
                              (error 'unit/lang
                                "Cannot compose ~s: not in defmac family" l))))
                      ?body)])
                 see figure 6.10 ))
            #f
            ?imports
            ?exports
            `((,*prim-accumulator))))

  (values compiler
          'dummy-compiler-info))
```

Figure 6.9 : The defmac-compound-compiler Language Compiler (Scaffold)

```
`((values
   (lambda (,*body ,*imports ,*exports)
     (let ([,*namespace (make-namespace 'all-syntax)])
       (parameterize ([current-namespace ,*namespace])
         (eval `(define-macro declare-primitives
                  (lambda (new-prims)
                    `(send ,,,*prim-accumulator
                           adjoin
                           ',new-prims))))
         (let ([,*prims-list
                 (list
                   ,@(map (lambda (compiler-body)
                            `(begin
                               (send ,*prim-accumulator clear)
                               ,@(map (lambda (e)
                                        `(eval ',e))
                                      compiler-body)
                               (send ,*prim-accumulator get)))
                          compiler-bodies))])
           (values (map expand-defmacro ,*body)
             #f
             ,*imports
             ,*exports
             ,*prims-list)))))
   'dummy-elab-info))
```

Figure 6.10 : The defmac-compound-compiler Language Compiler (Core)

```
(unit (import lookup-language)
      (export)

  (define prim-accumulator%
    (class object% ()
      (private [prims '()])
      (public
        [adjoin
         (lambda (more-prims)
           (set! prims (append prims more-prims)))]
        [get (lambda () prims)]
        [clear (lambda () (set! prims '()))])
      (sequence (super-init))))

  (values (lambda (u)
            (let ([prim-accumulator (make-object prim-accumulator%)])
              (invoke-unit u prim-accumulator)))
          'dummy-rts-info))
```

Figure 6.11 : The defmac-compound-compiler Language Run-Time System

for units in the base and compound languages from sharing values that they could otherwise exchange. Instead, run-time units are cached upon invocation, and the defmac-compound-rts language compiler (figure 6.12) generates code that sequentially applies the cached run-time unit values to provide the primitives for the compound language unit.* The corresponding run-time system provides no primitives (since all the primitives come from the constituent units) so it contains no definitions and performs no interesting actions (figure 6.13).

*Note that this slightly alters the model of section 4.1: now the **unit/lang** mechanism must handle a list of units rather than a single unit.

```
(unit (import lookup-language)
      (export)

  (define *u (gensym "u:"))

  (define (compiler body imports exports)
    (values
     `((values
        (lambda (,*u)
         ,(let loop ([langs body])
            (if (null? langs)
                `,*u
                `((send (lookup-language ',(car langs))
                        rts)
                  ,(loop (cdr langs))))))
        'dummy-rts-info))
     #f
     imports
     exports
     '(())))

  (values compiler
          'dummy-compiler-info))
```

Figure 6.12 : The defmac-compound-rts Language Compiler

```
(unit (import lookup-language)
      (export)

  (values (lambda (u)
            (invoke-unit u))
          'dummy-rts-info))
```

Figure 6.13 : The defmac-compound-rts Language Run-Time System

6.5 Communication Channels

A traditional implementation of coroutines [23] uses two linguistic constructs: **coroutine**, for declaring a coroutine, and **resume**, for resuming execution across coroutines. The former enables programmers to specify the initial arguments and body of the coroutine. The resulting (uninvoked) coroutine is a first-class value. The latter expects its first argument to evaluate to a coroutine; it resumes this coroutine's computation with the second argument as a resumption parameter.

The implementation of **resume** is as follows. Each coroutine has a local suspension procedure that saves the coroutine's current continuation in a local state variable, then resumes the specified coroutine with the resumption parameter. The value of **resume** is mutated upon (re)entry into each coroutine to its local suspender. Since the definition of the suspender and its use are in different lexical scopes, the name **resume** must be made available through the global namespace.

There are three problems with this implementation strategy. First, it exposes the name **resume** to both inadvertent and malicious mutation by the user, who may not understand the implementation of coroutines—i.e., the channels are neither private nor secure. Second, it requires the user's language—which may have functions, say, but not continuations—to have the features necessary to implement the coroutine mechanism. Finally, it fails to work in a modular context, where coroutines can dwell in dinstict, separately-compilable units, because of the dependence on a global name.

The name **resume**, which is a run-time value used to communicate between compile-time syntactic forms, is known as a *channel*. The problem of how to use channels to describe powerful programming paradigms, while avoiding their baneful characteristics, has been unresolved for nearly two decades [personal communication: Bruce F. Duba, Matthias Felleisen, Daniel P. Friedman], but **unit/lang** can resolve this problem, which is similar to several others presented in this section, without difficulty.

6.6 Macro-Defining Macros

A macro use may sometimes expand into the *definition* of another macro. Should macro-generating macros be disallowed and, if not, how should they be processed? These are semantic decisions, so they must be entrusted to the language designer. Choosing to allow these and implementing the resulting expander is, in fact, quite easy in framework.

Consider the Scheme+def-syn language described in section 4.3. As described in that section, the language compiles into Scheme, which does not recognize macros; therefore, a macro-generated macro would not be handled correctly. To allow these, the compiler implementor needs to check the generated code and, if it contains any macros, specify Scheme+def-syn itself as the target language. This process would continue using Scheme+def-syn as the target so long as there are macro definitions in the source. Presumably, a pass will eventually fail to introduce any new macros. The resulting body is now a regular Scheme program, which is correctly fed to the Scheme compiler.

This example highlights several strengths of the **unit/lang** framework:

- Macro-generating macros, which are a powerful programming technique [15], are easy to support.

- The ability for a compiler to choose its target language based on each unit's content allows for great flexibility, and (as this example illustrates) permits the reuse of existing languages to solve complex problems.

- The process of expanding code that uses macro-generating macros does not need to lose crucial phase separation.

- The explicit recursion, in the form of choosing the current language, in the definition of Scheme+def-syn makes clear the potential for non-termination in the macro-generating macro process. It also highlights the condition necessary to terminate the recursion. The **unit/lang** framework makes this explicit, highlighting its value not only as a programming tool but also as a specification language. In contrast, such design decisions are usually hidden deep in the details of the implementation of a traditional macro expander, making it difficult to reason about such behavior.

6.7 Defining Complete Languages: The Lambda Calculus

Because compilers can rewrite the body of a unit arbitrarily, language implementors can define syntactic abstractions much more powerful than macros. For example, we can implement the language LC that compiles the lambda calculus (LC) into Scheme. It can either map LC procedures, variables and application to their Scheme counterparts, or it can use structures to represent procedures and provide interpretive support in the run-time system. Traditional macros cannot implement this transformation for two reasons. First, macros only extend the syntax of an existing language, whereas the LC compiler defines the entire syntax, thus disallowing non-LC terms. Second, most macro systems support only keyword-based expansion, whereas the LC compiler needs to examine all terms including juxtaposition (which is interpreted as application).[†]

Programmers who use LC would typically want LC procedures created in one LC unit to be usable when they are imported into another. The LC implementor ensures this by making sure each LC unit uses the same representation for procedures. To disallow cross-component usage, the implementor would link to a different representation—using, say, tags or generative datatypes to distinguish between them—each time the LC linker was invoked. For contrast, consider these two LC run-time system units:

(**unit/lang** Scheme (**import**) (**export**)

[†]This compiler could be expressed as an expansion-passing style elaborator [14].

```
  (define-struct lam (var body env))
  (lambda (u)
    (invoke-unit u make-lam lam-var
                   lam-body lam-env)))
```

This unit left creates the structure once (during registration) and supplies the same constructor and selectors to all LC units. In contrast,

```
(unit/lang Scheme (import) (export)
  (lambda (u)
    (define-struct lam (var body env))
    (invoke-unit u make-lam lam-var
                   lam-body lam-env)))
```

this unit creates a new (generative) structure for each application of the linker procedure. This is a *semantic* choice, so **unit/lang** rightly places it in the hands of the language implementor.

While LC is a toy example, the underlying choice is present in many real applications. One such application is MrEd [22], an extension to Scheme that defines a graphical, high-level operating system (OS). MrEd can be thought of as a language in this system. The compiler implements various language constructs both to control the capabilities and privileges of programs and for using the graphical libraries. The run-time system provides library support for their execution. Programs can communicate MrEd values (related to OS privileges and graphics) across MrEd units. To keep communication protocols simple and flexible, the MrEd run-time system uses the same representation for these values across all its programs. Because of this, MrEd cannot be thought of as a unit at the same level as MrEd applications. If each client program linked explicitly to the MrEd run-time system, then the run-time system would be instantiated once per program. Each instantiation would produce a different representation of its values, preventing MrEd units from communicating effectively. Instead, we must think of MrEd as existing before any individual MrEd unit is invoked, and supplying representations to each such unit. Representing MrEd as a language in this system clarifies this distinction and its implementation.

6.8 Interface Languages

The preceding discussion focuses on languages *within* components, but does not cover their interfaces. In fact, the interfaces between components are not always as simplistic as the examples suggest. Each language must take care to disallow the import of certain kinds of values, particularly ones that it cannot assign meaning to, or that can have harmful effects. (The former subsumes some instances of the latter.) An example of the former would be the import of continuations into a language without that control operator; one of the latter would be the import of a function that does not terminate into a language with only terminating computations. This is the problem of *interoperability*.

This dissertation presents only the mechanisms, not policies, for linguistic frameworks. There are few attempts to deal with the general problem of translating data between languages [52, 56], and these fail to address the most difficult problem, which is addressing how to perform, or reject, a data coercion request in a meaningful manner. From the perspective of **unit/lang**, this problem is easy to express. It is the question of what happens to data in the interstices between units, i.e., as it is conveyed during linkage in a compound unit. Since these must often be transformed, if only in their representations (e.g., using boxed or unboxed representations for numbers), these interstices must allow programmers to express regular program operations.

One way to address the problem of expressing these transformations would be to create a new construct, **compound-unit/lang**. A **compound-unit/lang** would be to a **unit/lang** what a **compound-unit** is to a **unit**: it would have the same syntactic structure, except augmented with an annotation that determines the language of the expression positions in the **compound-unit**. These expressions evaluate to the units participating in the linkage. The language can wrap these units in adapter units that perform the data conversion. The languages would therefore be *data conversion languages*, and an implementation can provide a stable of such languages for common pairs of languages.

6.9 Datatypes

Suppose a programmer wants to design a new definition language for describing dataypes.[‡]) A program that uses this language might look like

(**define-datatype** *Exp*
  [*num* (*n number?*)]
  [*plus* (*le Exp?*) (*re Exp?*)])

The **define-datatype** construct introduces new structures, whose selectors and other attributes must be hidden from the user. Programmers use these datatypes through the **cases** construct, whose first position specifies a datatype (such as *Exp*), followed by a sequence of branches consisting of a pattern-match and expression pair.

The programmer can design a language to implement *datatype* by writing two units:

(**unit/lang** DDCompilerDef
    (**import** ...) (**export** ...)
  (**define-datatype** *Exp* ...))

(**unit/lang** DDRTSDef
    (**import** ...) (**export** ...)
  (**define-datatype** *Exp* ...))

where the DDCompilerDef and DDRTSDef languages generate the appropriate compiler and run-time structure definitions. The bodies of the two units are identical. These units form the compiler and run-time system for a new language, Scheme+Exp.

[‡]This example is inspired by the construct in the second edition of *Essentials of Programming Languages* [24].

The DDCompilerDef language has a compiler and run-time system. Its run-time system does not provide any interesting primitives. Its compiler uses Scheme+def-syn as its target language, and turns the above unit into the following Scheme+def-syn unit:

```
(unit/lang Scheme+def-syn
    (import ...) (export ...)
  (define-syntax cases ()
   [(cases expr clause ...)
    (let ([v expr])
      ;; ... ⟨expand each clause to test and dissect v⟩ ...)])
  (declare-primitives (Exp? num num-n num? plus plus-le plus-re plus?)))
```

The DDRTSDef language also has a compiler and run-time system. Its compiler turns the datatype specification into a Scheme unit:

```
(unit/lang Scheme
    (import ...) (export ...)
  (define-struct Exp ())
  (define-struct (num struct:Exp) (n))
  ...
  (define (mk-num n)
    (if (number? n)
        (make-num n)
        (error ...)))
  ...
  (lambda (u)
    (invoke-unit u Exp? mk-num num-n num? ...)))
```

The corresponding run-time system is also uninteresting, since these are primarily syntactic transformations.

Finally, we're ready to use this apparatus. The user writes

(**unit/lang Scheme+Exp**
    (**import** ...) (**export** ...)
  (**cases** *Exp v*
    [*num* (*n*) ...]
    [*plus* (*le re*) ...]))

The compiler translates this into a **let** and **cond** with references to the variant predicates and selectors. The interface to these is specified by the *declare-primitives*. The interface is satisfied by the closure returned by the run-time system. Therefore, execution proceeds apace.

6.10 Signatures

While units provide extremely powerful abstraction facilities, it can be laborious to repeat the names of imports and exports, especially since the exports of one unit often correspond to the imports of another. This problem becomes intractable in large libraries such as MrEd [22], which has hundreds of exports. To combat the problem, MzScheme [20] offers a layer atop units called *signatures*. A signature is a bundle of names that can be used in imports and exports of special units called **unit/sig**, indicating that they operate on signatures rather than individual names.

Since signatures are simply a syntactic abstraction over an existing mechanism, it should be possible to define them using **unit/lang**. Unfortunately, they cannot be defined using the model described in section 4.1. The problem is that the use of signatures inherently alters the import and export lists of a unit. In contrast, the type signature given for compilers,

$$Expr_L\ list \times Id\ list \times Id\ list \longrightarrow$$

$$LanguageName \times Expr_{Target}\ list\ \times Id\ list \times Id\ list \times Id\ list\,,$$

(implicitly) specifies that they leave the lengths of the import and export lists unmolested; they are only permitted to rename these identifiers.

In principle, compilers should be allowed to not only rename but also alter the signatures of units, for maximal expressive power. Modifying the implementation to permit this is easy. This extension now permits the description of language features such as signatures. Linking signed units requires signature elaboration; this is easily done by using **compound-unit/lang**. For instance, the link clause for an interpreter might look like

(**compound-unit/lang**
  (**import**)
  (**link**
    [RTS ExpRTSImplLink ($\langle an$ ExpRTSImpl unit$\rangle$)]
    [ENV EnvImplLink ($\langle an$ EnvImpl unit$\rangle$)])
    [INT ExpRTSLink ($\langle an$ ExpUse unit$\rangle$ RTS ENV)]
  (**export**))

where ExpRTSLink, EnvImplLink and ExpRTSImplLink are languages that handle the expansion of signatures for linking clauses.

Chapter 7

Preserving Linguistic Abstractions

ROS: What are you playing at?

GUIL: Words, words. They're all we have to go on.

—Tom Stoppard, *Rosencrantz and Guildenstern are Dead* [53]

Programming languages are accompanied by a host of language-processing tools, such as compilers, static analyzers, debuggers, and so on. By translating new languages into existing ones, we have shown how to reuse existing language implementation technology, including compilers and run-time systems. We must consider the problems that arise when we reuse these tools.

A language designer who defines a new language in terms of compilation into an existing language would typically prefer to reuse these existing tools for his new language. At the same time, programmers in the new language would expect feedback to be in terms of their source, not just in terms of the generated code. This is essential to maintain the abstraction boundary, i.e., the impression that the programmer is coding in the new language, not in the destination language of the compiler. Understanding feedback in terms of the destination language of compilation can be especially difficult if the compilation process is complicated and introduces a lot of code that does not correspond directly to the source program. Without remedy, this problem would considerably hamper and inhibit the use of linguistic abstraction.

This section presents two approaches to addressing these problems. Consider the program in figure 1.3. Figure 1.4 shows the same program (translated into a parenthesized version of Java), except it makes the Adapter explicit by using an **Adapter** construct, which is translated into the equivalent of the code in figure 1.3.

The use of an Adapter makes it clear that there is a mismatch between *OutputRoutines* and *VendorGraphics*, and highlights how this can be overcome. The equivalent of *Client*'s Java declaration is then

(**class** *Client*
  (**fields** (*VendorGraphics var* = (**new** *GraphicalOutput* ())))))

This results in the error message

```
test.cj:15.33-15.56: Incompatible assignment type. Can't convert
  GraphicalOutput to VendorGraphics.
GraphicalOutput created by Adapter at test.cj:7.2-7.8.
```

which uses the Adapter declaration to report the error in terms of the programmer's pattern code, not just the expanded constructs. The error message frees the programmer from having to track manually what each construct can introduce, which can be especially difficult since the expansion of the construct can change over time. This error message is produced by a generic type-checker for the parenthesized version of Java; section 7 describes how this information is generated in greater detail.

The problem in Scheme is exacerbated by the relatively large size of the language compared to the size of the core, the difference being made up by syntactic abstractions. The experience is particularly dramatic in DrScheme [16], where a program in the beginner's language level routinely expands into programs at advanced levels—yet beginners must be shielded from this complexity. For instance, even the following very simple beginner program

(**define** (*f x*)
  (**cond**
   [(= *x* 0) 1]
   [**else** (∗ *x* (*f* (− *x* 1)))]))

expands into a program with several features (**define-values**, **case-lambda**, **if**) not taught to beginners:

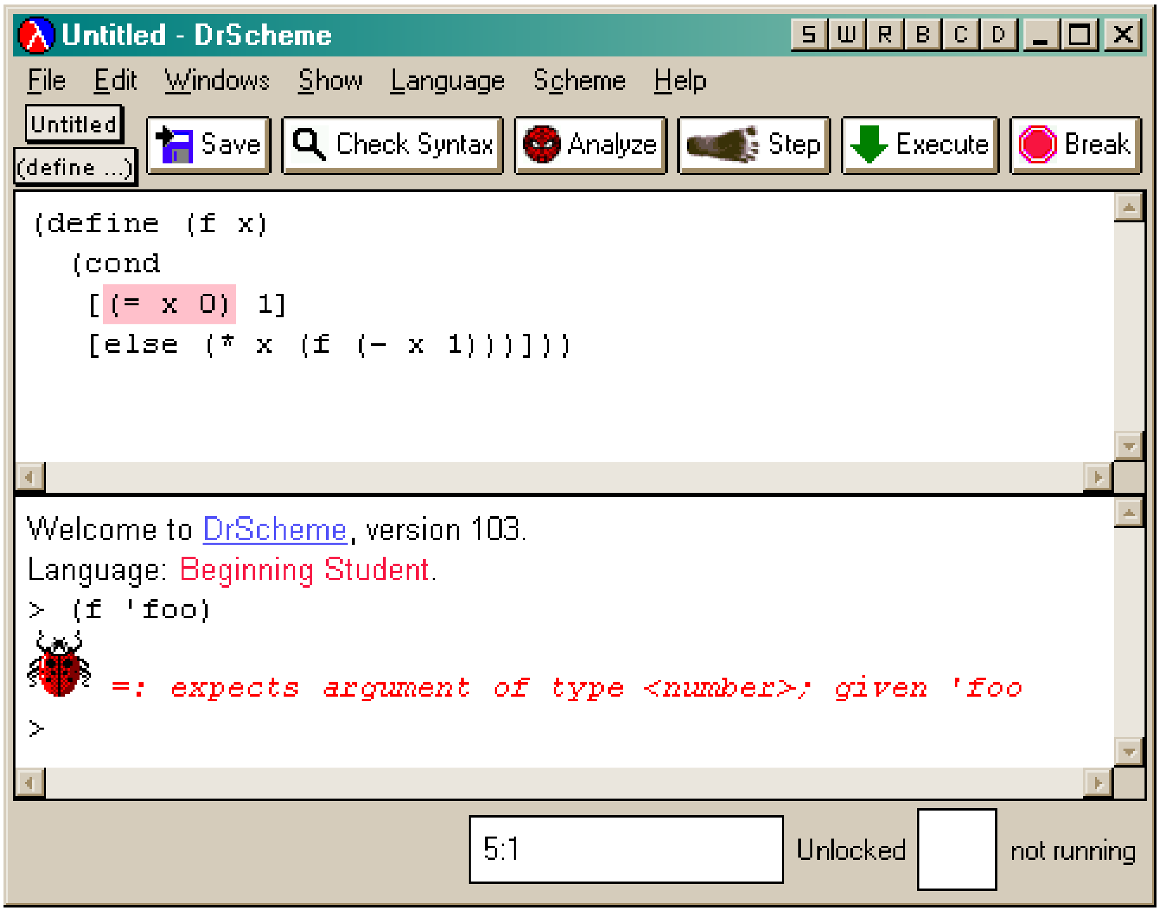

Figure 7.1 : Reporting Errors

```
(define-values
  (f)
  (case-lambda
    [(g643:x)
     (if (= g643:x (quote 0))
       (quote 1)
       (* g643:x (f (- g643:x (quote 1)))))]))
```

(the mangled names arise from hygienic expansion [33]).

Since we expect linguistic reuse to encompass tool reuse, our framework must provide a suitable interface for tools that wish to communicate with the programmer. The interface consists of two kinds of information, and protocols that process this information. The two kinds of information are:

source-correlation For each term, we must note the source location of the phrase that generated it. This information is maintained through the compilation process. Thus when a tool needs to provide feedback about a term, it can report it in terms of the source phrase that the programmer needs to examine. This is especially helpful in conjunction with graphical interfaces, since they can highlight the appropriate source text.

elaboration-tracking We must also maintain a history of transformations applied to each elaborated term. In complex examples, a destination language term might arise from the repeated compilation of a source term. A programmer must therefore sometimes be given an explanation of the series of transformations that led to an error. This is especially useful to the developer of the transformations.

By providing an appropriate interface, the tools themselves do not need to process this information. We prototyped an interface of this flavor for a previous system [36]; figure 7.1 shows the effect of these features on error reporting in actual user programs. We wish to define a similar facility for **unit/lang**.

Chapter 8

Related Research

Man invented language to satisfy his deep need to complain.

—Lily Tomlin

Components are typically defined as separately compilable, externally connected, independently deployable and multiply instantiable units of black-box reuse [54]. McIlroy [41] introduced the concept of a factory of software components in 1968. Components are gaining prominence with the growing popularity of protocols such as COM, CORBA and JavaBeans, all of which provide some form of component-based reuse. Proponents of this technology envision each component being written in the language best suited for its task, with these protocols providing ways to exchange data between the components. In reality, the protocols tend to be extremely low-level, and reflect the data languages of specific low-level computation models. Still, components promise to grow in importance as they mature. It is therefore essential to study their interaction with syntactic abstractions, a powerful, traditional notion of abstraction.

Szyperski [54] offers a thorough review of standard component protocols and identifies technical criteria that characterize components and suggest reasons for their popularity. He also emphasizes the "vital importance" (page 162) of data safety for components that is one of the motivations for this work, since we enable each language to define its own notion of safety (which may be at a higher level than that of the underlying virtual machine).

Some of these macro systems support so-called "referential transparency" [10]. A macro is said to be referentially transparent when free references in a macro refer to variables in the scope of the macro's definition, not its use. Referential transparency is an attempt to fix the meaning of expansions by, for example, guarding against re-binding of names between the definition and use of a macro. However, it cannot prevent against mutations to the environment before the macro was defined. It is also unclear how to extend referential transparency to systems where the meaning of every part of the syntax, not just the primitive names, can be altered.

Other authors have considered the interaction of macros and modules (which are often not as general as components, because they hard-wire their import dependencies on other modules and thus cannot be deployed independently). Blume [6] shows that traditional tag-based implementations of hygiene do not function properly in the presence of modules, especially when they export by name, and shows how to modify hygiene algorithms to avoid these problems. Curtis and Rauen [11], Rees [45] and Shalit [46] all have module systems that allow modules to export macros. The free names in these macros refer to bindings in the module where the macro was defined, even if that module does not export those bindings. These macros are meaningless in a component-based world, since no component can implicitly expect the presence of any other specific component—in particular, the one that defined the values referred to by the macro—at run-time. Waddell and Dybvig [57] have a similar system except that they require top-level modules to make all potentially implicit exports explicit, but this still does not address the independence of components. Queinnec and Padget [44] describe a more flexible system, but it depends heavily on first-order, named packages rather than components. Davis, et al [13] separate the specification of macros and the values needed by the macros, but this choice is made by the client and is also deeply tied to a first-order package system. Also, in contrast to these systems, **unit/lang** units can be treated as first-class values (as they are in MzScheme), and can be instantiated multiple times. Bawden [2] presents an implementation of "first-class" macros for Scheme. His terminology, however, is misleading. The macros are not first-class; their enclosing lexical environment is the first-class entity. Furthermore, his modules are not components, but namespaces with just three operations: one for defining the interface to the namespace, one for defining a namespace, and one for opening the namespace. There are no constructs for specifying imports and exports for modules or for linking modules to form larger modules. His effort thus fails to address the core issues of component-oriented programming [54].

This work can exploit popular specification methods such as macros, templates and parser generators to simplify the description of compilers. There is a large corpus on such specifications [1, 3, 8, 12, 10, 14, 15, 29, 35, 37, 39, 40, 51, 55, 60]. Many of these works ignore the interaction with components; indeed, many presume textual substitution on a per-file basis in the tradition of `cpp`, paying no heed to modular boundaries. These systems do offer different combinations of pattern matching, hygienic macro expansion, and type systems to validate the code expansions and the type-correctness of expanded code. The approach might appear complex compared to these, but as figure 4.3 and figure 4.4 show, defining a language in terms of macros and the procedures they depend on is straightforward with the right abstractions. A programming environment can mask most of the differences in appearance between this approach and a traditional macro system by generating part of the unit wrapper.

This work can also be seen as an attempt to add linguistic support to high-level operating systems [22]. Other projects have attempted to create high-level operating systems. These have centered primarily on creating safe single address spaces [5, 27, 59]. None of these directly address the need to create additional layers of linguistic abstraction atop these kernels. Similarly, these systems do not provide a mechanism for integrating multiple (safe) programming languages into the same OS. None of these efforts even identify the criteria that such an integration must obey. We provide a model that can be adapted to many of these systems.

Chapter 9

Limitations and Future Work

If you can't understand it, you certainly can't trust it.

—Doug McIlroy

This presentation discusses work conducted in the context of dynamically-typed languages. While all the results here are *typable*, they do not exploit the prescriptive uses of types. It is possible that type declarations can enhance the utility of these results, especially for multi-stage programming [55].

In the model presented in this dissertation, the toolkit that accompanies a model is rather sparse. In particular, it is limited to the evaluator alone. Realistic systems will have a wide variety of tools for static and dynamic use. I believe it may be fruitful to consider the use of simple specification mechanisms, such as set-based analysis [18, 28], to construct domain-specific type-checkers and program verifiers.

The model herein is just that: a *model*. As languages grow in number, it becomes difficult to create, use and administer them. It may be useful, to make the model more accessible to unsophisticated programmers, to provide interface aids, such as a graphical interface, that simplify the specification of languages, especially in simple cases such as the construction of macro libraries. Specifically, we must improve the granularity of language definitions. While individual programmers can currently reuse the components used to define the compilers and run-time systems of languages, we provide no institutional policy for doing this. We would instead like to be able to define language *fragments*, and combine these to create whole languages (as we currently do for macro languages). This would, for instance, permit a programmer to combine several extensions to a common base language into one language, and use different combinations of these extensions for different components in the program, and serve as the natural extension of vocabularies.

Chapter 10

Contributions and Conclusion

> PLAYER: We keep to our usual stuff, more or less, only inside out. We do on stage things that are supposed to happen off. Which is a kind of integrity, if you look on every exit as being an entrance somewhere else.
>
> —Tom Stoppard, *Rosencrantz and Guildenstern are Dead* [53]

Components and generative programming are both powerful software construction techniques. With components, programmers can increase the reuse potential of code and can create applications by gluing together components. Software generators effectively define domain-specific or "little" languages and map them into existing languages, thereby promoting the reuse of language technologies. Their marriage is especially potent, since it allows programmers to incorporate modules written in these domain-specific languages into larger applications.

My work represents the first attempt to consider the semantic needs of the output of generative programming. In the process, it unifies linguistic extensions via both macros and libraries, and shows how the two can interact. It lays out a mechanism that is general enough to encompass several existing generative programming techniques. This work also maps some of these techniques, most prominently macros and templates, into this framework.

This work has also addressed several concrete technical problems that were previously partially or wholly unresolved. This work offers the first explanation of how to implement macro-generating macros while preserving phase-separation. It extends this solution to creating towers of languages closed over their semantics. Finally, it demonstrates the use of the run-time system to transparently communicate between compile-time entities.

The structure of the solution presented in this document has the elegant property of permitting *towers of languages*. As the examples have shown, the definition of a language may use the definition of other languages, which in turn depend on language definitions. As a result, programmers can build arbitrarily deep layers of abstractions, just as they currently do with module systems. This is a unique feature of this work not shared by prior work on linguistic abstractions.

In addition, the implementation associated with this dissertation has formed one of the key bases of DrScheme [16], which is used at hundreds of institutions worldwide for both secondary school and university education and research projects. Many of the benefits that distinguish DrScheme accrue from the features of this implementation. The same implementation has been reusable enough to construct toolkits for related languages such as XML [7].

I believe this work has potential for widespread application. As the barriers between programming languages and operating systems dissolve [22], programmers increasingly confront the question of what a programming language is in this new context. The work in this dissertation answers this question, and shows how programmers can create specialized languages with well-defined semantics that reflect the constraints of various domains.

This work has not addressed the practical implications of using a large number of languages that interact intimately. We do not have experience building such systems, nor have we addressed all the needs of daily use. The average user might find the solution described in this work unwieldy and overly general. Designing a system that is pragmatic while preserving the essence of this work is left for future work. Finally, we have also neglected the demands that typed languages impose, and the benefits that they engender. We believe these and other extensions to the results of this dissertation are necessary and would be fruitful.

Bibliography

[1] Aitken, W., B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter and C. Simonyi. Transformation in intentional programming. Microsoft Research white paper, September 1997.

[2] Bawden, A. First-class macros have types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.

[3] Bawden, A. and J. Rees. Syntactic closures. In *ACM Symposium on Lisp and Functional Programming*, pages 86–95, 1988.

[4] Bentley, J. L. *More Programming Pearls*. Addison-Wesley, 1988.

[5] Bershad, B., S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Symposium on Operating System Principles*, pages 267–284, 1995.

[6] Blume, M. Refining hygienic macros for modules and separate compilation. Technical Report TR-H-171, ATR Human Information Processing Research Laboratories, 1995.

[7] Bray, T., J. Paoli and C. Sperberg-McQueen. Extensible markup language XML. Technical report, World Wide Web Consortium, Feburary 1998. Version 1.0.

[8] Cardelli, L., F. Matthes and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital SRC, 1994.

[9] Cartwright, R. S. and M. Felleisen. Extensible denotational language specifications. In Hagiya, M. and J. C. Mitchell, editors, *Symposium on Theoretical Aspects of Computer Software*, pages 244–272. Springer-Verlag, April 1994. LNCS 789.

[10] Clinger, W. and J. Rees. Macros that work. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[11] Curtis, P. and J. Rauen. A module system for Scheme. In *ACM Symposium on Lisp and Functional Programming*, pages 13–20, June 1990.

[12] Czarnecki, K. and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 1999.

[13] Davis, H., P. Parquier and N. Séniak. Talking about modules and delivery. In *ACM Symposium on Lisp and Functional Programming*, pages 113–120, 1994.

[14] Dybvig, R. K., D. P. Friedman and C. T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, January 1988.

[15] Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[16] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001. To appear.

[17] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, number 1292 in Lecture Notes in Computer Science, pages 369–388, 1997.

[18] Flanagan, C. and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):369–415, 1999.

[19] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[20] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[21] Flatt, M. and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[22] Flatt, M., R. B. Findler, S. Krishnamurthi and M. Felleisen. Programming languages as operating systems (*or*, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, September 1999.

[23] Friedman, D. P., M. Wand and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[24] Friedman, D. P., M. Wand and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, second edition, 2001.

[25] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.

[26] Graham, P. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[27] Hawblitzel, C., C.-C. Chang, G. Czajkowski, D. Hu and T. von Eicken. Implementing multiple protection domains in java. In *USENIX Annual Technical Conference*, 1998.

[28] Heintze, N. Set-based analysis of ml programs. In *ACM SIGPLAN International Conference on Functional Programming*, 1994.

[29] Johnson, S. C. YACC — yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1975.

[30] Jones, M. P. and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, New Haven, Connecticut, December 1993.

[31] Kelsey, R., W. Clinger and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.

[32] Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[33] Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[34] Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[35] Kohlbecker Jr, E. E. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[36] Krishnamurthi, S., Y.-D. Erlich and M. Felleisen. Expressing structural properties as language constructs. In *European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 258–272, March 1999.

[37] Krishnamurthi, S., M. Felleisen and B. F. Duba. From macros to reusable generative programming. In *International Symposium on Generative and Component-*

Based Software Engineering, number 1799 in Lecture Notes in Computer Science, pages 105–120, September 1999.

[38] Krishnamurthi, S., K. E. Gray and P. T. Graunke. Transformation-by-example for XML. In *Practical Aspects of Declarative Languages*, January 2000.

[39] Lipkin, D., J. Marsh, H. Thompson, N. Walsh and S.Zilles. XSL Transformations Version 1.0. World Wide Web Consortium Working Draft, July 1999.

[40] Mauny, M. and D. de Rauglaudre. Parsers in ML. In *ACM Symposium on Lisp and Functional Programming*, pages 76–85, 1992.

[41] McIlroy, M. D. Mass produced software components. In Naur, P. and B. Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138–150, October 1968.

[42] Moggi, E. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[43] Perlis, A. J. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13, September 1982.

[44] Queinnec, C. and J. Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, October 1991.

[45] Rees, J. Another module system for Scheme. Scheme 48 documentation, 1 1994.

[46] Shalit, A. *The Dylan Reference Manual*. Addison Wesley Longman, 1996.

[47] Sheard, T. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.

[48] Smaragdakis, Y. and D. Batory. Scoping constructs for program generators. Technical Report 96-37, Department of Computer Sciences, University of Texas at Austin, December 1997.

[49] Smaragdakis, Y. and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.

[50] Stallman, R. M. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, 1993.

[51] Steele, G. L., Jr., editor. *Common Lisp: the Language.* Digital Press, Bedford, MA, second edition, 1990.

[52] Steele, G. L., Jr. Building interpreters by composing monads. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, January 1994.

[53] Stoppard, T. *Rosencrantz and Guildenstern are Dead.* Grove Press, New York, NY, 1967.

[54] Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[55] Taha, W. and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, 1997.

[56] Trifonov, V. and Z. Shao. Safe and principled language interoperation. In *European Symposium on Programming*, 1999.

[57] Waddell, O. and R. K. Dybvig. Extending the scope of syntactic abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1999.

[58] Wadler, P. The essence of functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[59] Wahbe, R., S. Lucco, T. E. Anderson and S. L. Graham. Efficient software-based fault isolation. In *Symposium on Operating System Principles*, pages 203–216, December 1993.

[60] Weise, D. and R. Crew. Programmable syntax macros. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.