An Integrated Lexically-Scoped "Macro" Facility for Compiler Extensibility

—including—

An Extensibility Manifesto

FAST AND LOOSE PROPAGANDISTIC PARTIAL DRAFT

—

NOT FOR GENERAL DISTRIBUTION

—

COMMENTS WELCOME

Paul R. Wilson, Donovan Kolbly, and Stephen Carl
Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
wilson@cs.utexas.edu
donovan@tkg.com
esteban@cs.utexas.edu

May 5, 1996

Abstract

Syntactic extensibility is achieved in many languages through the use of *macroexpansion*. Macro facilities allow syntactic constructs to be implemented by simple source-to-source transformations, effectively allowing programmers to "reprogram the compiler" from within the programming language. This facility is useful for implementing programming languages, and for extending them in novel, often application-specific ways. Advanced macro facilities provide even greater power, allowing many analyses and optimizations to be implemented outside the compiler, and allowing layered modularity in overall system implementation.

Unfortunately, conventional macroexpansion facilities have several weakness. Most of them have grave weaknesses with respect to scope, making them extremely error-prone and difficult to use correctly for nontrivial purposes. Recent "hygienic macroexpansion" facilities solve the basic problems of scope resolution for many purposes, but are weak in terms of implementing novel scoping mechanisms. Due to lack of integration with compilers, they are also weak with respect to supporting novel analyses and optimizations in source-to-source transformations.

In this paper, we present a novel mechanism for syntax extension and scope resolution. By integrating our mechanism into the normal scoping mechanisms of the compiler, we provide a clean, unified solution. Perhaps more important, this integration provides new power to macro writers, allowing them to exploit and enhance the normal workings of the compiler to cleanly and efficiently implement language extensions and/or optimizations.

Despite the oft-noted "tower of Babel" potential of highly extensible programming languages, we argue that this powerful kind of extensibility is necessary

precisely to *avoid* a proliferation of incompatible and badly-designed languages. We argue that the normal demarcation between programming languages occurs *at the wrong level of abstraction* and cannot be maintained in many situations—in fact it *isn't* maintained well in practice, despite the conventional conceptual boundaries between languages, libraries, and application programs.

We attempt to present a different vision of the proper role of programming languages in the overall scheme of software development. Much of our philosophy is similar to that of researchers in reflection and metalevel programming, but not widely appreciated by language designers or application developers. We attempt to make this philosophy intelligible to a wider audience, as well as extending it in useful directions.

Contents

1 Introduction

Many languages provide syntactic extensibility in the form of macroexpansion facilities; programmers may write extensions to the language as syntactic transformations from new constructs to constructs understood by the underlying compiler. Unfortunately, most macroexpansion facilities have severe limitations and are very error-prone due to scoping problems. More recent *hygienic* macroexpansion systems eliminate scoping problems by taking special actions to preserve lexical scope by automatic variable renaming.

In this paper, we present a new scheme for lexically-scoped syntactic extension, called *syntactic exposures*. Our goals for syntactic exposures are similar to those of hygienic macroexpansion, but more ambitious. Where hygienic macroexpansion is implemented as a prepass, and generally compiles to a standard base language, our syntactic extension facility is integrated into the front end of the compiler itself, as a relatively straightfoward refinement of the normal scoping mechanisms of the compiler.

This approach sacrifices some portability, in that the compiler itself is modified slightly to handle syntactic extensibility. This lack of portability could be avoided by implementing our facility entirely as a traditional prepass, and this may be desirable in some situations—our mechanism is clearer and more intuitive than traditional hygienic macroexpansion, and the resulting "precompiler" would provide many advantages.

We have chosen to integrate our scoping mechanism into the compiler, however, for two reasons. The first is that much of the work done by a good syntactic extension facility is *the same work* done by the scoping mechanisms of the compiler—unifying the two mechanisms simplifies and clarifies the structure of the system. This may have advantages for implementing other aspects of the system. (For example, source-level debuggers must be able to map code transformations backward; having a single transformation mechanism may simplify this, compared to having multiple kinds of transformations which must be decomposed and handled separately.)

The second reason for an integrated approach is that it allows novel forms of compiler extensibility; we can implement a metalevel programming facility that allows the compiler to be extended cleanly from within the language for many purposes. Our approach is similar to those used by *metaobject protocols* [KdRB91] and *open implementations* of compilers, but allows novel forms of *reflection on scope*, which other systems do not. This allows the efficient implementation of scoping mechanisms and various analyses and optimizations outside the compiler proper. Looked at another way, our compiler can provide hooks that allow the language to be extended or partially reimplemented without detailed knowledge of compiler internals.

Despite the extreme power of this approach, for normal uses it provides a high-level facility to the programmer, which subsumes normal macros and hygienic macroexpansion; naive users need not deal with the potential complexity of an "open" compiler. Sophisticated users who need more power can be presented with a layered set of abstractions of the compilation process, allowing them to perform progressively more sophisticated tasks with a smooth learning curve.

While this work has been done in the context of an extended dialect of the Scheme language, our eventual goal is to produce a compiler for a highly extensible multi-level systems programming language, which can be bootstrapped from approximately the level of C to encompass features of higher-level languages such as Lisp, Smalltalk, ML, C++, or P2.[1] It should also be extensible in application-specific ways, to include features such as sophisticated declaration processing and optimization; this should cleanly support features such as constraint checking and query optimization for novel sublanguages.

Currently, we benefit from Scheme's simple and regular surface syntax. In the long term, however, we intend to develop a language with a syntax that is more conventional, and more familiar to most programmers. We believe that our integrated approach to syntactic processing will make this easier than if we used a traditional macroexpansion prepass. When syntactic extensibility is combined with nontrivial syntax, the usual separation between parsing and further processing becomes hard to maintain. We expect the integrated approach to lend itself particularly well to dealing with the context-sensitivities of processing a more conventional surface syntax in the presence of syntax extensions.

[1] P2 is a powerful language for defining composable software abstractions, with support for "nontraditional parameterized types" which can be composed in several ways at several levels of abstraction [?].

Goals of the Paper. This paper is intended to address several audiences with different expertise and different degrees of interest; some readers may wish to skip or skim much of it. By and large, the material presented here should be intelligible to anyone with a basic undergraduate education in computer science. In a few places, it will be helpful to have had a course in compilers, but the main points should be understandable without even that.

For the general reader literate only in basic computer science, we want to provide an introduction to extensible programming languages, the best example being Lisp. Lisp has strengths that have generally gone unappreciated by the implementors and users of other languages. We believe that this is partly due to Lisp's unconventional syntax and an unfortunate split between the Lisp community and the mainstream programming language community. We intend to show that Lisp's syntactic extensibility mechanisms, though flawed, are extremely valuable; their problems can be fixed, and the results are applicable to a wide variety of languages.

Lisp programmers may find our discussion of Lisp's strengths and weaknesses. Many Lisp programmers use macros in the wrong ways, and fail to recognize what they're most appropriate for. The weaknesses of Lisp macros discourage sophisticated uses, because they are unnecessarily error-prone; the strengths are often not recognized for what they are—a transformational grammar facility for compiler customization.

For programming language researchers and developers, we intend to show that recent innovations in syntax extension mechanisms (colloquially, "macros") have made syntactic extensibility much more attractive. A powerful macro system can support the development of sophisticated languages and compilers in a layered way.

For researchers in extensible languages, we present our own approach to lexically-scoped syntax extension, which we believe is cleaner and more intuitive than hygienic macroexpansion. Important aspects of the compilation model can be exposed in a comprehensible way to programmers who are not compiler experts, allowing them to extend or optimize programming languages conveniently and relatively safely.

Structure of the Paper. The remainder of the paper is structured as follows.

Section 2 discusses two basic uses for syntactic extension mechanisms: implementing programming languages, and extending programming languages in domain- or application-specific ways.

Appendix A elaborates themes from Section 2, further motivating the development of syntactic extensibility, by examining current problems in software development. We argue that the current proliferation of incompatible languages and awkward application programs is due in part to the lack of appropriately extensible languages. We also argue that many problems not widely recognized as "language" problems are actually due to a lack of a good unified and extensible linguistic framework.

Section 3 lists desirable features for extensible languages. **[blah blah]**

Section 4 describes several common kinds of macros, particularly with respect to their scoping implications, motivating the mechanisms we develop in this paper.

Section 5 discusses traditional mechanisms commonly used in compilers for Scheme and other block-structured languages. We also discuss a common idiom for implementing pass-by-name (deferred expression evaluation) without violating lexical scope. These runtime mechanisms are the inspiration for our extended compile-time mechanism for syntactic extensibility.

Section 7 presents a simplified version of our new mechanism, using an interpretive execution model. A succession of refinements shows how progressively more sophisticated scoping issues can be addressed. This corresponds to an interpretive prototype we have built for a simple interpreter for a subset of Scheme.

Section 8 shows how the interpretive solution can be transformed straightforwardly into a compiled version, where all of the syntactic processing is done at compile time. This closely corresponds to the mechanism we have implemented and used in the RScheme compiler.

Section 10 discusses grammatical aspects of our system, and compares it to traditional parsing and compilation techniques. We explain that while Lisp-like languages are often considered syntactically crude, their surface simplicity is misleading. Lisp and Scheme have a two-level, *extensible*, *transformational* grammars; they have simple phrase-structure grammar (surface syntax) and a higher-level user-extensible transformational (macro) component. Such grammars are not context free, yet they can still be parsed straightforwardly, in a natural top-down manner.

While conventional Lisp macros allow users to specify transformations in terms of surface syntax, this is the source of many errors, because the programmer's *intent* is usually to write transformations over the *abstract* syntax. Lexically-scoped macro facilities essentially infer the programmer's semantic intent, providing an elegant and straightforward means of writing a constrained but very useful set of abstract syntax transformations. We sketch a straightforward extension of this approach to a transformational *attribute* grammar; this essentially turns the compiler into a compiler toolkit that is exposed in a convenient way within the language.

The use of two-level non-context-free grammars raises interesting issues in parsing, especially if a more conventional surface syntax (like Algol or C's) is desired.

Section 11 gives examples of potential uses of our mechanism. Beginning with simple examples that rely primarily on the straightforward uses of the scoping mechanism, we progress through more difficult examples requiring increasing compiler support and knowledge of compiler principles.

Section 12 sketches an extension of our ideas to the creation of simple but radically extensible language and compiler, called an *exocompiler*. In this scheme, most of the work done by a normal compiler is done *outside* the compiler, in a layered and extensible set of libraries written in the programming language itself.

Section 14 briefly surveys related work.

Section 15 summarizes the main points of the paper and suggests avenues for future research.

2 Motivation (Part I)—Uses of Syntactic Extensions

A variety of languages use macro preprocessors or precompilers to implement language features by *source-to-source transformation*. The idea of source-to-source transformation is that programmers can define new, extended, or optimized versions of constructs—including data type definition facilities and control constructs—in terms of preexisting constructs understood by a "base" compiler for the unextended language.

There are typically two kinds of users of such a system: developers of languages language implementations, and normal programmers who need to extend or optimize a language for particular purposes. These kinds of users have different needs, but a single well-organized mechanism may serve them both well. In fact, the differences between these two kinds of users are often overestimated, however, as we will argue below, and in more detail in Appendix A.

2.1 Implementing Languages by Transformation

It is common in many programming language communities—particularly the Lisp and functional programming communities—to implement languages largely by source-to-source transformation. A language may be defined in (at least) two levels: primitive constructs and higher-level constructs that may be expressed in terms of the primitives. A compiler can automatically simplify the high-level constructs into combinations of primitive constructs; these rewritten expressions "show their bones" to the compiler, making it simpler for the compiler to perform analyses and optimizations.

Writing source-to-source transformations is often harder than it should be, however, because source forms often *don't* show their bones; in particular, subtle scoping issues are tricky to resolve correctly, because the source language does not directly reveal scoping issues to the compiler. The writer of a source-to-source transformation must be very careful to infer the *meaning* of expressions, which may not be obvious from the "text" of the program due to context-sensitivities. Typically this is done by *renaming* variables to disambiguate variable names. Unfortunately, this kind of renaming transformation has two major disadvantages:

- *It is lossy.* In some cases, the names of variables are significant, and if an expression has its variables renamed *too early*, crucial information is lost or obscured. The underlying assumption behind renaming is that there is a simple *canonical* base language that user code can be reduced to, and that all significant analyses and optimizations can be performed most easily on this canonical representation. This assumption is sometimes false—particularly where source-to-source transformations are being used to implement scoping facilities or to optimize expressions based on knowledge of scope. The quick reduction of scope information reduces implementors' freedom to change the scoping facilities of the language. For example, the loss of explicit representation of scopes makes it difficult to cleanly implement an efficient object system with encapsulated objects, whose methods have privileged access to the private fields of those objects.
- *It makes transformed code difficult to understand.* In practical terms, one of the major advantages of source-to-source transformations is that they are easy to understand; a reasonably sophisticated user of the language can understand the transformations into the canonical base language, because that language is a subset of the language the user is very familiar with.[2] However, transformation into a language that is *too* low-level, *too soon* obscures issues, rather than clarifying them.

These issues are familiar to developers of compilers for Lisp and functional languages. Rather than relying on explicit source-to-source transformation inside the compiler itself, compiler writers often use the same easy-to-use high-level syntactic extension facilities that are presented to normal programmers. In many cases, the high-level facilities are quite capable of performing the needed transformations and generating excellent "intermediate" code in the base language understood by the compiler. Most such "high-level" extension facilities are not *high-level enough*, however, even for most uses that compiler writers put them to. They are error-prone because they often deal with transformations *at the wrong step* in the overall compilation process, as will be explained later.

What compiler developers need is a high-level syntactic transformation facility that gracefully extends to allow lower-level analyses and optimizations—rather than the traditional separation between language-level macros and compiler-internal source-to-source transformations.

We believe an integrated approach is necessary to achieve this, so that analyses and optimizations can be performed at the appropriate times during the overall compilation process.

2.2 Using Syntax Extension in Normal Programming

Application programmers in many languages are familiar with the uses of macroexpanders and preprocessors of various kinds. The development of a large

[2] For example, the standard conversion of a Scheme `let` expression—into a procedure call with an anonymous lambda expression—is fairly comprehensible to Scheme programmers, even if it is rather ugly, and obviously somewhat harder to read than the original `let` expression.

application is often best viewed as a two-level process of developing a domain-specific programming language, and developing applications within that language. Most commonly, this takes the form of building libraries and then combining libraries and other code into the overall application. The distinction between *libraries* and *languages* is often overstressed, however—to build the application, a programmer must learn the "language" of the library, and often the effort required to master libraries is much greater than the effort required to master the traditional "programming language" from which it is used.

As we see it, what is really going on is that the normal process of building programs involves (by its very nature) the construction of new languages—layered library interfaces, etc. Languages are extended by adding new data types (record types, classes, etc.) and new functions or methods. Traditional programming techniques limit such language construction to extending the *vocabulary* within the existing syntax.

2.3 Software Engineering Issues and the Macro Myth

A lack of macros is often touted as a *feature* of conventional programming languages—it is argued that macros make it too easy for programmers to create unreadable dialects of languages. (Wirth, among others, has made such claims [?].) Since we disagree, we refer to this as the "Macro Myth."

Actually, we believe that there is considerable merit to this claim—defining new syntax *is* an order of magnitude "more serious" than defining new vocabulary; it should not be done casually. But the argument is not as strong as it seems at first glance, for four reasons:

- In many cases, defining a small amount of new syntax can greatly *increase* the clarity of code without putting an excessive burden on a reader who has never seen that syntax before. The solution is not to bar syntactic extensibility, but to use it tastefully. Normal programmers should be discouraged from casually inventing new syntax, but when significant new abstractions are introduced, a graceful and appropriate syntactic extension is often the most appropriate way to add it to the language.

- In many other cases, "syntactic extension" facilities are not used to define *new* syntax, but to *reimplement* old syntax [KdRB91]. Lisp programmers, in particular, often implement optimized versions of constructs by writing powerful macros that analyze expressions and transform them into equivalent expressions with different performance characteristics.

  In some cases, this is simply straightforward optimization, implemented outside the compiler proper. In other cases, it may be necessary to adjust the performance characteristics of a construct to suit the underlying system, *without* breaking the abstraction presented to higher-level code [KL]. At first glance, it may seem that reprogramming language features violates abstraction—programmers write lower level code than necessary to achieve the proper result.

  In practice, however, the alternative is usually to bypass abstraction layers entirely to achieve acceptable performance—an abstraction layer is only useful if is cost is acceptable. By making it possible to customize the implementation of an abstraction, the abstraction layers can be maintained rather than bypassed, preserving important separations of concerns—the correctness of code relative to underlying abstractions vs. the *efficiency* of implementation of those abstractions.

  In still other cases, the capabilities of a construct are simply *extended* in a backward-compatible way. For example, it is legal and desirable for a standards-conforming Ada implementation to provide garbage collection—the availability of garbage collection does not break programs that do not rely on it. Unfortunately, Ada's design makes it impossible to add garbage collection to the language *at the language level, as allowed by the standard* in a portable way. Programmers must either lobby the implementors of Ada to provide garbage collection, and to maintain the feature in future releases, or they must rely on awkward and error-prone programming conventions to provide garbage collection themselves.

- In some cases, the creation of a new and possibly incompatible language or dialect is inevitable; for example, due to a fix of a troublesome language flaw, or to a shift in programming paradigm (such as changing some features from imperative to declarative form). In such cases, it is desirable to preserve the parts of an earlier language that are still appropriate, while changing others. (All other things being equal, it is better to have a slightly incompatible new dialect than an entirely incompatible new language.) This can minimize the impact of incompatibilities, including code translation costs and cognitive load on programmers. If an extensible system can be adapted to implement the new language, it may also allow the use of an existing high-quality language implementation for the new language, rather than requiring the development of a new implementation from scratch.

- In many cases, libraries and even application programs are actually *domain-specific programming environments*, and are best cast as such. Extending a language in domain-specific ways is often an alternative to the creation of entirely new programming systems, rather than a dilution of the consistency of a single programming language. If a language is too rigid, the battle has been lost—a new language (often in the guise of a library or application) will be created from scratch.

(Examples will be given in Appendix A and Section 11. As will become clear, the boundaries between some of these issues are intrinsically fuzzy.)

Many of the criticisms of extensible languages are overstated, at least. The critics are often unaware of the software engineering principles used by people who use macros tastefully. Rather than barring macros outright, therefore, it is more appropriate to restrict their use to the "worthwhile" cases by proper education, programming style guidelines, and perhaps automatic mechanisms that restricts their use to "authorized" programmers.[3]

In providing these features, it is of course desirable that the facility not be error-prone—in fact, it should encourage *early error detection*.

In general, our argument is not that languages should be entirely modifiable; a language designer may have good reasons for keeping a language fixed in certain ways. However, the current (extreme) fixity of languages and implementations has two drawbacks:

- Many useful language extensions are difficult or impossible to implement cleanly and efficiently by anyone except the compiler developers—despite the fact that many of these extensions are straightforward and depend only minimally on "compiler" functionality. Compiler writers could expose *just a little more* of the implementation of a language, and make it far easier for systems programmers to build things like persistent object stores, garbage collectors, concurrency control systems, and so on.

  The traditional conceptual boundaries between "languages" and "libraries" lead to awkward, limited, and/or unreliable *middle-level* support software. Even if *most* programmers are not allowed to "mess around with the language," there is a class of *system software developers* who should be allowed to do so—and should supported in their attempts to add value to language implementations.

- Even if language designers and implementors continue to seal off most of the implementation of languages from the users, programming languages should provide mechanisms for incorporating linguistic features into *application* programs; inevitably, higher-level languages and language-like application environments will be designed and built for application-specific purposes. Without proper language support, they will continue to be built incoherently and badly, due to normal most developers' lack of expertise in programming language design and implementation. A general-purpose programming language should therefore provide a *clear and reusable framework* for the implementation of higher-level domain-specific software.

  Even language designers who believe their languages are too good to let others modify should find this appealing: they should see it as an opportunity to get implementors of higher-level software to incorporate *their* well-thought-out linguistic framework into the higher levels of system design.

[fwd ref & summarize appendix]

[3] This is a rough syntactic analogue of the "unsafe module" feature of Modula-3, which requires that modules using dangerous low-level features be flagged as such. In general, such constraints are usually enforced by extralinguistic mechanisms such as coding guidelines, often augmented with source-code control software. For example, important kernel code may be unmodifiable by unauthorized programmers, and authorized programmers are bound to preserve backward compatibility for any important programming interface, unless explicitly authorized to break dependent code.

3 Desiderata for a Better Language Framework

3.1 Multiple Audiences

3.2 Features

[need a list:

Modularity, good default components...

Easy snarfing of general-purpose programming features (procedural programming ability)—for escapes to general procedural programming from within a different framework.

Good basic programming features (scoping, encapsulation, etc.)

Easy snarfing of linguistic features (scoping and encapsulation, mechanisms, etc.)

Ability to override built-in language features to refine functionality or to disable it if it conflicts with something.

Good surface syntax, so people won't be inclined to monkey with it more than necessary, even though they can.

Extensibility in surface syntactic ways (for rule or constraint declarations, etc.) as well as semantic ones.

Parsing help for data as well as code: good default read/write/display/pickle libraries

Surface syntax extensibility for data as well as code, to avoid a proliferation of yacc hacks or worse...

Layered design with simple learning curve—easy to do simple stuff, not much harder than necessary to do amazing stuff. (Not easy.)]

[alternative ideas:

Standard libraries (with picklers, etc.)

Persistent stores to avoid needless translations, sharing problems

None of these is incompatible with our approach, and generally are enabled by our stuff. Gotta be tasteful, though—no silver bullet.]

[Many of these are well-understood principles of language design, and it it is natural to extend them to language framework and library design—library design *is* language design. Library designers should have language support...]

4 Macro Usage Types

Macros can be used to define a variety of constructs. To understand what a good macro facility must do, it is important to understand the kinds of uses to which macros are put. We will present four basic categories of increasing sophistication, and which require increasing power from the macro system.

4.1 Pass-By Name Constructs

Many control constructs have the semantics of *pass--by-name procedures*; they resemble normal procedures, in that their argument expressions are passed to them and the body of the procedure uses those arguments.

However, for pass-by-name constructs, the arguments are not evaluated *before* calling the procedure; the procedure itself decides whether arguments should be evaluated, when they should be evaluated, and in what order.

A simple example of this is an OR construct. OR can be written as a normal procedure which simply takes arguments and returns the logical or of them. This is may not be what is desired, however.

We may want an OR that only evaluates its arguments in order, and only evaluates as many arguments as necessary—if one returns a true value, the OR routine should return that value without evaluating subsequent arguments. For example, suppose we call OR with arguments that are themselves procedure calls, e.g. `or(foo(),bar())`. If the call to `foo` returns a true value, we would like for OR to avoid calling `bar`.

This OR is not a normal (pass-by-value) procedure—the "short circuiting" of evaluation of the second argument makes it a control construct, like an if statement. (Similarly, we may have a short-circuiting AND, which returns false immediately if its first argument is false, without evaluating its second argument.)

This short-circuiting of argument evaluation is not just an efficiency hack; if the procedure `bar` has side effects (e.g., updating values of variables, modifying data structures), the results of evaluating it in the normal pass-by-reference way would be different from evaluating it pass-by-name.

More generally, a pass-by-name procedure may need to carefully control the order of evaluation, and may need to evaluate some arguments an arbitrary number of times. For example, an iterative operator might be desired—one that evaluates an expression repeatedly to transform data until the procedure finds no more changes that need to be made.[4]

Generally, when defining a construct like this, we want to control when and whether argument expressions are evaluated, but we do *not* want to change the scope rules of the language at all. For example, suppose we call or with argument expressions that have variable references: `or(foo(a,b),bar(c,d))`. The argument expression `foo(a,b)` refers to the variables `a` and `b` at the site of the call to `or`.

To keep things understandable, we want to preserve lexical scope—`a` and `b` should always refer to the lexically apparent variables `a` and `b` at the call site, rather than some other variables `a` and `b`. For example, passing the expression `foo(a,b)` into `or` should not change the meaning of `b` if there happens to be a *different* variable named `b` that is visible to the code for `or`.

Other well-known examples of this kind of construct are sequenced conditional constructs (e.g., `if`...`else if`...`else`...`end`), `repeat`...`until` loops, `while` loops, etc.

4.2 Binding Constructs

While call-by-name constructs are common, they are definitely not the whole story. Programmers often use macros to define constructs that *control scope* by introducing *new* lexically-scoped variables.

For example, loop constructs often define iteration variables. Pascal's `for` statement is an example of this. The following use of `for` defines an iteration variable `i`.

```
for i := 1 to 10 do
   write(i);
```

We would like to be able to write a procedure-like construct that has this kind of effect. For simplicity, we'll use procedure-calling syntax so that we can focus on scoping and control issues rather than superficial syntactic issues. (These "surface" syntax issues will be addressed later.)

[4] An example of this might be a peephole optimizer, which repeatedly scans through assembly code finding opportunities to do local simplifications. Each local simplification may reveal more opportunities for the next pass to recognize things that can be simplified. Another example is a general "fixpoint" operator, which may iterate indefinitely until no more opportunities for changing things are found. Fixpoint computations are very common, e.g., for massaging complex data representations from multiple semantically equivalent forms into a canonical form that multiple tools rely on.

If we use a procedure-call-like syntax, the above code becomes:

```
for(i,1,10,write(i));
```

Notice that this is definitely *not* a call-by-name construct. The argument `i` is *not* an expression that refers to anything at the call site. It is just a *name*, and the `for` construct should create a *new* variable by that name. That's the whole point of the first argument to `for`—it is just a name, and the `for` construct itself is supposed to define the loop variable.

The argument `write(i)` isn't passed by name, either. The `i` is intended to refer to the loop iteration variable `i` created by `for` itself, not whatever `i` is in scope when `for` is called. (There likely isn't a variable `i` in scope at that point, and if there is, it's the wrong variable anyway.)

The `for` example is interesting in that the normal lexical scope rules *are* preserved, from the point of view of the user of the construct; pass-by-name is insufficient to implement `for`, however, because it is itself an operation that creates a new scope.

Luckily, there is a very clear and natural programming idiom that allows the straightforward definition of constructs like `for`, given a sufficiently clean and expressive macro system.

Other kinds of constructs require more sophisticated manipulation of scope. For example, an object system with encapsulated instance variables may require that the instance variables be visible (only) to the code for methods of the class. In effect, entering a method introduces a new scope, where the fields of an object are visible. Likewise, a monitor definition may introduce a new scope, creating variables that are visible only to monitor methods. [**discuss later...**]

4.3 Constructs That Violate Lexical Scope

Still other kinds of simple constructs require the ability to violate lexical scope. Lexical scope is almost universally agreed to be the right *default* scoping rule, but there are particular cases where a different scope rule is useful.

A simple example of this is a breakable loop construct. It may be desirable to add a `break` statement that interacts with looping constructs like `while` and `for`, allowing a controlled exit from the loop at an arbitrary point in the loop body.

For example, we might use a breakable `while` loop this way:

```
while (more_data_to_process())
{
  ...
  if (found_answer())
  then break;
  ...
}
...
```

Which is roughly the same as using the (much less structured and notoriously unsafe) `goto` statement and a label:

```
while (more_data_to_process())
{
  ...
  if (found_answer())
  then goto loop_exit;
  ...
}
loop_exit:
...
```

Notice that the `goto` version is less clear, and we have introduced a label that is visible to the programmer unnecessarily. We would like to be able to write a loop macro and a break macro that hides the goto and label. The label `loop_exit` is intended to be used only to break the loop, and probably should not be visible in the outer scope—it should only be visible to the code that implements the `break` statement.

This is a very simple example, and more sophisticated constructs raise more subtle problems of scoping control. For example, rather than just having breakable loops, we may wish to have a structured exception-handling mechanism with try blocks and catch clauses.

4.4 More Sophisticated Analyses and Transformations

The above categories of macro usage are all based on relatively simple processing of constructs; an `or` construct can defined much like a normal procedure, as long as the programmer can use pass-by-name arguments. A `for` macro is almost as straightforward, and a breakable loop is only slightly more complex. In all of these cases, the programmer provides a "body" expression, much like a procedure body, and shows where to plug in the arguments.

Macro systems like Lisp's allow far more control over how constructs are implemented. Programmers in Lisp often define new constructs that require the ability to analyze and transform their arguments; they use the Lisp language itself from inside macros to write procedures that analyze expressions as data structures, and generate the appropriate transformed expressions as data structures. Philosophically, this is compiler writing, rather than normal coding, though the distinction is blurry. The programmer may provide a library of macros and subroutines, which essentially extends the compiler with new constructs and optimizations.

This may sound like a very bold thing to do, but it really isn't that big a step. Many of the analyses and optimizations are quite simple, and the new constructs often fit in quite well with the basic language. They may not even extend the language at all, but provide an optimized version of existing constructs.

In the following, we will present several examples that may clarify this idea.

4.4.1 Record and Class Definition Facilities

In some languages, the built-in record definition facility may be weak, or even nonexistent. Programmers may wish to provide a new record definition facility, or an object-oriented class definition facility. This facility may be entirely new, adding an important feature to the language. On the other hand, it may simply *reimplement* a feature of the language to *improve its performance characteristics* for a class of applications. We will focus on the latter case first.

Records. [This example is too long—maybe not the best example, either...]

In most languages, the compiler has complete control over the representation of records. For example, a C compiler typically concatenates the fields defined for the record (struct), in the order specified in the struct definition. The C compiler also limits the programmer's ability to choose representations for the basic data types. For example, the boolean type may be implemented as a four-byte value, rather than a single bit or byte, because operations on words are faster than operations on bits or bytes on many machines. The compiler concatenates the record's fields, and inserts padding as necessary so that the fields are aligned on the proper address boundaries. (For example, a double-precision floating point field may need to be aligned on 8-byte boundaries on many machines. The compiler may insert unused "padding" in the layout so that each field is aligned properly.)

Because of this, the representation of a record is sometimes very inefficient. A boolean field that could be represented by a single bit may take 64 bits in some cases, because the compiler represents the boolean with a 8-byte value, and may add another four bytes if the following field requires 8-byte alignment.

Programmers therefore often "pack" their critical data structures by hand, to reduce the space cost of the compiler's layout rules. They define the fields in an order that results in a good physical layout, rather than in an order that reflects the logical grouping of related fields. They may use explicit bitfield declarations to force booleans to be represented as single bits, etc.

For many situations, it would be more appropriate to define a packing algorithm that chooses a representation automatically, optimizing for either speed or space, as desired. Then programmers could write their records definitions in a natural, logical way that reflects the logical roles of those records in the programs.

[the following is too detailed—it relies on knowledge or trust about Lisp macros to understand that this is actually easy... hard to give the motivating examples this early in the paper w/o the background implementation knowledge being clear yet.]

If C had a macro system like Lisp's, this would be easy. The record definition construct could be redefined by defining a record-definition macro that intercepts each record definition, analyzes it, restructures it, and passes it on to the normal built-in record definition facility built into the compiler.

For example, a record definition generally includes a list of field definitions; in C, this list is not available to the programmer—it's entirely internal to the compiler. In Lisp, the text of a record definition construct is converted to a list representation *before* macroexpansion. A macro can examine the list, and perhaps reorder it, before passing it on to the compiler for compilation.

A simple structure packer could look for a keyword saying to optimize for space, and if that keyword is present, sort the list by descending sizes of the field types. This would put the largest fields at the beginning, where no padding would be required, followed by the smaller sizes with their looser alignment constraints. The type specifications for small types (e.g., booleans) could be converted to bitfield specifications, to ensure that the compiler woul pack them tightly rather than expanding them to machine word size for speed.

[why this is an interesting example:

*** not really defining a new abstraction—salvaging an old one (records and fields) by adjusting its performance characteristics so that it doesn't have to be bypassed. This makes it possible to write layered code by keeping the contract between layers fixed (or nearly fixed) while adjusting the underlying layer when necessary. (Cite Hoare about a layer of abstraction only being usable if it doesn't hide a big performance hit? Talk about mechanism and policy, and separation of concerns of correctness from efficiency? blah blah...**

*** it's a performance-critical systems programming issue; most people who know Lisp-style macros only know them as a feature of a high-level language, not a medium-level systems hacking language like C. Extensible compiler stuff seems very appealing for low-level programming, because you can transform code into a close-to-the-metal efficient form.]**

Classes and Methods. **[other, more sophisticated examples, which require a little more machinery:**

*** reimplementing classes sparsely so that they look like normal records and fields at the language level but are implemented like property lists to handle situations where you may have a zillion fields, but most have default values you dont actually need to store. (Briefly describe metaclasses, Cite Gregor...)**

*** reimplementing records with compression.**

These examples require the ability to transform accessors, not just layouts—must consistently refine the code along with the data.]

4.4.2 Data Analysis and Optimization

4.4.3 Code Analysis and Optimization

5 Background (Part I): Implementing Lexical Scope and Pass-By-Name

To explain how to implement lexically-scoped macros, we must first explain how to implement lexical scope. The mechanism we describe is simple and well-known; it is used in many compilers for Scheme and functional languages, and is applicable to any block-structured language (e.g., Algol and its descendents Pascal, Ada, Modula, Java, etc.). C is also approximately block structured, and the same basic mechanism will work for it.

The basic idea of lexical scope is very simple: a variable name should refer to the nearest enclosing binding of the variable in the textual nesting of the program. When you see an occurrence of a variable name in the text, you can find the definition of the variable it refers to by looking progressively outward through the nesting of expressions until you find a binding construct that binds that name.

There is also a straightforward procedural mechanism for preserving this association in an interpreter or compiler, as explained below.

We will also explain a traditional idiom for implementing lexically-scoped control structures in terms of first-class procedures; a generalized, compile-time version of this basic idiom is the basic mechanism of our macro facility.

5.1 Basic Features of Scheme

We will use Scheme for our examples, because it has a simple surface syntax and it is actually the language for which the most advanced macro systems have been developed. To make the examples comprehensible to readers without a reading knowledge of Scheme (or Lisp), many examples are accomanied by footnotes giving a translation to a more conventional syntax, and Appendix B provides a quick overview of Scheme and its syntax and semantics.

We will briefly summarize the relevant features of Scheme here, to help in the reading of our examples, but the reader should consult the appendix if this is insufficient.

For readers new to Lisp and Scheme, the first thing to recognize is that Lisp uses fully parenthesized prefix expressions, where the operator name comes first, followed by its arguments, e.g., we add 1 and 2 with the expression `(+ 1 2)`, rather than `(1 + 2)`.

There is no distinction between operator syntax and procedure call syntax. A call to the function `foo` with arguments 1 and 2 is therefore written `(foo 1 2)`, just like the call to `+`, but unlike conventional languages' `foo(1,2)`. The procedure name goes on the inside of the parentheses, and no commas are necessary.

Scheme is dynamically typed, which means that we need not declare the types of variables and procedures; this eliminates unnecessary declarations from out examples.

For the most part, Scheme code can be read like code in any other procedural language, because Scheme is a procedural language. The syntax is terse, with few keywords, and heavily parenthesized, but noticing the indenting in the examples should clarify the structure.

[**For the first few examples, we will provide translations in a more conventional Algol-like syntax... this should ease the transition to Scheme syntax for non-Lispers...**]

5.2 A Simple Interpretive Model

5.2.1 Recursive Expression Evaluation

5.2.2 Managing Binding Environments

5.2.3 Environments and Closures

5.3 Applying the Model in a Compiler

5.3.1 Recursive Outside-In Compilation of Expressions

5.3.2 Binding Environments and Compile-Time Environments

5.3.3 Environments and Closures Again

5.3.4 Compile-Time Environments and Compiler Phases

5.3.5 Handling Analyses and Optimizations

5.4 Implementing Pass-By-Name and Control Constructs With Thunks

Recall that one of the major uses of macros is to define pass-by-name control constructs. Pass-by-name is easy to simulate in languages like Scheme (and ML, et al.), which have first-class lexically-scoped procedures.

We will explain this technique, which is the inspiration for our solution to the macro scoping problem. The simple solution presented here is not a full solution, however. As we will explain later, we can factor out the runtime costs by doing the work at compile time; we can also extend it to handle macros that are not pass-by-name.

Since a first-class procedure is really a closure, it captures the environment in which it was created, so that when it is evaluated, its variable references will be resolved in that environment—the "meaning" of the procedure doesn't change. This is exactly the effect that we want for the arguments to a pass-by-name construct.

We can therefore ensure that the arguments to a pass-by-name construct (e.g., `or`) are lexically scoped if we wrap the expression up as a procedure, and create a closure at the call site. Then we can pass the (pointer to) the closure into the pass-by-name routine. Inside the routine, the expression can be evaluated by simply calling the closure—it will restore its environment of definition and execute its body in that environment.

Consider the `or` example. If we attempted to write `or` as a plain (pass-by-reference) procedure, without worrying about pass-by-name, it would look like this:[5]

```
(define (or a b)
   (if a
       a
       b))
```

and we could use it thus:[6]

```
   ...
   (or (foo a b) (bar c d))
   ...
```

To get the effect of pass by name, we simply wrap the argument expressions in lambda special forms:[7]

```
   ...
   (or (lambda () (foo a b))
       (lambda () (bar c d)))
   ...
```

Now when the argument expressions are evaluated, they will create zero-argument first-class procedures, whose effect *when called* will be to evaluate the wrapped expressions.

A zero-argument procedure used for this purpose—fixing the meaning of an expression in a particular scope—is called a *thunk*.[8]

Thunks make it easy to capture the meaning of an expression before passing it to a routine. For this to work, the routine using the thunks must be modified slightly as well—since it is not being passed raw values, it must call the thunks to force evaluation and get the actual values.

We can modify the `or` procedure to expect thunks, and call them when their values are needed. A first stab at this might look like this:[9]

```
   (define (or a b)
      (if (a)
          (a)
          (b)))
```

(The parentheses around `a` and `b` convert them from plain variable references to zero-argument procedure

---

[5] Or, in pidgin C/Algol syntax:

```
procedure or(a,b)
{
    if a
    then a
    else b
    endif;
}
```

[6] Or, in pidgin C/Algol syntax:

```
    ...
    or(foo(a,b),bar(c,d));
    ...
```

[7] Or, in pidgin C/Algol syntax:

```
    ...
    or(make_procedure () { foo(a,b) },
        make_procedure () { bar(c,d) })
    ...
```

[8] This basic mechanism was developed for handling call-by-name in Algol compilers; the compiler automatically generated code to create closures of pass-by-name arguments. [Ing61].

[9] Or, in pidgin C/Algol syntax:

```
procedure or(a, b)
{
    if a()
    then a()
    else b()
    endif;
}
```

calls to the thunks, which are the values of the argument variables `a` and `b`.)

There's a problem with this attempt, however. For pass-by-value, it was safe to use a value of an argument multiple times—as `a` is used in the condition part and again in the "then branch" of the `if` expression. The actual argument is evaluated *before* calling the procedure, and the multiple uses are just multiple lookups fetches of the value of the argument variable.

Calling a thunk twice is *not* the same thing. In the thunk-handling version above, we evaluate the expression `(a)` twice, which evaluates the body expression twice as well. This is not what `or` is supposed to do. We can fix this by changing the code for `or` slightly: we evaluate `(a)` once, and cache the result in a local variable:[10]

```
(define (or a b)
  (let ((temp (a)) ; call thunk a, save value
     (if temp      ; a true?
         temp      ; if so, return a's value
         (b))))    ; if not, call b, return value
```

Notice that the introduction of a temporary variable here is [**fundamental**]. [**...awkward...**]

When pass-by-name is being used, the goal is to give the programmer *control* over when arguments are evaluated. It is therefore the responsibility of the programmer who implements the pass-by-name routine to ensure that the arguments are evaluated exactly when intended. A compiler cannot automate this, because the compiler doesn't know what the programmer's intent is. This problem also occurs with macros, whether or not they are lexically-scoped, and it is the main reason why macro definitions must always be written with some care.

[10] In pidgin C/Algol:

```
procedure or(a, b)
{
    let temp = a() in
    {
        if temp
        then temp
        else b()
        endif;
    }
}
```

5.5 Problems Implementing Binding Constructs

The above solution to the call-by-name problem works, and preserves lexical scope, if programmers are willing to modify their programs to create and call thunks when the effect of pass-by-name is desired.[11]

Using thunks has two drawbacks, however:

- *It is not general.* It handles only pass-by-name constructs, not *binding* constructs like `for`, or constructs that are not lexically scoped, like breakable loops.
- *It is fairly slow.* Closure creation and calling can be surprisingly fast, with a good compiler—a handful of machine instructions each—but is still significantly slower than executing the same code inline. It is therefore not suitable for defining very frequently-used constructs whose overhead must be negligible.

[**need transition here**]

6 Background (Part II): Macros and Preprocessors

In this section, we provide background material for understanding basic issues in macro systems and in implementing lexical scope.

We will briefly describe some existing macro systems, and then describe the normal scoping mechanisms used in many compilers for lexically-scoped (but not extensible) languages. These latter mechanisms form the basis for our lexically scoped macro facility, which differs significantly from conventional macros.

[**need to talk about precompilers too...**]

6.1 Simple Substitutional Macros

Macroexpansion is typically accomplished by "textual" substitution, often thought of in terms of character strings. A macro is defined in terms of a *pattern* and a *body*. The macroexpander's job is to detect occurrences of the pattern, and then replace them with

[11] This can be automated by the compiler, as was done in some Algol compilers. It requires the compiler to detect when a pass-by-name argument is being passed at a call site, and wrap the arguments in thunks. It also requires the compiler to notice where a pass-by-name argument is being used inside a procedure.

occurrences of the body. The result of this substitution is passed on to the interpreter or compiler in place of the pattern occurrence. So for example, a C macro to swap the values of two variables (e.g., for sorting) a value might be defined as follows,[12] using the `#define` preprocessor directive:

```
#define SWAP(var1, var2)  \
  { int temp = var1;      \
    var1 = var2;          \
    var2 = temp           \
  }
```

and used thus:

```
 { int a = ...
   int b = ...
   ...
   if (...)
      SWAP(a, b);
   ...
 }
```

Macroexpansion happens before compilation, in a separate pass over the source text—traditionally, an entire file is macroexpanded by running it through a program called `cpp`, and the output of `cpp` is sent to the actual C compiler program, which doesn't understand macros at all.

The call to `SWAP`, above, is rewritten by `cpp` and the text passed to the C compiler looks like

```
 { int a = ...
   int b = ...
   ...
   if (...)
      { int temp = a;
        a = b;
        b = temp;
      };
   ...
 }
```

[blah blah... problem with scope of `temp`, which we'll discuss later, using different example...]

[12]Notice that `SWAP` is not an inline procedure; given C's (and Scheme's) pass-by-value procedure calling, SWAP cannot be written as an procedure—it refers to the variable bindings (storage) of its argument variables, not just their values. If the values were simply passed to a swap procedure, the procedure could not update them—it would have its own copies (in its argument variables, which are just local variables) which it could update, but it could not update the variables visible at the call site.

A macro is *not* a procedure—it's a textual substitution—but the C macro preprocessor is oriented toward procedural syntax. Macro call patterns can be simple literals which expand to a fixed pattern,[13] or they can be patterns that *look like* a procedure call, but are not.

As in the case of `SWAP`, these call-like expressions can take arguments. While the body of a macro is substituted in place of a "call" to the macro, argument expressions are substituted in the *opposite direction*—into the the appropriate places in the body. (In the above example, `a` and `b` are substituted into the macro body for `var1` and `var2` when the body is substituted for the call to swap.)

Several points about the C preprocessor:

- It is recursive, and handles nested macro calls.
- It does not "understand" much of C syntax. It uses a very simple scanning algorithm and is oblivious to any syntax that doesn't look like a preprocessor directive or procedure call. Argument expressions may include control structures, or just about anything else, but the C preprocessor does not interpret them as such—it just treats the text of the argument as a string which should be substituted into its output where the argument name occurs in the body.
- Because of this, it does not understand scope. It looks for literal occurrences of macro names; during expansion, it looks for literal occurrences of argument names in the body, which tell it where to substitute argument expressions. It does not notice any declarations which may be significant to the *meaning* of variable names. (In the above example, there is a potential problem with the temporary variable `temp`, which will be explained later.)
- This simple textual substitution model is quite limited. The main mechanism is simple pattern matching and substitution, and the programmer has little control over macroexpansion. There are features for string concatenation (so that argument strings can be expanded into a single expanded string, rather than two separated by a space), but not much else.

[13]For example, to define constants (as in `#define PI 3.14159`), since C has no other facility for defining named constants.

This simple approach to macroexpansion means that little syntax checking is done at macroexpansion time, and the programmer has no way of adding any. The macroexpander blindly subtitutes text strings and passes the result to the compiler. If the expression is legal C, the C compiler will accept it. If it is a C syntax error, the compiler will complain.

The programmer also has no opportunity to make a macro analyze its argument expressions, or the context in which it is called, and provide different translations for different circumstances. (For example, in `SWAP`, we used an integer variable `temp`; this limits its use to swapping integers. There is no way for the macro to examine the types of its arguments and define `temp` appropriately.)

6.2 Lisp Macros

Lisp macros are far more powerful than C macros. In Lisp, there is a standard representation of expressions as data structures, and the programmer can write an *arbitrary Lisp procedure* to transform this data structure before it is passed to the compiler or interpreter.

To understand how this works, you must understand the general structure of a Lisp (or Scheme) system. **[is this redundant... do we explain the reader above?]** When a textual expression is to be compiled or interpreted, it is first *partially parsed* by a routine called the "reader." This routine "reads" the text of the code, and constructs a data structure that reflects the *surface syntax* of the expression. Lisp code consists of parenthesized prefix expressions, and there is a straightforward transformation into a tree data structure consisting of possibly nested lists and "atoms." (Atoms include numbers and "symbols" which are often used to represent identifiers in the source text.)

The power and expressiveness of Lisp macros comes from several things:

- Macroexpansion occurs after early parsing, and operates on a convenient representation of nested expressions. Macroexpansion therefore occurs in the middle of the front end of the compiler—the programmer can intervene in the compilation process at a more appropriate step than with textual macros like C's.
- There is a convenient and standard data structure used to represent programs; the programmer can effectively modify the parsing process in a portable way.
- The programmer has the full power of a general-purpose procedural language available in writing macro transformers, as opposed to just simple textual substitution.
- The syntax of Lisp is fairly simple and regular. This makes it feasible to analyze programs and transform them appropriately, implementing part of the "compiler" as macros, outside the compiler proper.
- Lisp macros can pattern match on variable-length argument lists, and either use an explicit representation of the argument list, or "peel off" arguments one at a time during recursive macroexpansion. This makes it easy to create constructs with options or variable numbers of components—e.g., data structure definitions that allow any number of fields, control constructs with optional branches, etc.

Lisp macros are *procedural*, allowing the programmer to write a transformation procedure that will be applied to a call to a macro.

[How redundant is this with earlier and later stuff?:]

Lisp code is initially parsed by a "reader" routine to yield a (usually nested) list representation called s-expressions. The nested parentheses correspond to the nested list structure that the reader creates. The macroexpander recognizes calls to macros, and calls the transformation routine specified for the macro.

For example, if we've defined a macro `swap` that takes two arguments, a macro call `(swap a b)` will be translated into a call to the `swap` transformation procedure with the argument expressions `a` and `b`. Note that the macro expander is passed a data structure representation of `a` and `b`—in this case just symbol objects used to represent the identifiers—*without evaluating them as expressions.* The transformer takes those arguments and returns an s-expression which will then be interpreted or compiled.

We can write a procedure to take the arguments (e.g., `a` and `b`) and return an s-expression that will do the swapping when it is executed, In this case the macroexpander will return a nested list representing a let expression and some assignments:[14]

[14] In pidgin C/Algol:

```
let temp = a in
{
   a := b;
```

```
(let ((temp a))
  (set! a b)
  (set! b temp))
```

While it's not the most convenient thing to do, we can write the **swap** macro definition procedurally, using the built-in procedure **list** to construct nested lists:[15]

```
;; ugly procedural macro definition for swap
(defmacro (swap var1 var2)
  (list 'let
        (list (list 'temp var1))
        (list 'set! var1 var2)
        (list (set! var2 temp)))))
```

Here we are telling the compiler what transformation routine to execute when it sees an expression of the form (swap *expression expression*). The specified macro transformer is simply a procedure that expects two arguments, and does what the body of the macro definition says. (The macro facility will "destructure" the call expression, passing the argument expressions, unevaluated, to the transformation procedure as the avlues of its arguments **var1** and **var2**.)

We use **list** to construct nested lists, and we quote the symbols that we want to appear literally in the resulting list. Where we don't want a literal, we use an expression that will be evaluated in the usual way. In this example, all of those expressions are just references to the argument variables of the macro transformation procedure, i.e., **var1** and **var2**. The *values* of those variables are the symbols that were passed as arguments to the macro transformer, which we assume name variables in the program being transformed, and those symbols appear in the resulting list.

For most purposes, of course, writing arbitrary transformation procedures by hand is overkill, and for complex macros it's very tedious and obscure. It is simpler and safer to use simple "substitutional" macros than to write a procedure that operates on an explicit data structure and emits an explicit data structure. Programmers usually prefer to write macros in terms of source code patterns and source-level substitutions, rather than modifying an "intermediate representation."

Lisp therefore provides a convenient way of writing simple substitutional macros, which is just a slightly syntactically sugared special case of general procedural transformation. This is really just an idiom, using the backquote and comma features [**described earlier ?**]; **[they're just what the doctor ordered, since backquote generates mostly-literal s-expressions and fills in the holes.]**

The following definition uses backquote to specify that the macro should simply return a list of a given structure (the backquoted list), except that certain "holes" in the list (preceded by commas) should be filled with the values of expressions computed inside the transformation routine.[16]

```
;; cleaner template-filling version of swap
(defmacro (swap var1 var2)
  `(let ((temp ,var1))
     (set! ,var1 ,var2)
     (set! ,var2 temp)))
```

We stress that this is just a particularly useful idiom—we're really writing a procedural macro, but the body of the procedure just uses backquote to construct the list that is returned. The body expression using backquote and comma is just a shorthand for the uglier expression using explicit calls to **list**.[17]

```
    b := temp;
}
```

[15] An equivalent (ugly, procedural) version using C/Algol syntax to pidgin syntax:

```
macro swap(var1, var2)
{
    list('let temp = ', var1, 'in {',
         var1, ' := ', var2, ';',
         var2, ':= temp; }');
}
```

[16] In pidgin C/Algol:

```
macro swap(var1, var2)
{
    ' let temp = [var1] in
      {
          [var1] := [var2];
          [var2] := temp;
      } '
}
```

Here we have extended our pidgin notation to use [and] to delimit unquoted expressions, much like Scheme's prefix use of comma.

[17] Backquote itself is an interesting macro. The character ` is interpreted specially by the reader, and the effect is to interpret the following expression as though it were nested in a call to the **quasiquote** macro. (For example, `(foo bar) is read as (quasiqote (foo bar)).)

quasiquote itself is just a (fairly simple) procedural macro. It analyzes its argument s-expression and generates code that con-

In effect, backquote just reverses the usual expression evaluation rules of Lisp: by default, nothing is evaluated except where the comma says to. This allows us to specify a mostly-literal list without explicit quoting and list-construction procedures. We use backquote to provide a template for the code we want to generate, and comma to fill in the holes in the template.

Most Lisp macros are written using this idiom, because it's simple and sufficient for most purposes—most macros are just stereotyped bits of code which need some filling-in.

On the other hand, some macros require much more sophisticated processing, and then procedural macros are very useful. A macro transformation procedure can examine its arguments, which may be complicated s-expressions, rather than simple symbols, and perform arbitrary analyses. Based on this analysis, the macro can use an arbitrary procedure to generate the appropriate transformed expression.[18]

6.3 Problems With Conventional Macros

Despite their power and their convenient features, Lisp macros are far from perfect. They share several weaknesses with more limited macro systems like C's, and these weaknesses sometimes become more apparent because it is natural to do more ambitious things with Lisp macros.

The best-known problems with conventional macros is that they are oblivious to scoping issues, making code transformations dangerous—even simple substitutional macros can have this problem. An expression that occurs in a given context may be substituted into a different context, and its meaning may be inadvertently changed.

For sophisticated uses involving more analysis and more sophisticated transformations, there is a generalization of this problem—the macroexpander has little *contextual information* about the expressions it transforms. It also has *an inappropriate representation* for many high-level purposes—the surface syntax of an expression does not directly reveal deeper properties of the expression, and it is the macro transformer's (often laborious) job to infer them.

6.3.1 Out-of-context name resolution.

The major scoping problem with conventional macros is that the association between *identifiers* in the program and *variables* is not obvious.

A macro body may contain an expression that refers to a variable that is visible at the point in the program where the macro is defined, but when the macro body is substituted into a macro call site, the same name may refer to a different variable—visible at the call site—or to no variable at all. The expression has been copied from one place in the program, where it seems to mean one thing, into another place in the program, where it means something quite different.

Because arguments to macros are substitutions in the opposite direction, the same problem can happen going the other way. An expression passed as an argument to a macro call may mean one thing at the call site, but another at the point where it is substituted into the body of the macro—identifiers used in the argument may refer to different variables.

Capture of definition-site names at the call site. The first problem occurs when a macro writer attempts to refer to variables that are visible at the definition site from within the body of a macro, and then the macro is used in a different scope, where the variable name is not visible, or refers to the wrong thing.

Suppose a macro is defined in a scope that encapsulates some data structures that the macro must operate on. For example, suppose that we would like to build a tracing routine `emit-trace` that takes an expression and writes its value to an output port. (A port is an **[output thingie, like a file descriptor in C].)** We may want to write this as macro, so that it can write the *expression* to the output port as well as its value, e.g.:

```
; define a variable port to say where to send trace output,
; initialize it with port returned by opening file "trace.txt"

(define port (open-output-port "trace.txt"))

   ...

; define a macro that writes an expression
```

structs a list of the that form—with the exception that comma'd expressions are used as-is, rather than interpreted as specifications of sublists to construct.

[18] For example, a macro that defines a record type may examine an argument list of fields, sort them, generate definitions of procedures to access them, add the new record type to a table of known types, etc.

```
; and its value to the port named port
(defmacro (emit-trace expr)
  `(port-display port
                 ',expr)
                 " has value: "
                 ,expr))
```

This macro's transformer **[defined yet?]** is a procedure which will return the mostly-literal body expression following the backquote. The argument `expr` is substituted into the body expression at two points, using the comma (`unquote`) operator. The first time, it's just after a quote (`'`), which will suppress its evaluation. The second time, it's *not* quoted, so it will be evaluated when the body expression is evaluated.

(Note that the comma's effect happens at macro substitution time—the value of the variable `expr` will be substituted into the backquoted s-expression. Once the backquoted list has been constructed by template filling, the whole list, including the quote, is passed to the interpreter or compiler.)

Now suppose we call the macro thus:

```
 ...
(emit-trace (2 + 2))
 ...
```

Assuming we're using an interpreter (for simplicity), here's the sequence of events that happens when the interpreter reaches this expression and evaluates it:

- The interpreter examines the expression, and finds that it has the syntax of a call, and starts with a macro name, `emit-trace`; it therefore calls the macro transformer for `emit-trace`, to transform the expression, handing it the s-expression verb—(+ 2 2)— as its argument `expr`.
- The transformer creates a copy of the backquoted expression, substituting `(+ 2 2)` for the comma'd occurrences of the argument name `expr`. The result is the s-expression:

  `(port-display port '(+ 2 2) "has value" (+ 2 2))`

  and the expander returns this expression to the interpreter.
- The interpreter interprets the evaluated expression. This is just a procedure call, so it:
  - evaluates `display-port` to get the procedure to call,
  - evaluates `port` to get the first argument
  - evaluates `(quote (+ 2 2))` to get the second argument; the quote form supresses evaluation, so the value is just the s-expression `(+ 2 2)`.
  - evaluates `(+ 2 2)`, which is another procedure call; normal recursive evaluation returns the value `4`.
  - It calls the display procedure with these arguments, and it writes its argument values to the output port: `(2 + 2) has value 4`

In most cases, this will work fine. But notice that when the interpreter evaluates the variable `port`, it does it in the environment of the call site. (That is, the interpreter doesn't do anything to change the meanings of variable names when it starts interpreting the result of macroexpansion. Whatever variables are visible at the call site are still visible.)

Suppose that `emit-trace` is called from an environment where a *different* variable named `port` is in scope.

For example, suppose that `emit-trace` is called from a site elsewhere in the program, inside a let expression that defines a local variable named `port`.

```
(let ((port (open-port "warning.log")))
   ...
  (emit-trace (+ 2 2))
   ...)
```

This time, when the interpreter looks up the variable `port`, it will find the *local* variable that is in scope.

Capture of call-site names in the macro body. The second problem occurs when a macro body defines variables internally, and the argument expressions substituted into the macro body end up referring to those variables by mistake.

Suppose we want to define a macro named `or`, as mentioned previously, which acts as both a logical *OR* operator and a short-circuiting control construct. Our `or` macro takes two arguments; it unconditionally evaluates its first argument expression, and if that value is true, it returns it as the value of the `or` expression. If the first expression's value is false, however—and only if it is false—it evaluates the second argument expression and returns the result.[19]

[19] This is the way `or` normally works in Lisp and Scheme, with

There's a minor problem to implementing `or`, which we must address before come to a major problem. We might try to implement it the completely obvious way, [**like our first stab at defining `or` or as a procedure using thunks to fake pass-by-name**]:

```
(defmacro (or a b)
  `(if ,a     ; a true?
       ,a     ; if so, return a
       ,b))   ; if not, return b
```

Here backquote is used to give the literal part of an if expression, and comma (unquote) is used to plug the transformer's arguments in. Backquote is thus used to define a template for an expression, and comma is used to fill in holes in the template with the results of any desired computation. (In this simple example, the computations are simply lookups of the values of the argument variables of the transformation procedure.)

[**the following description of the extra-evaluation bug (and the temporary variable fix) is now redundant with the new text on using thunks to implement pass-by-name**]

The problem with this is that the expression `a` gets evaluated twice, first as the condition and then to return its value (the "then" branch). This doesn't matter if the argument to `if` is a literal, but if it is an expensive expression, it will cost twice as much—worse, if it has side effects, they will happen *twice*, which is a bug.

(This kind of "problem" is common to all macro systems—arguments to macros are evaluated wherever they are plugged into the body. This is what makes it possible to define control structures with macros—you can ensure that arguments are evaluated only when and if they should be. It is also why it is dangerous to use macros for simple inlining; macros are *not* procedures; their arguments are not evaluated exactly once before the call.)

Instead, we might define an `or` macro like this:

```
(defmacro (or a b)
  `(let ((temp ,a))
     (if temp
         temp
         ,b)))
```

the exception that it can take any number of arguments. We will ignore the fact that Lisp and Scheme already provide `or`; note that it typically *is* implemented as a macro—there is no need for the compiler to understand `or` directly.

This macro will evaluate its first argument, then save it in a local variable named `temp` to avoid having to evaluate it twice. The "then" branch of the `if` simply returns the value by evaluating the variable (looking up its contents) so the expression doesn't get evaluated twice.

The use of a local variable solves one problem, but it creates another. When the macro is expanded, an argument expression will be substituted into the body in place of `b`.

Suppose we call the macro `or` with an argument expression that happens to use the variable name `temp`, which is intended to refer to something at the call site:

```
(let ((temp (member x temporary-employees))
      (perm (member x permanent-employees))
  (if (or temp perm)
      (display x "is an employee")
      (display x "is not an employee")))
```

Here the `or` happens to check to see if either of two variables has a true value. Unfortunately, one of them is named `temp`.

Expanding the macro, this example becomes

```
(let ((perm (member x temporary-employees))
      (temp (member x permanent-employees))
  (if (let ((temp perm))
         (if temp
             temp
             temp)) ; Sees wrong temp.
      (display foo "is an employee")
      (display foo "is not an employee")))
```

Here the name `temp` has been substituted into the else branch, where it unintentionally refers to the inner `temp`, rather than the outer temp, as intended. No matter what happens, the call to `or` will always return the value of `temp`, which is not what was intended.

Avoiding Name Clashes by Hand. [**gensyms, etc.**]

6.3.2 General Lack of Contextual Information

6.3.3 Lack of Block Structure.

7 An Interpretive Implementation of Lexically Scoped Syntax Extensibility

[**I haven't written this section yet, but the ma-**

terial is covered in Stephen Carl's masters thesis.]

7.1 Syntactic Closures

7.2 Advertent Capture for Binding Constructs

7.3 Explicit Violations of Lexical Scope

8 Implementing Syntax Extension Efficiently in the Compiler

[I haven't written this section yet, but the material is covered in Stephen Carl's masters thesis.]

9 Exploiting the Integrated Approach

9.1 Making Compilation Comprehensible

9.2

9.3 Revealing Compiler Phases

10 A New Look at Syntax, Parsing, and Compilation

Grammar and parsing issues in extensible languages have received very little attention compared to the grammar and parsing of conventional programming languages. Lisp has generally been given short shrift in the study of syntax, apparently because Lisp's syntax has been regarded as either entirely trivial, because it's "just parenthesized prefix expressions"—or as a "hack," because macros allow arbitrary manipulation of syntax in an uncontrolled, procedural way.

We think that Lisp raises several important issues of syntax that have been overlooked because of the emphasis on less expressive and better-understood languages. A large body of research and technology has arisen around the syntactic processing of conventional languages, but Lisp has gone along without the benefit—or the limitations—of this established knowledge.

Researchers in extensible languages would do well to consider why Lisp has so little trouble handling extensibility; even if you don't like fully parenthesized prefix expressions, you may want some of Lisp's flexibility in a more conventional-looking package. In that case, it is worth considering exactly *what* is special about Lisp syntax—and exactly what that implies for the design of extensible languages and their compilers.

We believe that Lisp exhibits several properties that other languages should emulate:

- *It is not context-free*, because it is grammatically extensible within the language,
- *it has a two-level syntax*, which is superficially trivial, but deeply powerful,
- *it has a natural, top-down model of parsing and compilation*, and
- **[what else?]...** .

Lisp's grammar is not context-free, because Lisp is too expressive to be limited in this way. Yet Lisp is easy to parse, because its surface syntax is trivial, and because *parsing is not a prepass*—it is integrated with interpretation or compilation in way that is so straightforward that it is generally overlooked. (Introductory programming texts using Scheme often present a simple Scheme interpreter in Scheme, and neglect to point out that it performs parsing, perhaps because the parsing is so obvious and easy that it doesn't seem to deserve the name.)

Lisp has a two-level syntax, in that its *surface syntax* is just parenthesized prefix expressions, but it has a higher-level, *extensible transformational grammar* that describes the really important aspects of its syntax.

The transformational component of Lisp grammar is seldom remarked on, because it is so simple and because it is usually implemented procedurally, rather than through compiler-generation tools that require a well-developed formalism.

In this section, we will attempt to explain the grammatical issues in Lisp and Scheme, and their implications for extensible languages generally. We will use some traditional terminology, and generally assume that readers have some idea what a grammar is, and particularly a context-free grammar. (We will provide explanatory footnotes, but the reader who is not fluent in syntactic issues may wish to consult a compiler textbook, such as [ASU88] if anything is not clear.)

10.1 The Myth of Context-Free Grammars

In the conventional (non-Lisp) programming languages community, the "syntax" of a language is almost always assumed to be context-free, or approximately context-free; context-free grammars are the *sine qua non* of modern programming language syntax.[20]

It is widely recognized—though seldom stressed—that real programming languages are not entirely context-free; context-free grammars are used to describe the "syntax" of the language, and parsers for these syntaxes are used as front ends to compilers. The grammars for these parsers generally leave out important syntactic issues, however, and later phases of the compiler must perform additional checking and analyses to determine whether a program is legal, as well as to compile it to executable form. That is, there are extra constraints on a language that are not captured by the context free grammar.[21] These checks are often performed by *ad hoc* procedures in the "action" routines specified for grammar productions, or by more structured mechanisms (such as attribute grammars).

Our view of this is that context-free grammars do *not* generally capture the actual syntax of programming languages—context-free grammars are stressed because they are mathematically well-understood, and because tools exist to process them automatically. In practice, they only express certain aspects of the actual grammar.

Most enhancements of context-free grammars *restrict* the language accepted by a compiler. That is, additional machinery is added to detect errors that are not detected by a parser for the grammar. In contrast, Lisp's context-sensitive *transformational* grammar allows the language to be *extended* as well, increasing the number of programs that can be accepted and compiled.

Context-sensitive grammars are often thought to be dangerous and difficult to understand and process; we believe that this is incorrect. *Some* context sensitive grammars are too weird to deal with, of course, but Lisp's grammar stands as an example that not *all* are—some are quite simple.

We find the traditional restriction of programming language grammars to CFG's to be completely unintuitive, because parsing is part of the overall process of compilation, not an end in itself—a parser does not simply recognize legal programs, it constructs an intermediate representation that encodes crucial facts about the program in ways that are simply not expressed by the context-free grammar. Many of these facts are context-sensitive.

We believe that the use of context-free grammars and conceptually separate parsing phases may not only limit languages' flexibility, but make them harder to implement as well. For example, error checking in a parser is often context-sensitive, and it is often encoded in *ad hoc* ways because a context-free grammar doesn't provide a natural mechanism for addressing the crucial issues.

10.2 Transformational Grammar

The traditional notion of context free grammars stems from Chomsky's early work [Cho56, Cho57], and his categorization of three kinds of grammars that require increasingly sophisticated processors. While Chomsky's scheme was originally devised to assist in understanding natural (human) language, it has great utility in understanding programming languages as well.

At the bottom of the "Chomsky hierarchy" are regular languages, which can be summarixed with *regular expressions*, now commonly used for lexical analysis of programming languages. These languages can be processed by finite-state machines, which have "no memory" except for the current state in a fixed graph of states (and an input buffer).

[20] Intuitively, a context free language is one which can be recognized by a parser that uses a stack to keep track of the nesting of expressions. A "context-free" grammar is not actually entirely context-free: the meaning of a token may well depend on the surrounding context, but the parser can keep track of the relevant information using a stack, and can consult the top of the stack to determine what's what.

In addition, the relationships between productions in the grammar provide a kind of context, by constraining the conditions under which certain kinds of expressions may legally appear.

Thus the contextual information required for processing a "context-free" language is minimal, but certainly *not* nonexistent.

[21] For example, in Pascal a variable must be defined before it is used. This is a syntactic constraint, but it is not captured by any context free grammar. Additional mechanisms must be used to ensure that "grammatically" correct programs do not violate this constraint.

Above regular languages (in the Chomsky hierarchy) are languages characterized by *phrase structure grammars*, which correspond fairly directly to the computer science notion of context-free grammars. Parsing these languages requires at least a stack, in order to recognize the nesting of expressions.[22]

Above context-free languages are *transformational* grammars, which include a context-free component which is used for initial parsing, and a *transformational* component, which transforms initial parse trees into a later representation.

In a Chomskian transformational grammar, a basic set of phrase structure rules describes the surface syntax of the language, and additional *transformational rules* describing higher-level regularities. For example, an English sentence might have a surface syntax form that reflects passive voice or active voice. A sentence in active voice will have a phrase structure that is different from the equivalent sentence in passive voice.[23] Transformational rules may transform this into a more basic (or more nearly canonical) form which eliminates that distinction; that is, the active-passive distinction is collapsed because the sentence "means the same thing" independent of which form was actually used to utter it.

For languages like Lisp and Scheme, the semantics of constructs are often defined in terms of lower-level constructs, by simple transformations. (Such definitions are often called "metacircular.") For example, Scheme's `let` binding construct is officially defined in terms of an equivalent expression using a `lambda` expression and procedure calling.

In actual implementations, this may be (but may not be) reflected by the implementation strategy of defining constructs with macros, which automate the translation into the more nearly canonical form. At some point, macro transformations bottom out into the "primitive" syntax understood by the compiler proper.

Notice that a transformational syntax is *not* necessarily *extensible*. **[But it's still not generally context-free.]** For example, before Scheme macros were standardized, the standard Scheme language was defined transformationally, but was not syntactically extensible.[24] A small set of *essential syntax* constructs was defined using English descriptions and denotational semantics, and a set of other *syntax* forms was defined in terms of transformations into essential syntax.

10.3 Predictive (Top-Down) Parsing with Extensible Grammars

Lisp and Scheme are parsed in two phases, which are both *top-down*,[25] using *predictive* parsing schemes[26] similar to those often used for the LL class of context-free languages.[27]

[22] A stack is generally required for nested expressions, in order to remember and match earlier events, e.g., ends of statements with their beginnings, after processing subexpressions.

[23] For example, "The girl hit the ball." vs. "The ball was hit by the girl." The phrase-structure parse of each corresponds roughly to simply diagramming the sentence, but does not reveal the deep equivalence of these sentences.

[24] Actually, Scheme macros still aren't quite standardized—the standard contains an appendix which defines an optional macro system. **[Say this earlier, ref from here.]**

[25] Top-down parsing recognizes higher-level constructs in the eventual parse tree before lower-level constructs. In Lisp terms, the outer expressions are recognized before the inner (nested) expressions, and this recognition is used to control expectations during parsing.

In contrast, *bottom-up* parsing may recognize inner expressions (subexpressions of nested expressions) *before* recognizing the enclosing expressions themselves. The parse stack may be used to store tokens whose syntactic roles have not yet been understood while the parser looks ahead to recognize isolated subexpressions. Later, the recognized subexpressions are grouped together when the enclosing expressions are recognized.

In either case, parsing is usually done during a left-to-right scan of the sequence of input tokens, which are read into a limited input buffer.

A top-down parser (such as a recursive descent parser) generally recognizes the beginnings of outer expressions quickly, as they are read into the input buffer—the "leftmost" parts of expressions are recognized as the beginnings of expressions.

A bottom-up parser (such as a shift-reduce parser) generally recognizes entire subexpressions only when their "rightmost" parts appear in the buffer. The recognition of subexpressions then may trigger the recognition of enclosing expressions whose constituents have already been read.

[26] Predictive parsing is top-down parsing without backtracking, and with a single token lookahead; intuitively, the compiler always "knows where it is" in a top-down parse, recognizing the outer expressions before its subexpressions. The parser only looks ahead a single token in the input stream, using it to quickly disambiguate which kind of subexpression it is encountering. **[Are Lisp and Scheme LL(1)ish, or LL(0)ish? you don't have to look an extra token ahead to disambiguate infix binary expression syntax, but you do have to look ahead of a left paren to see what the operator is... that's kind of missing the point, though—only the reader is anything like LL(1), and the interesting parser is OI(n) or whatever we want to call it, because it operates on trees, outside-in, independent of the branching factor.]**

[27] "LL" languages can be parsed by left-to-right scan of the input token stream (the first "L"), and by recognizing phrase categories as their leftmost few tokens appear during the scan (the second "L"). An LL(k) grammar requires at most k tokens

10.3.1 Reading (i.e., phrase structure parsing)

The initial phrase-structure parse of Lisp or Scheme is the recognition of s-expressions, i.e., parenthesized prefix expressions. Traditionally, the stream of characters that represents a program is parsed and a (usually nested) "list" representation of the program is created. The s-expression data structure is really a phrase structure (surface syntax) parse tree. Traditionally, this is done by a fairly simple recursive procedure called the *reader*, but "reading" is really parsing. The reader uses a stack (the recursion stack) to recognize a very simple (roughly LL(1)) grammar.

The resulting tree (nested s-expression lists) consists of nonterminal nodes (lists) and terminal nodes which may be literal numbers or strings or other data structures, or *symbols*, which can serve as identifiers (e.g., variable or macro names). At this point, all nonterminal nodes are of the same type, which might be called "compound expression of some sort."

After this initial phrase parse, the surface syntax parse tree is fed to an interpreter or compiler, which traverses it top-down to interpret it or to compile it.

10.3.2 Evaluation or Compilation

After this simple initial parsing, a much more interesting parse is performed. Traditionally, this parsing is performed incrementally, interleaved with evaluation (interpretation) or compilation, rather than as a recognizably separate parsing phase.

Parsing is top-down, in a way that reflects the semantics of the language. The top-level structure of an s-expression is examined and categorized, and an appropriate interpretation or compilation routine is called. That routine controls the interpretation or compilation of subexpressions, and may call the main evaluation or compilation routine recursively to deal with them. This recursion implements the top-down parsing as well as the evaluation or compilation.

In the case of a construct recognized as a macro definition, a special macro-definition-handling routine is called. It registers the macro definition, extending the grammar of the language. (In our system, this is done by simply extending the compile-time environment with a new syntax binding frame.)

In the case of a construct recognized as a macro call—that is, a compound expression with a macro name in the operator position—the macro processing routine (transformer) for that routine is called. This invokes the transformation routine specified by the macro definition, and the expression is transformed, and then compiled or interpreted by another recursive call to the compiler or evaluator.

10.3.3 Why top-down processing is natural and appropriate.

We believe that top-down parsing is more natural than bottom-up parsing, because it reflects the semantics of programming languages, and is more powerful. Programming language semantics are essentially top-down; for example, a reference to a variable makes no sense it all unless it occurs in a context where the variable is defined—and the definition must generally occur in a higher-level, enclosing expression (at least conceptually).

It is widely believed that bottom-up parsing is strictly *more* powerful than top-down parsing [ASU88]. For example, for any LL language (parseable top-down), a bottom-up LALR parser[28] can be constructed. The LL languages are a subset of the LR languages. It is often said that top-down parsing is therefore less powerful than bottom-up parsing, and most parsing tools are based on bottom-up techniques, or a mix of top-down and bottom-up techniques.

Why are we not convinced of the superiority of bottom-up parsing? Because *the proof of that LR languages are a superset of LL languages depends on both being context-free* **[I think...check Brosgol] [, and neither transformational nor extensible.]** [Bro74]. Lisp and Scheme are extensible transformational languages; for such languages, context-sensitivites are crucial, and top-down parsing is much more appropriate. Lisp and Scheme are LL-like, but not LL. The restrictions that make them LL-like are extremely helpful in avoiding the restrictions required for context-free grammars.

Consider a local macro definition, visible only within a `let-syntax` block. For example, we might define the for macro locally and use it within a block, as in Figure 10.3.3.

of lookahead in the input stream to distinguish between different kinds of expressions; a predictive parser relies on an LL(1) grammar, using a single token lookahead to uniquely determine what kind of expression is being parsed.

[28] LALR parsing uses (limited) lookahead (the "LA") LR processing, i.e., bottom-up parsing where the input is scanned left-to-right but "right-sentential" forms are recognized. (That is, expressions are recognized as their rightmost parts are seen.)

```
...
(let-syntax ((for (slambda (var init limit body)  ; changes grammar
                     (let ((var init))
                        (repeat-until body
                                       (>= var limit))))))
   ...
   (for i 1 10 (write i))
   ...)
...                                               ; grammar reverts
```

Figure 1: The for example again.

Phrase structure rule:

```
FOR_EXPRESSION ::= ( for ID EXPR EXPR EXPR )
```

Transformational rule:

```
(let ((ID EXPR)) (repeat-until EXPR (>= var EXPR))) ::= FOR_EXPRESSION
```

Figure 2: A phrase structure rule and transformational rule.

This introduces a new phrase structure grammar production into the language, shown in Figure 10.3.3. This includes two productions: a phrase structure parsing rule that describes the surface syntax, and a transformational rule that describes the transformation used to transform the phrase structure.[29]

Both productions are in effect only within the `let-syntax` block, and removed from the grammar when the block is exited.

Processing `let-syntax` top-down is easy; a new scope frame is added to the compile-time environment, making the new production available for the parsing of the body of the `let-syntax` form. The grammar of the language is quite naturally affected by which grammar-affecting constructs are in scope.

Parsing local macros bottom-up is tremendously more difficult; it would require that a *use* of the local macro be recognized *before* the enclosing *definition* of the macro is seen and recognized as such. It is difficult to imagine that a bottom-up parser for an extensible grammar could be anywhere near as simple as Lisp's top-down approach. Bottom-up parsing suffices for context-free grammars precisely because they are context-free: they do not directly express the natural nested and *scoped* structure of actual programming languages.

Consider the identifier `i` in the above example. Suppose that before entering the `let-syntax`, a different binding of `i` was in scope, and it referred to a function or macro, not just another variable.

In Lisp or Scheme, this is not really a problem, because the surface syntax is very simple and regular, and does not commit to the syntactic roles of identifiers. Only later parsing depends on the referent of an identifier. In a more traditional syntax, however, this could be a very grave problem, because the the initial phrase-structure parse may depend crucially on whether an identifier is a statement keyword, or the name of a local variable or procedure—or the name of a new syntactic form that shadows the old.

Consider the difference between a variable named `if` and the syntactic keyword `if` that begins an `if` ...`then` ...`endif` statement. Clearly, if the keyword `if` is not reserved, a bottom-up phrase-structure parse is quite difficult. The role of the identifier is fun-

[29] We have not shown the mechanism for variable binding during rule application—i.e., how particular subparts of the parse tree (e.g., the ID and three EXPR's) are communicated from the right hand side to the left, so that the correct structural transformation occurs. We have avoided cluttering up our pidgin BNF with variable binding issues, because they wouldn't show anything that isn't obvious from the program text and our discussion of how macros work.

In a sense, that's exactly our point—a lexically-scoped macro system is close to the ideal notation for a certain useful class of transformational grammars.

damentally a top-down issue, because it depends on nested scopes.

[where does this go?:]

An important side benefit of top-down parsing is *early error checking*. For example, revealing syntactic properties of identifiers in a top-down manner often makes it possible to detect syntactic errors immediately, at the points where they occur. For example, an illegal assignment to a syntactic keyword such as `if` or `for` can be detected immediately. Conversely, an attempt to use a syntactic keyword as the name of a variable is an obvious error. Without a top-down analysis of intervening scopes that may shadow an identifier, this is far more difficult. An extra pass is required, and error reporting is made more complicated.

Top-down parsing (integrated with scope analyses) detects semantic properties early and often, simplifying the construction of a good-quality compiler.

10.4 Surface Syntax Constraints

Lisp's surface syntax has some very convenient properties for supporting transformational and extensible grammars. The main advantage is that transformations are expressed as *tree* transformations, not transformations of *strings of tokens*. Any syntactically extensible language is likely to benefit from a similar two-level parsing strategy.

While a macro definition may appear to specify mappings from strings of tokens to strings of tokens, in fact it specifies a mapping from subtrees of the initial parse tree to subtrees of the transformed parse tree.

Recall that Lisp and Scheme, macro call syntax is always of the form (*macroname* ...). That is, *it specifies a kind of node in the phrase structure parse tree* (the input s-expression) of a particular form—a compound node (list node) with the macro name symbol in the operator position. The result of the macro transformation is also always an s-expression as well—a single subtree (s-expression) that replaces the input subtree. (The surface syntax of macro definitions ensures this.)

Lisp's initial parsing into phrase structure trees makes top-down parsing extremely simple and powerful. There is no need to look ahead a limited number of tokens during parsing, because the initial phrase-structure parsing ("reading") ensures that all of the tokens have been read and assigned their places in the initial parse tree. A macro transformer can "look ahead" by *subtrees* that represent phrases, rather than by *tokens*. That is, the programmer can write transformations over parse tree nodes, without worrying about the left-to-right phrase structure parse that constructs them.

This is much more natural for macro writers, who naturally think in terms of phrases, rather than sequences of tokens.

10.5 Phrase structure transformations vs. abstract syntax transformations

Given this understanding of macros as a transformational grammar facility, we can now begin to see what is deeply wrong with traditional macros—and what lexically-scoped macros are really "about," in linguistic terms.

Traditional Lisp macros are broken because they specify *surface syntax transformations*, while programmers generally think in terms of *abstract syntax transformations*. Lisp macros transform the phrase structure parse tree in ways that may not respect their abstract syntactic properties—properties that are usually intuitively obvious to programmers.

Typically, programmers think of transforming programs that consist of *meaningful expressions*, not sequences of tokens or even parse trees. The programmer's intent is usually to move expressions around without changing their meanings. Therefore, a deeper scope-resolving parse is required to fix the meanings of expressions before transformations can be safely applied.

Transformation rules are therefore not really transformations of (phrase structure) parse trees, but of a more sophisticated data structure (e.g., a DAG) that can encode scoping independently of the eventual position of expressions in the tree.

In some cases, however, the programmer's intent is to transform the meanings of expressions—or more precisely, to define meaning for expressions that otherwise would have no particular well-defined meaning.[30] The `for` macro illustrates this, because it implements the scope that gives the body expression a precise meaning.[31]

[30] Recall that we use "meaning" to include the referents of identifiers, e.g., what storage is referred to by a name, but *not* the states of mutable variables. In an imperative language, state-dependent aspects of the meanings of expressions are implicit and encoded procedurally.

[31] Readers familiar with Chomsky's work may rightly wonder what's going on here, with respect to the "autonomy of syn-

For these kinds of constructs, a lexically-scoped macro system provides a convenient language for writing transformations over otherwise indeterminate expressions, and fixing their meanings by successive transformations. A natural idiom (binding of identifiers passed as arguments to macros) allows the programmer to express sophisticated scope manipulations.

While this may all seem extremely weird when cast in linguistic jargon, it is astonishing to us how *easy* it is. Experienced Lisp programmers find it quite natural—except for the awkwardness due to Lisp macros' broken treatment of scope. With lexically-scoped macros, the major obstacle to programming with transformational syntax is eliminated.

10.6 Incremental Two-Level Parsing

10.7 Attributed Extensible Grammars

11 Example Uses

[Most of these are "systemsy" examples... need some examples of "applicationsy" uses, e.g., wrapping things for checking application-level properties.]

11.1 Implementing an Object System

[e.g., **TinyClos**]

tax" issue. One of Chomsky's major theoretical claims is that syntax is largely "autonomous," i.e., independent of meaning. (The classic example sentence is "Colorless green ideas sleep furiously." This sentence is perfectly grammatical, yet entirely nonsensical.)

While we won't comment on the controversial issue of autonomous syntax for natural language (except to say that there's clearly *something* right about the idea), we must part company with Chomsky here. Our goal is not to design a natural language like English or Urdu, of course, but rather the most "natural" programming language that is powerful enough for our purposes.

We deliberately keep the syntax close to the semantics, to keep the class of syntactically legal programs relatively small, and to allow early checking of semantic properties. In fact, one of the major points of this section is that traditional programming language) parsing techniques based on context-free grammars are *too* autonomous, because parsing is not really independent of other aspects of compilation. A compiler doesn't just recognize legal programs—it must fix their meaning and generate code that implements that meaning—so there is little point in pretending that grammar is separate from other aspects of program analysis.

11.2 New Scoping and Storage Classes

[e.g., **fluid variables, thread variables, run-static vs. persistent-static variables**]

11.3 An Object System With Encapsulation

[using first-class compile-time environments to define a new storage class, and implement its scope by inserting a new kind of compile-time environment frame in the chain...]

11.4 Closure Analysis

[decorating compile-time-environments during first compile pass, exploiting it during second]

11.5 Smarter Smart Pointers

11.6 Adding Garbage Collection to a Language

12 Toward an Exocompiler

12.1 Trading Off Range and Performance

12.2 Layering and Revealing Abstractions

12.3 Revealing More Compiler Phases

12.4 A Framework for Interleaved Analysis and Optimization Phases

13 Miscellaneous Issues

13.1 Surface Syntax

13.2 Type Systems

13.3 Compile Times, Link Times, and Run Times

14 Related Work

14.1 Macro Preprocessors

14.2 Scheme Macros

14.3 Dylan

14.4 Linguistic Reflection

14.5 Compiler Toolkits

14.6 Metaobject Protocols and Open Compilers

14.7 Partial Evaluation and Abstract Interpretation

14.8 UNCOLs

15 Conclusions

A Motivation (Part II)—Hidden Costs of a Lack of Extensibility

The consequences of *not* providing graceful syntactic extensibility are far more serious than is widely realized. We believe that much of the *current* tower of Babel of programming languages is precisely due to the *lack* of clean and cleanly extensible languages.

In this section, we explore several ubiquitous problems in software development which may at first appear unrelated. We argue that they all stem in part from a lack of extensibility in popular programming languages. These concerns motivate much of the rest of the paper.

A.1 Faking Syntactic Extensibility by Convention

It is common in languages without syntactic extensibility to implement libraries which are very large and crucially dependent on non-obvious constraints on the ordering of events. These constraints could be made obvious, or automatically enforced, by an equivalent library with a more appropriate language-level syntax.

A.1.1 Faking control structures.

An example of this is Microsoft's "critical section" abstraction used in the 32-bit Windows API. A so-called "critical section" is *not* actually a critical section, because critical sections are a *language feature*, not a library.

A real critical section is a *region of code* protected by a lock (specifically, a mutex).[32]

But since the Windows software is intended to be usable with the C language, and C is not gracefully

[32] Critical sections are used to ensure that potentially conflicting operations do not occur at the same time.

For example, one routine may modify a data structure, and during the process of modification, the data structure may be in an inconsistent state. Other routines should only examine the state of the structure *before* or *after* the modification routine runs, not while the modification is in progress.

Critical sections allow these unsafe regions of code to be marked, and associated with a "lock" that represents the right to access the data structure. At the beginning of the critical section, the lock is acquired, and at the end, it is released. If another a process has already acquired the lock, any other process must wait to for the lock before entering its critical section. The lock is called a *mutex*, because the critical sections operate with *mutual exclusion*—they can't be doing their ("mutually exclusive") operations at the same time.

extensible, the programmer is provided with a poor substitute: a library of data types and functions and a *set of guidelines* for using them in a way that achieves the same effect as critical sections. The programmer must remember that the proper use of a critical section is to *begin* a critical section with one function call, perform the operations that must occur with mutual exclusion, and then *end* the critical section with another function call. The programmer must get this right: there must be matching beginning and ending calls, and they must both operate on the appropriate lock:

```
 ...
BEGIN_CRITICAL_SECTION(the_appropriate_lock)
  ...
END_CRITICAL_SECTION(the_appropriate_lock)
 ...
```

We are convinced that it is simpler and less error-prone to provide a macro with a syntax similar to other control constructs, thus:

```
 ...
critical_section (the_appropriate_lock)
{
  ...
}
 ...
```

This syntax makes it clear what is really going on: a critical section *is* a control construct, so it should *look like* a control construct. With a sufficiently well-designed macro system, it should be possible to perform syntax checking so that simple mistakes of usage are detected automatically at compile time, and other mistakes simply can't happen. For example, failing to release the lock becomes a syntax error (due to a missing closing brace). Accidentally releasing a *different* lock at the end of a critical section simply doesn't happen, because the macro automatically releases the lock that was acquired. (Programmers also won't change one lock expression and forget to change another.)

This syntax can easily be recognized by a programmable pretty-printer, or an interactive source code indenter such as the one in the Emacs editor, or a syntax-directed editor: it is only necessary to program the pretty-printer to indent a critical section the same way as a while loop, by recognizing `critical_section` as a keyword that is similar to `while` for formatting purposes. Similar arguments apply to checkpointing and transaction constructs (e.g., `do_transaction () {...}` as opposed to `BEGIN_TRANSACTION() ... END_TRANSACTION()`) and many other kinds of simple control constructs.

This example is very simple, and perhaps the full significance of the point is not apparent. For more complex control structures, such as monitors, the advantages of clear syntax may be far more extreme. Achieving the effect of monitors by convention is more complex and error-prone, and yet so stereotyped that it is easy to provide automatically with a powerful macro system. [**The example of monitors will be discussed in Section 11...?**]

In short, control constructs should look like control constructs, and the correct association between control constructs and data that they operate on should be *as automatic as possible.*

A.1.2 Faking declarative constructs.

Just as real libraries end up providing control constructs disguised as procedural interfaces and coding conventions, they also often provide *declarative* constructs which appear as function calls. It is common for normal applications to preprocess some of their data, to reformat it and set up optimizations to be applied later.

A striking example of this is OpenGL, the most popular open standard "library" interface for interactive 3D graphics. OpenGL is really *a domain-specific language disguised as a library.*

OpenGL is based on a concept of objects, which typically represent 3-D objects (or components of complex objects) which may appear in a scene. OpenGL programmers use procedure calls to declare object types by calling C functions that pass descriptions of the objects. Later, the programmer can use the object types defined in that way, by calling different C procedures to create instances of those types, perform defined operations on them, and so on.

OpenGL thus has a data type definition facility as well as a procedural facility for manipulating instances of types. [**the procedural component is bad... too much mixing of structural and algorithmic stuff... simulation and rendering follow a simpleminded traversal of the object structure graph, and you have to code procedural parts with the knowledge of that fixed control structure...**]

The fact that OpenGL is implemented as a C library has serious consequences for programming style. The declarative features are rather awkward, because they are presented through a procedural interface.

Thus conceptually *compile-time* operations occur at *run time* in a C program—the program goes through a declaration phase where it declares objects by a series of function calls. This occurs at run-time with respect to C—but it is really a compilation phase with respect to OpenGL viewed as a language. Unfortunately, this means that normal static type checking of OpenGL type declarations cannot be done at (C) compile-time—it is deferred until application runtime, so errors may only be caught when a user attempts to run a program.[33]

An OpenGL declaration phase consists of a series of function calls that delimit and nest object definitions *in time* to achieve the effect of nested declarations. OpenGL requires that these calls occur in the proper order, so that it can infer nesting relationships. For example:

```
 ...
begin_class_definition(...)
   ...    /* description of 1st component */
  separator(...)
   ...    /* description of 2nd component */
  separator(...)
   ...    /* description of 3rd component */
end_class_definition(...)
 ...
```

(We have changed the names of the function calls for clarity.)

This code defines an object with three sub-parts, analogous to a class with three instance variables (fields or slots). The calls to the begin and end procedures tell OpenGL to start and stop accepting information that defines a type. The calls to the separator procedure tell OpenGL that a description of a subobject has ended and that the description of the next subobject has begun. At the end of the overall object definition, an OpenGL implementation is free to take the information presented by the procedure calls and construct an optimized representation of that object. For example, on some machines, an implementation might construct a binary space partitioning tree to represent a spatial decomposition of the whole object in a way that allows fast rendering without the need for a hardware depth buffer (z-buffer).[34]

Like any implementation of a a programming language, the OpenGL implementation may perform a sophisticated mapping from the "language level" declaration to a representation that is appropriate for an underlying platform—that is, it can include an optimizing compiler as well as a runtime execution environment.

We believe that declarations should *look like* declarations, and programming languages should make it easy to provide automatic checking for novel kinds of declarations.

Unfortunately, since OpenGL is cast as a library, rather than a language, many opportunities for early checking are missed. The C compiler cannot tell whether OpenGL class-definition function calls are properly ordered, just as it cannot tell whether a Microsoft "critical section" is properly delimited—it has no concept of such structure, and no declared syntax to check. If the programmer defines an object inappropriately, e.g., by failing to end the declaration with the correct function call, very strange things may happen at runtime. A simple syntactic declaration construct would turn this into a C compile-time error.[35]

[33] We are not arguing that *full* checking could be done statically for OpenGL, because OpenGL supports runtime creation of new object types. Thus at runtime there may be alternating "definition" and "execution" phases. This is very similar to what happens in interactive implementations of Lisp and functional programming languages, however, and those systems typically *do* use a compiler that can check many properties of expressions—and optimize on the basis of the results—at run time. What appears to the user as an interpreter is really an interactive compiler—expressions are compiled and executed at run-time rather than actually interpreted.

[blah blah... the object-oriented language Self...]

[blah blah... real systems often have several levels of nesting of phases, where the compile-time/run-time split is *relative to a given level of abstraction***...]**

[34] A z-buffer is a hardware or software mechanism for quickly performing occlusion computations during graphics rendering. The z-buffer ensures that if something is "behind" something that has already been drawn, it does not actually show up in the output. Z-buffering is common but problematic in two ways: it requires hardware support for fast operation, and it is not *scalable*—items are still drawn, even if they don't show up in the rendered picture. With hardware support, the constant factors can be reduced dramatically, but the scalability problem remains. Whether z-buffering is appropriate may therefore depend on whether hardware support is available, and on the complexity of the scene and the availability of higher-level algorithms to efficiently avoid rendering occluded items.

[35] There is an excellent argument for making OpenGL a library rather than a language feature: it can be used from languages other than C, via a simple procedure linkage convention. This does not preclude providing a clean syntactic wrapper within a given language, however, much as it is possible for monitor constructs in a language to use an underlying semaphore mechanism provided by the operating system.

The procedural nature of OpenGL definitions may provide useful power, at the expense of static checking. For example, a C procedure that defines a kind of object may conditionally call other C procedures to define subobjects, so that C computations

The lack of a clear compile-time/run-time distinction in OpenGL makes it difficult to perform some optimizations on the code that OpenGL executes at run time. A typical OpenGL implementation is really a combination of a compiler and an *interpretive virtual machine*—OpenGL compiles OpenGL declarations (inferred from the sequence of declaration calls) into an *intermediate form* which is *interpreted* by the OpenGL runtime system to simulate objects and render them.

With a clearer distinction between compile time and run time, it would be possible for an OpenGL to compile *different machine code* optimized for the needs of a particular application.[36]

(One way of doing this would be for OpenGL to compile to C, use a C compiler to generate machine code, and link the specialized machine into an OpenGL program. This does not preclude defining new types at OpenGL runtime—new types could be compiled to the usual interpretive form, or the C compiler could be called at runtime and the resulting machine code dynamically linked into a running applicaton.[37])

While OpenGL provides one of the clearest examples of a language disguised a library, we believe that this phenomenon is extremely common. Very many applications have a well-defined "initialization" phase, which may be viewed as a *compilation* phase at a higher level of abstraction—data structures are constructed to represent things in ways which will improve performance later in the execution of the "real work" phases of the program. (Often this is procedural information that is interpreted later.)

Unfortunately, the potential for checking and optimization in these initialization phases is often missed, due to lack of language support. A good syntactic extension facility would make it easier to check constraints and optimize representations early in the overall compile-initialize-run cycle. A syntactic extension facility therefore should provide "hooks" by which application-specific constraints can be detected and/or enforced.

It may seem that providing a sophisticated language model would be unnecessarily complicated—most application writers are not compiler experts, and are not fluent at coding compiler analyses and optimizations. For many purposes, however, the constraints and optimizations are obvious to the application programmer, because they are *application-level* properties and behaviors, not usually thought of in language terms.

For example, an OpenGL implemention might keep track of whether *any* object definition includes surfaces that require expensive kinds of rendering; if not, a fast but less general rendering method could be chosen. In this case, the analyses and optimizations are *graphics* issues, with which the programmer is quite familiar, not traditional "compiler" issues. One of our goals is to provide an easy-to-use framework by which applications can be optimized by programmers who are *not* compiler experts and never will be.

[OpenGL also has other severe problems that are really language problems, such as a lack of separation of the declarative and procedural components in rendering—it uses a fixed traversal strategy for complex objects, rendering their parts in an inflexible order that may not result in efficient rendering... This problem might not exist if OpenGL were designed *as a language* by people with a reasonable *language framework* to start from—one that made it easier to see the procedural/declarative distinction, and easier to implement and exploit it cleanly.]

A.2 Using "Little Languages"

While macro facilities are often decried as inviting a proliferation of incompatible dialects—and mutually uncomprehending programmers—we believe that this situation *already exists* precisely because conventional languages are too inflexible.

One of the clearer symptoms of this is the enormous proliferation of "little" programming languages, including shell programming languages, scripting languages, extension languages for particular applications

can control the construction of OpenGL types with significant variations. (In effect, C is used as a procedural macro preprocessor to customize type definitions fed to the OpenGL language preprocessor.) This power is not usually used, however, and providing a high-level syntax for the normal, stereotyped usage does not preclude providing a lower-level "unsafe" syntax that defers checking. This is analogous to the ability to provide both monitors (a relatively safe high-level construct) and semaphores (a more powerful low-level construct) in a concurrent programming language.

[36] For example, code that is optimized for the particular shape, texture, and reflectance properties of polygons that an object includes, or optimized occlusion and clipping code based on the geometric character of the objects in a scene.

[37] These strategies are used by several compilers we know of, including the Portable Cedar comiler [?] and our own RScheme compiler.

(often confusingly called "macro" languages[38]), configuration languages, data definition languages, and so on.

Consider a small sampling: the C shell, the Bourne shell, the Korn shell, tcsh, TCL, awk, sed, make, GNU config, REXX, Perl, Python, Icon, various spreadsheet "macro" languages, various word processor "macro" languages, various communication program scripting languages, AppleTalk, JavaScript, Visual BASIC, AutoLISP, various database extension programming languages, etc., etc., etc.

These are more-or-less application-specific languages designed for particular purposes, precisely because a conventional general-purpose languages are *too inconvenient* for these specialized purposes.

These languages are often well-designed in certain application-specific ways, including features that are convenient for stereotyped uses important to particular kinds of users. Unfortunately, many of them—but not all—are terrible programming languages in a general sense; they typically have poor control constructs, obscure syntax, weak data type definition facilities outside a narrow domain, little or no support for debugging, confusing error reporting, weak or inefficient memory management, and so on. Many are unnecessarily difficult to compile to fast machine code, or even decent code for an interpretive virtual machine.

The usual arguments for such languages is that they are "the right tool for the job"—whatever job they were designed for. Unfortunately, these languages are often used far beyond the original scope of application fro which they were designed, for several reasons.

Often, shell or scripting or extension languages are easy to use because they are *interactive* languages, based on a simple interpreters. Programmers find these languages much more convenient for program development than traditional compiled languages, and gravitate toward using them for larger and larger fractions of their applications.

These languages also end up being extended, because programmers generally would rather use only *a few* little languages, even if they believe in using "the right language for the job." There is a natural evolution toward not-so-little languages of broader scope.

[38] Some of these languages are actually just procedural languages without syntactic extensibility. Others are actually based on macroexpansion, generally of a primitive and error-prone sort

A.2.1 Tcl.

The Tk Command Language, Tcl (pronounced "tickle"), is an excellent example of this. Tcl was designed and developed by John Ousterhout as a scripting language that could provide flexibility to large C applications, especially those using the Tk graphics library. The general idea is that Tcl can be linked with a large application, to provide a small interpretive language that controls the execution of large amounts of compiled C code, typically in response to user commands from a graphical user interface.

[Actually, this makes Tcl a bad example in a certain way: it was designed to subsume a lot of little languages, by providing a framework for turning any application into an application-specific language that looks like Tcl... unfortunately, it's the wrong language for that... on the other hand, it's special-purpose in the sense that it was originally mostly a "user interface" programming languae... clarify]

Putting a small interpreter in control, to glue together modules of application code, gains great flexibility—in effect, the program becomes an application-specific programming environment, where pieces can be flexibly combined in an interactively changeable way. This is especially appealing for developing interactive programs' user interfaces.

Originally, Tcl programs were expected to be very small—little "scripts" of a few lines, or a few dozen lines, or at the most a few hundred lines. It was assumed that the large majority of a sophisticated program would be written in C, and only a little "glue" would be written in Tcl. Essentially, Tcl would function as a programmable dispatcher to large volumes of C code. Because of this intention, Tcl was not designed to be fast, or to be good for "programming in the large."

It did not work out as expected. As Ousterhout reports [Ous94], his expectation was quite wrong: programmers quickly found that developing code in the interactive Tcl environment was far more appealing than writing C code and dealing with the usual edit-compile-link-run cycle. Many programmers took to writing entire programs in Tcl (many using the Tk library as well), including many relatively large programs. In fact, *most* Tcl programs are written *entirely* in Tcl, many of them thousands of lines long, and a number of them tens of thousands of lines long. Because of this, the need for more language features became apparent: e.g., a module system for organizing large programs, an object system for organizing data type definitions, and so on.

Unsurprisingly, Tcl is not a pretty language. In our view (shared by many others, e.g., Richard Stallman [?]), Tcl is a bad programming language. Its core syntax and semantics are very arcane, exhibiting problems in areas such as scoping, typing, and storage management that were solved decades ago for languages such as Lisp and Smalltalk.[39]

Tcl is extremely slow—typically hundreds or thousands of times slower than C, and at least dozens of times slower than a fairly good but straightforward Scheme interpreter—because it was not designed from the start to lend itself to fast implementations. It was assumed that the "glue" provided by Tcl would be used in a coarse-grained way (e.g., interpreting responses to user button-clicks) and that the vast majority of the "real work" would be done in optimized C code.

Stallman has argued that the problem with Tcl and many other special-purpose languages is that *they need to have basic general-purpose features* to be really useful [?], and that *they should be based on generally good languages.*

We emphatically agree with this assessment—there is no need for the current tower of Babel, because most of the idiosyncratic differences between the zillion "little languages" are precisely that—idiosyncratic differences. This is *not* to say that most of these languages don't include very nice features that make them particularly appropriate for their application domain—most of them do. It is only to say that there could be one or a very few unifying *base* languages to which such extensions could be added.

[39] Some examples:

- supporting general heap allocation using garbage collection [McC60, Wil],
- eliminating the distinction between expressions and commands to avoid unnecessarily complex syntax [McC60],
- good default rules for expression evaluation and quoting so that syntax can be regular yet still powerful,
- efficient syntactic extensibility through macros rather than dynamic calls to a run-time command interpreter (evaluator) [?],
- compiling a lexically-scoped "interpretive" language to an interpretive virtual machine code for high portability with good performance [?],
- ensuring that such a language can be compiled to good machine code when performance is critical, [Ste78, KKR+86, Kra88, Kel], etc.

(Not to mention lexically-scoped macros.)

Stallman's particular recommended language is Scheme—a small, simple, easy-to-implement, and very clean dialect of Lisp. We agree that Scheme is a very nice programming language for most purposes, and that it would be a good choice as a base for many small application-specific variants. That is not our point here, however; we do not wish to argue for Scheme in particular (though we would be happy to elsewhere). The point is really that *any good language will do*, if augmented with appropriate mechanisms for extensibility—but most of Babel is occupied by languages that are *bad* to some significant degree, due to basic mistakes in programming language design.

A.2.2 Make, Spreadsheets, and Constraint Satisfaction.

Make. The UNIX utility `make` (now available on many operating systems) is an interesting example of a little programming language, even though many users don't recognize it as a programming language at all. In its most common usage, `make` keeps track of dependencies between files used to generate an executable program, recompiling changed files and relinking applications as necessary.

Make is more general than this, however. It is a simple example of a *constraint programming language*, where the user mostly provides declarative specifications of constraints that must be satisfied (e.g., that a new version of a program requires new object code versions of any changed source files) and simple procedures for satisfying those constraints (e.g., that a new version of a module can be created by recompiling the source file).

Because it is really a relatively general (if simple) constraint satisfaction system, make can be used for many purposes involving simple constraints and actions. For example, we have used it to automate the running of simulation experiments that require multiple input data sets, multiple stages of processing with multiple intermediate representations, and the generation of output in multiple graphical formats [WJNB95]. (From `make`'s point of view, input data is equivalent to a program, and data processing steps are equivalent to compilation and linkage.)

Rather than being based on a pre-existing programming language, `make` was written from scratch, in C, as a one-off utility program. Though features for more sophisticated programming have been added over the years, `make` shows its heritage in its arcane syntax[40] and its limited and irregular support for general-purpose computation.

We believe that the limitations, irregularities, and general ugliness of `make` could easily have been avoided if make were programmed as an extension to a simple general-purpose programming language. A language providing clean syntactic extensibility should allow the easy definition of a clean syntax for dependency declarations. It would be easy to provide hooks for general-purpose procedural programming that would make `make` more generally and easily useful—e.g., to determine an appropriate procedure for satisfying a constraint in an application-specific way.

Spreadsheets. Spreadsheets are another good example of simple constraint-satisfaction programming systems. The "user" (programmer) declares constraints between data objects (variables) arrayed in two dimensions, by defining values in terms of other values (e.g., "the difference between cell A6 and cell B12" or "the sum of items in column B").

Spreadsheets are a very simple variety of constraint satisfaction system, applicable to certain very common kinds of problems in bookkeeping and data analysis. They constrain the ways in which the user can state problems, so that a very simple constraint satisfaction algorithm can efficiently compute answers.

Unfortunately, most spreadsheets have very weak, confusing, and inconsistent language features, which have little or nothing to do with their basic function. For example, the most popular spreadsheet program has a feature for giving intuitive names to cells in the array of variables. Unfortunately, these names are just "nicknames" for specific cells in the array, rather than logical identifiers of objects that play particular roles in a computation. As a result, a spreadsheet can be modified by moving things around, and the nickname will *refer to the wrong variable.* A mechanism for automatically updating dependencies between the values is available, but it works for some things and not others—it must be used carefully, and mistakes are often made.

Constraint Satisfaction (sub-)Languages. Constraint languages are interesting in that they may use a general-purpose language to good effect—in their internal implementation and/or as an "escape." (To application-specific procedures for hard problems that can't be solved by a general-purpose constraint satisfaction strategy. **[give example, and or citations.]**)

Such systems may need to *restrict* the availability of language features to make them easy and safe to use the declarative features. In general, a constraint language must prohibit the execution of code that may undermine the constraint satisfaction strategy—or at least provide a simple model to the user so that the user can avoid the problem. For example, if executing a constraint-satisfaction procedure undoes a previously satisfied constraint, a simple constraint solver may either fail to properly solve the problem, or be vulnerable to infinite loops or strange performance problems. (In a typical use of `make`, a compiler that modified its source files could be problematic. In a spreadsheet, uncontrolled side-effects to data may undermine the "natural order recalc" strategy used to resatisfy constraints efficiently)

For some applications, therefore, an extensible language should be *limitable*—it should be possible to *disable* features of the base language that may conflict with high-level extensions that have been added. If these features correspond directly to syntactic constructs, a straightforward approach is to provide a new syntax definition that simply signals an informative error.

Scope control is also important—the user should not *unnecessarily* be prevented from using powerful procedural features within the overall declarative context. **[A programming language should make it easy to hide what needs to be hidden, and reveal what is most useful to reveal.]**

These are fundamental linguistic issues, which are best addressed via linguistic mechanisms. Otherwise, programmers of "applications" such as spreadsheets will reinvent the wheels of programming languages, and *often get them wrong.*

Latex. LaTeX is another widely-used application that may be viewed as a programming language—a programming language for formatting documents.

[LaTeX is implemented as TeX macros. TeX macros aren't great...]

[blah blah...]

[Bad scope rules—TeX procedures that aren't in the LaTeX manual are visible to LaTeX programs by default, easy to make weird

[40] For example, different kinds of whitespace mean different things semantically. The use of multiple spaces where a tab is expected is an error (!).

mistakes. Could hide them by default, but provide access only where useful, or by escape mechanism.]

[Poor syntax and weak/late checking generally, making it easy to screw up and get totally confusing error messages]

Discussion.

A.2.3 Design rule checking

Many kinds of applications (and languages) require some form of *design rule checking*, to ensure that combinations of features will combine correctly. This is similar to the notion of type checking at the language level, but the properties checked may be more subtle than the usual type conformance properties dealt with by type systems.

For a particular application (or extended language), the appropriate algorithms may be quite different from the algorithms used by a language's built-in type checker. For example, a software component may need to match in terms of fundamental behavior or performance properties as well as having its inputs and outputs match their declared types.

A very simple example of this is that a collection data type may only work in a particular application if it is stable under sorting. (That is, if multiple occurrences of the same key value occur in a collection, sorting the collection will not change their relative order.) In a CAD (Computer Aided Design) application, a circuit represented by a software component may only be appropriate if it can be implemented in the process technology being used for the overall design.

Design rule checkers are commonly implemented as separate, standalone programs, even if the items being checked are represented by normal data types in a programming language—a description of the type is maintained in parallel to the language type description, and perhaps written to a file or a database. Then a separate program reads the descriptions of components and the overall design, and checks whether the design meets the necessary constraints.

The UNIX utility `lint` is a simple example of a design rule checker. Early C compilers left out important syntactic checks, so a separate program was devised to check the syntax of C programs. This program also did additional checking, flagging legal but suspicious-looking constructs as likely errors—helping find the fluff or "lint" that should be removed to tighten up the program.

Problems with standalone checkers. The problem with `lint` and with standalone checkers for higher-level properties is that the lack of integration with the programming language makes them hard to implement, hard to maintain, and hard to use.

They are hard to implement, because they must often replicate work done by the compiler—they must be able to read some syntax describing kinds of components (data types, code, or whatever), and interpret it; this is essentially what a compiler does. If a compiler were extensible, it might be possible to program the compiler to perform the additional parsing and checks, only specifying the *new* parts of its task.

Standalone checkers are hard to maintain, as well. When a syntactic property of the system changes, the main system must be changed—and so must the checker, *even if the changed property has nothing directly to do with the checker's job.* For example, a slight syntactic variant of a control construct can confuse `lint`; `lint` must be able to parse the whole C language to reliably identify the parts of a program that it must check. `lint` therefore is essentially a entire front end of a C compiler, augmented with extra error checking features.

Standalone design-rule checkers for higher-level properties also tend to evolve in this direction. To provide coherent feedback to the user, they must take on more and more of the functionality of the main system; to stay compatible with the main system, they must be changed as the main system changes. Even trivial superficial changes to the main system can break the checker, whether or not it matters to the checker's task. For example, simply changing the identifier used for a keyword can break a checker whose task is is checking modules' semantic compatibility.

For these reasons, design rule checkers are often problematic. They often require major effort to implement and maintain. Worse, the are often broken or just too inconvenient to use.[41] We believe that this contributes to the poor quality of much current software. Correctness and reasonableness checks aren't done often enough, or aren't done at all, because they are *too hard to automate.*

[41] Witness the many lingering bugs in UNIX software [?] due to infrequent use of `lint`—programmers simply didn't use it enough. Sometimes they didn't use it because it was broken—e.g., not tracking changes to the evolving C language, or vendor-specific extensions. Sometimes they didn't use it because it wasn't a natural part of the programming process for overconfident and hurried programmers—i.e., most of them.

A.2.4 Query Optimization

Domain-specific languages and applications often use more-or-less declarative features, and rely on those delarative features to give them the flexiblity to optimize operations. In this respect they resemble more general-purpose declarative programming systems, such as Prolog or relational database systems. In fact, many programs are best viewed as specialized database programs.

In a high-performance domain-specific database system, for example, it may be desirable to incorporate features of relational databases. Large collections of data may be implemented as data structures that implement relations (tables), and *queries* over those relations may be expressed by the user/programmer and optimized by the system. For this to work, there must usually be a restricted set of operators that apply to the information processed by the query optimizer.

This allows the optimizer to analyze expressions and determine their important semantic properties, and then transform the query into an efficient procedure for getting the answer.

For example, if a database holds genealogical data, a user might ask the system to find all (known) people who were born after 1960, and then from the resulting set select all of those whose parents were born before 1930. The query optimizer may choose to get the result in a different way, reversing the order of the operations: it may first search for people born before 1930 and then filter the resulting set to find the ones with children born after 1960.[42]

For a query optimizer to be able to do this, it must be able to tell that rephrasing the query for more efficient execution *will not change the result.* (For example, that searching for the records of young people will not change the ages of older people.) In a general-purpose programming environment, this is not always possible—the code for one kind of query might interfere with the code for another, and the query optimizer might generate an incorrect procedure.

For example, in a domain-specific C++ program, a specialized search method might execute methods (arbitrary procedures associated with objects) that indirectly modify the state of *other* objects; in the general case, there is no way to optimize queries because the different steps of a search may have interacting side-effects—doing them in one order is not always equivalent to doing them in another order.

To enable query optimization, it may be desirable to restrict the language. For example, changing the "birth date" field of an object that represents a person may be illegal, except when done in ways that the query optimizer "understands"—e.g., by asking the query optimizer to do it with a "change birth date" query. (This might seem like an odd request, but sometimes databases contain mistakes that have to be corrected. **[Actually, I should probably change the example.]**)

To support elegant query processing, therefore, it should be possible for the programmer to constraint the actions that can be done procedurally by "arbitrary" (handwritten) procedural code.

For this example, the programmer should be able to add a new kind of declaration, "only modifiable through query optimizer" that can be used for values the query optimizer depends on, and somehow modify the compiler to disallow changes to such fields in code that is not part of a query.

Without the ability to do this, programmers often end up *not* using the most appropriate and general technology (e.g., query optimization) applicable to their programs—they pick a language and do what is most natural within that language's (limited) paradigm. If they do try to cast the problem in the most natural way, they get little support from the underlying language. For example, a query optimizer may only work properly if programmers obey certain conventions—e.g., commenting the fields that the query optimizer depends on, and respecting those comments by not changing them inappropriately.

A.3 Implementing Extensions with Preprocessors and Precompilers.

Unlike the implementors of most "little languages," many implementors of novel language features do *not* attempt to design an entire language from scratch. Some recognize that it is quite difficult to design a new general-purpose programming language *well*—there are too many issues involved, and most people know they don't have the expertise to do it. Others simply do not have the time to do a good job of designing an entire language, even if they believe (rightly or wrongly) that they would be competent to undertake

[42] It may do this because it "knows" that this procedure costs less, perhaps because it keeps statistics about the distribution of key values. Filtering on the second criterion first may thus produce a small set of candidates quickly, reducing the cost of the second filter. On the other hand, it may choose the opposite order, for example if it has a fast index of people born after 1950.

such a difficult project.

These people often design and implement useful *extensions* to existing languages. Researchers may do it to prove a point that it independent of the details of the language chosen, or because they think that a given language suits their purposes just fine except for a certain missing feature. Commercial developers may do for the same reasons, or because they must maintain compatibility with an existing language in wide use.

Some of these people design and implement entire new implementations of their chosen base language, just so that they can add a new feature or two. This is usually a bad idea.

Many others choose a simpler route, building a *preprocessor* or *precompiler*, which transforms a program in their extended language into a program in the base language. A straightforward preprocessor simply reads a file of source code in the extended language, rewrites the extended constructs into equivalent statements in the base language, and outputs a program in the base language.

Currently, C is a popular base language, because good implementations exist on almost every computing platform. C++ was originally implemented as a precompiler, which translated C++ constructs into C constructs, producing a C program that could be compiled with a C compiler.

C++ is now usually implemented as an integrated compiler, rather than by precompiling to C. But as good C++ compilers have become available, some people have chosen to build on that extended language, and precompile extended versions of C++ into C++.

A.3.1 Preprocessors.

Simple language translators are often called *preprocessors*. A preprocessor is typically a fairly small program, which recognizes only a few constructs and transforms them appropriately. The appeal of this approach is that a small change to the language should only require a small effort to build the preprocessor.

Unfortunately, this convenience is often less real than it appears at first, which accounts for the relative scarcity of widely-used programming languages based on simple preprocessing. (We exclude simple macro preprocessors like the C preprocessor or the Lisp macro facility—they serve a different purpose, which is discussed in the main body of the paper.)

Problems in implementing and maintaining preprocessors result in most preprocessors being junked once the user base for a language is large enough to support the development of a decent standalone compiler.

The reason simple preprocessors don't usually last long is that they are generally hard to use and inflexible. (The problem is not just that an extra program is required to do the preprocessing—preprocessing usually takes a small amount of time relative to the subsequent compilation, and the steps can easily be automated using a small script program that runs the preprocessor and the compiler.)

One problem is that a simple preprocessor does not "understand" the full language, and may get confused by subtle constructs in the source language that are *not* the province of the preprocessor. It must be able to pass those constructs on unmodified, to be dealt with by the compiler. Some preprocessors therefore impose an awkward syntax on the constructs managed by the preprocessor, so that they can be easily distinguished from the "rest" of the language.

Another problem is that a preprocessor may not just be faced with *legal* programs in the extended language—it may be faced with *buggy* programs. An erroneous program may be handled well by the compiler, which has a sophisticated parser that can diagnose errors and yield useful error messages. A simple preprocessor cannot easily do this, because it doesn't understand the full syntax of the language.

Even when the preprocessor does detect syntactic errors, runtime errors may be harder to diagnose because the preprocessor changes the mappings between the source language and machine code—"source-level" debuggers may not be able to map the preprocessed constructs back to the source the user actually wrote.

A.3.2 Precompilers.

Rather than using simple preprocessors, some language implementors implement *precompilers*; a precompiler "understands" the entire extended language—including base language constructs—well enough to perform relatively complete syntactic checking and often static semantic checking. A precompiler *is* a compiler, whose output happens to be a legal program in the base language.

Precompilers are much easier and safer to use than simple preprocessors, because they can give complete and coherent error checking, like any other compiler. They may also perform some sophisticated analyses and optimizations, and generate better code.

A major problem with precompilers is that they lose much of the benefit of the preprocessing approach. Like compilers, they are difficult to develop and maintain; they typically replicate most of the functionality of the front end of the normal base-language compiler, and perhaps some of the back end functionality. They require a large amount of expertise in compiler construction.

Another is that the resulting precompiler is likely to be incompatible with other preprocessors and precompilers, limiting its usefulness, as we describe next.

A.3.3 Mutual Incompatibility of Preprocessors/Precompilers.

The deep problem is that *preprocessors (or precompilers) are often mutually incompatible.* It may seem simple to simply "pipeline" preprocessors, using one to translate certain features, and sending its output to another that translates others, until all of the constructs have been "flattened" into equivalent operations in the base language. Unfortunately, it often doesn't work, especially if constructs may be *nested.*

In the case of nested constructs, the preprocessor that implements the *outer* construct should be invoked first, to flatten it out into the base language, and reveal the nested inner construct, handled by a later preprocessor. But if constructs may be arbitrarily nested—as is usually desirable—there is *no fixed order of preprocessing that will do this.* In fact, there is not even a *fixed number of times* that each preprocessor must be run: the program must be run through each of the preprocessors repeatedly, so that layers of constructs can be flattened as the appear.

For extensions to be even syntactically compatible, a *recursive framework, similar to a parser* is needed. This is what a macroexpansion facility provides, though most macro facilities have other limitations that make them impractical for implementing many language constructs.

A.4 One-off Hacking of Linguistic Features from Scratch

Large software development efforts often involve the development of a large number of tiny languages. **[blah blah...]**

readers/parsers for records,
picklers,
testing utilities that save and replay input events,
script-generating programs
code-conversion scripts,
data-conversion scripts,
etc.

A.5 The problem with Yacc

A.6 Discussion

[**....the current mess**] reflects a major failure on the part of the programming languages community. While language designers have been designing general-purpose languages, the barbarian hordes of *people with real work to do* have been designing, implementing, and spreading an infectious cloud of programming languages that have taken over much of the real world.

Much of the very real and hard-won expertise in the programming language community is going to waste, because programming language designers have missed an important point: *the real world requires application-specific languages.* Once that crucial point is grasped, the obvious question is "how do you get an application-specific programming language that *isn't a bad language*?"

Part of the answer (as Stallman correctly says) is probably to *start with a good language*—a good, small but general-purpose programming language, and to *make sure that it is cleanly and powerfully extensible.*

[blah blah...]

B A Brief Introduction to Scheme

B.1 Some Myths about Lisp (and Scheme)

For anyone who has never used a modern Lisp, and may have been misled by popular descriptions, we will briefly dispense with some myths about Lisp (and Scheme) which may interfere with appreciation Lisp's features. (Some of these myths are propagated by textbooks still in use, despite having been debunked many years or decades ago.)

None of these points bears directly on our arguments about extensibility; the reader is invited to skip or skim them if uninterested, and perhaps return to this section if any of the examples are confusing.

B.1.1 Myth: Lisp has only one structured data type.

It is often said that Lisp only has two data types: lists and atoms, where lists are made up of heap-allocated *pairs*. (Pairs are often called *cons cells*, and are simple structures containing two pointers.) Lists are indeed important in Lisp, but modern Lisps support the usual complement of data types, such as arrays, records, and character strings. Most Lisp systems also include object systems with inheritance, for object-oriented programming (e.g., CLOS, the Common Lisp Object System).

B.1.2 Myth: Lisp is always interpreted.

Good implementations of modern Lisps use sophisticated compilers, which generate native code similar to that generated by compilers for other languages. (An interpreter may also be available, for more flexible interactive use, but interactive compilers are also quite flexible.)

Macroexpansion is an integral part of the implementation of a good Lisp system, allowing code transformations *at compile time*.

B.1.3 Myth: in Lisp, code is data.

It is widely believed that in Lisp, code and data are interchangeable—code is just lists, and lists are interpreted by the `eval` function.

There is some truth to this—Lisp code is generally represented as lists called *s-expressions*, *at compile time*, but Lisp systems typically do not interpret s-expressions most of the time. They *compile* code to create executable procedures, which are *not* lists. Even interactive systems often use compilers rather than interpreters.

Most Lisp systems *do* include an `eval` function to execute code represented as lists. `eval` may interpret s-expressions, or it may get the same effect by compiling s-expressions and executing the result.

[blah blah...]

The ability to evaluate code at runtime *can* be useful, but should not be used casually. For example, it can be used to create self-modifying programs, but in general, self-modifying code is dangerous. In fact, modern Lisp code is almost never literally self-modifying. *New* procedures may be compiled and linked into the system in place of old code, but it is very unusual to modify s-expressions and interpret the modified version; in some systems, it's not allowed.

In general, most of the seemingly clever uses of the `eval` procedure have been replaced by more disciplined mechanisms, such as the use of macros. A macro facility is less powerful than `eval`, but that's usually a good thing: it's powerful enough to do amazing things, such as implementing control constructs less error prone. It is also typically far more efficient.

[blah blah...]

There are some good, disciplined uses of `eval`, but they are usually hidden in low-level code, and not part of normal programming.

For example, the basic interaction loop in an interactive system generally uses eval to interpret (or compile and run) expressions typed by the user. Normal interactive program development uses this feature, but normal programs do not usually call `eval` directly.

B.1.4 Myth: Lisp is slow.

It is widely believed that Lisp is interpreted and therefore many times slower than Algol-like languages. Neither is true. It is a matter of some controversy whether Lisp is slower on average than other general-purpose languages. We will not get into that debate here, because it is a controversial point that doesn't bear on our points in this paper.

B.1.5 Myth: Lisp has trivial syntax.

Lisp is often described as having trivial syntax, because code consists of fully parenthesized prefix expressions—rather than the usual mix of statement syntax, prefix function call syntax, and infix operator syntax.

While the regularity of Lisp's prefix syntax is significant, it is only part of the story. Within that *surface syntax* framework, there is a deeper notion of syntax. A given kind of expression often expects particular subexpressions in particular positions, and may treat different subexpressions differently—much like the syntactic variations between different syntaxes in other languages.

The difference is that Lisp has a *two-level* syntax, whose surface regularity makes it easier to automate the processing of deeper syntax. In particular, it makes it easy to write macros that extend the deeper syntax, and which perform syntax checks. A similar approach can be used with a more conventional syntax, allowing automatic generation of parsing routines for an extensible syntax.

B.2 Dynamic typing

Lisp and Scheme are dynamically typed—no type declarations are necessary; operations check their argument types automatically at run time.

This point is mostly orthogonal to the issues of scoping and macroexpansion that we are addressing, but not entirely. Dynamic typing makes it easy to construct s-expressions that are nested arbitrarily, despite having constituents of varying types. This, in turn, is quite convenient for writing code transformation procedures.

Any type system for an extensible language should be able to represent expressions in the language; this may favor at least limited dynamic typing, or a very expressive static type system.

B.3 All Values are Pointers to Objects

In Lisp and Scheme, almost all values are conceptually pointers to data objects on a garbage-collected heap. Implementations often optimize representations to avoid actual indirections in many cases, but for a basic understanding of Scheme, it is best to pretend that all values are pointers to objects.

This conceptual regularity makes Scheme code clean and simple; since everything's a pointer, operations that operate on the referents automatically dereference the pointers if necessary. No explicit pointer syntax is necessary to distinguish between pointers and nonpointers. Operations that need to dereference pointers do so automatically.

For example, the addition procedure `+` takes pointers to numbers, dereferences them, adds the integers, and returns a pointer to the result. (Conceptually, that is. The actual representations of numbers are often optimized.)

Operations that pass values *copy the pointers*, not the objects they refer to; parameter passing is always *by value*, not by reference, but the values are pointers; this is not the same thing as pass-by-reference, and is much less treacherous.[43]

(This is how many clean modern languages work, e.g., ML and most functional languages, CLU, Smalltalk and Self. Many other languages work this way for the most part, including Eiffel and Java.[44])

B.4 Parenthesized Prefix Expressions

Scheme code consists of (possibly nested) prefix expressions, written in a parenthesized prefix form. For example, the arithmetic expression `((a + b) * c)` is written `(+ (+ a b) c)`.

This may seem unintuitive, but it allows operators to take any number of arguments (e.g., you can write `(+ a b c d)`, rather than `(a + b + c + d)`, because `+` can take any number of arguments).

Procedure calls have the same syntax. E.g., calling the procedure `foo` with the argument `bar` looks like this: `(foo bar)`, rather than the conventional (Algol-like) `foo(bar)`. (Note that the procedure name goes inside the parentheses, with the arguments, rather than ahead of the left parenthesis.[45])

B.5 Operators are Procedures

In fact, in Scheme there is no distinction between operators (like `+`) and procedures (like `foo`). (Implementations may optimize away the procedure-calling cost

[43] **[blah blah... pass-by-reference would pass pointers to pointer variables. Pass by value passes contents of fields, which happen to be pointers values. You can't write `swap` as a procedure, for example, because it only gets copies of the pointers in variable bindings, not the addresses of the variable bindings themselves.]**

[44] It is also the recommended coding style for many situations in C++, since uniformly passing pointers by value eliminates many issues of when data structures should or should not be copied, because it avoids **[hassles with weird copy constructor issues, slicing problem]. [cite Stroustrup, blah blah...]**

[45] This is unusual for general-purpose programming languages, but it may seem less unusual if you regard it as being like *commands* in an operating system (e.g., `rm foo` or `del foo.txt`) with parentheses to delimit them for nesting purposes.

for common operators in most cases; that's not important here.)

Scheme's syntax is thus *more* intuitive than conventional syntax in the sense that a user-defined procedure can be used as an expression, in just the same way as a built-in operator.[46]

B.6 Procedures are First-Class

In Scheme, executable procedures are data objects that happen to have the property that you can call them as procedures. So, for example, the normal addition procedure is an object, and you can store pointers to that object in data structures, pass them as arguments to other procedures, etc. [**compare contrast function pointers in Pascal or C; define "first class"**]

Scheme also has a *unified namespace*, meaning that there is not a distinction between procedure names and variable names. A procedure variable is just a variable whose value happens to be (a pointer to) a procedure. The symbol `+` is just the name of a predefined variable, whose value happens to be the standard addition procedure. (This value can be changed by assigning a different value to the variable `+`. For example, the addition function can be *wrapped* by replacing it with another function but also performs auxiliary activities, like counting the number of times it is called, for profiling purposes.)

B.7 Statements are Expressions

In Scheme, there is no distinction between statements and expressions. Statements are expressions—some expressions change control flow or have side effects, but they're still expressions.

Control constructs also return values, which may or may not be useful in a particular situation. Often it is, and it allows the elimination of the `return` construct found in languages like C.

B.7.1 Some Control Constructs.

[IF, BEGIN, procedure call]

B.7.2 Variable Binding Constructs.

[define, let]

B.7.3 Procedure Definition.

[define, lambda]

B.7.4 Other "Special Forms."

backquote and comma, at least. Wave hands about others...

B.8 Basic Expression Evaluation

B.8.1 Literals.

B.8.2 Identifiers.

B.8.3 Procedure Calls.

B.8.4 Macros.

[macros aren't first class—you can take a pointer to a macro in the usual way, because it does its work at compile time, not run time.]

B.9 Some Example Scheme Code

[demonstrate why you don't need return statements, among other things.]

[46] Contrast this to C's lack of an exponentiation operator—a user-defined operation has a syntax (e.g., `expt(x)`) that is inconsistent with the built-in operator syntax, for no particular reason. Operator overloading (as in C++ and Ada) can help with this, but it does not address the unnecessary syntactic difference between binary operators and operators of higher arity.

References

[ASU88] Aho, Sethi, and Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[Bro74] B. M. Brosgol. Deterministic translation grammars. 1974.

[Cho56] Noam Chomsky. Three models for the description of language. *I.R.E. Transactions on Information Theory*, IT-2, September 1956.

[Cho57] Noam Chomsky. *Syntactic Structures.* Mouton, The Hague., 1957.

[Ing61] P. Z. Ingerman. Thunks. *Communications of the ACM*, 4(1):55–58, January 1961.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[Kel] Richard Kelsey. *Implementing Programming Languages by Transformation.* PhD thesis.

[KKR+86] David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *SIGPLAN Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. Published as *ACM SIGPLAN Notices 21*(7), July 1986.

[KL] Gregor Kiczales and John Lamping. Toward a new model for the engineering of software.

[Kra88] David A. Kranz. *ORBIT: An Optimizing Compiler For Scheme.* PhD thesis, Yale University, New Haven, Connecticut, February 1988.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, Reading, Massachusetts, 1994.

[Ste78] Guy L. Steele Jr. RABBIT: A compiler for SCHEME. Technical report, Massachusetts Institute of Technology, May 1978.

[Wil] Paul R. Wilson. Garbage collection. *Computing Surveys.* Expanded version of [Wil92]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps.` In revision, to appear.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer Verlag.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.