# Semantically-Sensitive Macroprocessing[*]

William Maddox

December 15, 1989

**Abstract**

Conventional procedure and type definition mechanisms are not sufficiently powerful to express many programming abstractions that can be captured by syntactic transformations. Unfortunately, conventional macroprocessing is oblivious to the semantics of the base language, resulting in scoping anomalies, poor handling of static semantic errors, and an inability to perform transformations dependent on semantic attributes of the manipulated program. We introduce a new mechanism, *semantic macros*, which permit such transformations a significant level of access to the static semantic properties of the program fragments they manipulate. In this way, new static semantic processing, including compilation of embedded languages with a rich static semantics of their own, can be incorporated into user-defined language extensions. A proof-of-concept language, XL, is described which embodies this mechanism.

[*]Sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292, and by IBM under IBM Research Contract No. 564516. William Maddox was supported in part by an AT&T Foundation Graduate Scholarship.

## 1 Introduction

Nearly every programming language provides a limited procedural abstraction facility in the form of user-defined subroutines. Most modern languages provide for user-defined data types as well. Yet these definitional mechanisms are too weak to admit many kinds of definitions analogous to the built-in language constructs, such as new declarative forms and control structures. This report proposes that this situation be remedied by allowing the user access to the same mechanism used by the language translator: computation over the syntactic domain of the language at translation time. We propose, however, that such a mechanism be *embedded* in the language as a part of the language itself, not an add-on implemented by the extralinguistic expedient of translator modification.

In Section 2, we critique the existing definitional facilities found in the so called "data abstraction languages," observing that they fail to support the full range of definitions legitimately called data abstractions, as well as ignoring most forms of control abstraction. Section 3 proposes a response, in broad terms, to these shortcomings. These ideas are developed further in Section 4, in which the requirements for an improved definitional facility are explored. Section 5 takes a brief look at Lisp, whose macro facility comes closest in current practice to satisfying our requirements. By examining the shortcomings of Lisp, we are led in Section 6 to a new formulation of a macro-like definitional mechanism in which static semantics play a central role. Section 7, the main body of this document, describes in detail the language XL, an experimental language designed to explore the notion of semantically-sensitive macroprocessing. In Section 8 we present four illustrative and non-trivial examples of semantic macros in use. Section 9 examines some of the choices made in the design of XL, pointing out alternative approaches worthy of further investigation. Section 10 briefly describes the prototype implementation of XL. The report concludes with a comparision of XL with related work, issues for future research, and conclusions, in Sections 11, 12, and 13.

## 2 What is wrong with conventional ADTs

While promising to support abstractions of a more general nature than might be classified as data abstractions, our work is motivated historically by several deficiencies we observed in languages that claimed to support extension via user-defined abstract data types (ADTs). It seemed that the previous round of research in so-called "data abstraction languages" [15][23] had left some unfinished business:

1. Heterogeneous (record-like) type constructors cannot be defined. Particular types with a fixed set of fields of fixed name and type can be expressed, but

there is no way to capture the commonality among all such types. Nothing akin to the Pascal `record...end` construct, which abstracts and encapsulates the very notion of "recordness," is expressible in terms of the definitional facilities provided. Such abstractions are parameterized with respect to entities that are not normally manipulable in data abstraction languages, e.g. field labels.

2. Operations on instances of ADTs are restricted to procedures and functions. Declarative forms and control structures specific to a user-defined type cannot be expressed.[1] We might want, for example, a special binding form that wraps initialization and finalization actions around its body or a type consisting of Prolog-like terms with a pattern-matching control structure.

3. Parameterization of ADTs consists of filling in holes in a "canned" code template. The programmer cannot express more complex transformations, including processing of complex literal representations such as algebraic terms in a symbolic algebra system, or selecting from among widely differing implementation strategies depending on the parameter values.

4. Control abstractions uncoupled to any specific data type are not supported. Useful examples include finite state recognizers, finite state transducers, decision tables, and parsers. Such abstractions are conveniently packaged in "little languages," implemented conventionally with preprocessors at the expense of smooth integration and security. Examples of such preprocessors include YACC [10], LEX [14]. By providing the functionality of these programs as language extensions, we can provide for cleaner handling of errors.

From these considerations, the following general properties of a more powerful definitional facility emerge:

1. The computational domain of abstraction definitions must include the (abstract) syntactic domain of the base language, as types, record field names, procedure formal parameters, etc. must be manipulated, but do not have values in the object-level (run-time) domain.

2. ADT operations may possess any syntactic class, i.e. they may compute (meta)values of any (meta)type in the syntactic domain, and furthermore may accept arguments of any such class. The term *syntactic domain* is intended to include what is usually referred to as static semantics as well as more patently structural properties.

[1]An exception: CLU and Alphard permit the definition of type-specific iteration schemes using the built-in concept of an *iterator* or *generator*. In our work, such a mechanism can be defined in the language, as will be demonstrated in the sequel.

3. The definition of an abstraction is not a code template, but a procedure expressing what the language processor must do in order to translate its instances.

Note that we have taken an explicitly translation-oriented view of the semantics of abstraction mechanisms and of programming languages in general. We have taken this position for philosophical reasons as well as pragmatic ones. The main philosophical justification is that we view an abstraction mechanism as *imposing* an interpretation on constructs of the base language (and ultimately the machine itself) rather than *implementing* an abstraction in the sense of simulating or executing it. In this view, abstraction is something imposed on the run-time world from the outside, and is not a part of it. Somewhat more pragmatically, it should be clear what constraint checking (e.g. type-checking) goes on at translation time, i.e. at some designated time before the input data to the program is known, and at run time, i.e. when the input is known. This aids reasoning about the program, including both its static well-formedness and its performance at execution time. Finally, an interpreter may be constructed trivially by composing the translator with the target processor, while the reverse construction requires a partial evaluator which must rely on complex analyses that are not decidable in the general case.

Program abstractions may in general involve well-formedness predicates taking global program structure into account, or involve programming idioms or conventions impacting textually distant regions of the program. We restrict our attention to the class of abstractions that can be encapsulated in the form of language constructs, in which an instance of an abstraction can be identified with a set of occurrences of syntactic forms from a set of such forms constituting the definition of the abstraction.

## 3 Embedded metaprogramming facilities

A programming language is an abstraction of the underlying machine, which must ultimately engender any effect of a program on the physical world. Such abstraction serves multiple purposes, facilitating the programming process itself as well as suppressing irrelevant machine-dependent detail that would hinder program portability. As a program is expressed in a language even at the lowest levels of programming (machine language), it is appropriate to view the abstraction provided by a programming language as a metalinguistic one, i.e. a linguistic abstraction of another language.

Metaprogramming, or "programming about programs" is an old and recurring theme in computer science. Any language processor, whether as simple as a macrogenerator or as complex as a compiler, is a metaprogram. Traditionally, however, programs have had little to say about *themselves*. A compiler may compile programs written in the same language as the compiler, but it is unlikely that the program being compiled directs the compilation in other than the most passive way.

Brian Smith [24] has proposed a new programming mechanism, *reflection*, in which a program can take control over the selfsame interpreter under which it executes and direct its own interpretation. Reflection is a kind of self-reference, but unlike recursion, which is "flat," i.e. constrained to the object domain, reflection can shift levels into the definitional (meta) domain and manipulate explicit representations of computational notions that are left implicit at the object level, e.g. environments and continuations. The essential trick in reflection is to embed a model of the language in itself, and then provide a level-shifting mechanism whereby code appearing textually at the object level can be "lifted" so as to take effect in the context of the processor. Reflection is an extremely powerful definitional mechanism, admitting any extension expressible in terms of the embedded model. In Smith's 3-Lisp implementation, the model is a continuation-passing interpreter for the full dynamic semantics of 3-Lisp.

The difficulty with reflection is that it is perhaps *too* powerful, at least to admit an efficient implementation. Friedman and Wand [6] suggest that macros, for example, those in Lisp, constitute a form of compile-time reflection. By restricting reflection to the static properties of a program only, we can "compile away" the overhead of maintaining an explicit and effective semantic model. Unfortunately, the only static property of a Lisp program made explicit by the Lisp model is the interpretation of an expression in terms of another Lisp expression. Indeed, few static properties of Lisp programs are incorporated into the semantics of the language itself, though many interesting properties of particular programs might be statically inferred. In this research, we investigate a static mechanism analogous to reflection in the context of a language with non-trivial static semantics, where such a restricted form of reflection would nonetheless be useful.

We propose a class of definitional mechanisms, which we shall term *embedded metaprogramming facilities*, permitting the extension of the static semantics of a base language in terms of an *embedded* and *procedural* model of its static semantics. By embedded, we mean that the definitional mechanism, including the model, is a part of the language. By procedural, we mean to emphasize that we are to be programming computations over a domain including the program itself, not making declarative statements or assertions about its properties.

Why should a metaprogramming facility be included in a language? Why couldn't it be added through some extralingual mechanism such as a preprocessor or another component of an integrated programming environment? We can think of several reasons:

1. Conceptually, the proposed metalinguistic abstraction mechanism is intended to complement existing procedure and type definition facilities. Though the abstractions it defines may require reference to meta-level concepts in their implementations, in use, they play a role identical to that of other abstractions. In particular, the operations on an ADT may include language extensions such as special declarative or control forms, and we would like to export them from the type manager in the same way as ordinary procedures.

2. The metaprogramming facility must "understand" the base language, in at least a superficial sense. By this we mean that the semantic model for the base language provides a primitive set of concepts, represented by data types and operations, through which extensions interface to the base language. Inasmuch as the semantic model is necessarily language specific, so is at least a large part of the extension mechanism. We note that conventional macro facilities, as well as the structural macros of Lisp and so-called syntax macros, achieve a degree of language independence through an impoverished view of the base language. We wish to enrich this view as far as is practical, resulting in stronger language dependencies.

3. Extralingual facilities are likely to be nonstandard. If programmers are to regard metalinguistic abstraction in the same way as ordinary procedural and data abstraction, they must be able to rely on a consistent treatment across all implementations. Programmers may justifiably avoid availing themselves of nonstandard language features that may decrease code portability.

At this point, we should add that we are not averse to environmental support, but insist that metalinguistic definitional mechanisms should not be distinguished in this way from other more pedestrian ones. We can conceive of a programming environment where the notion of "language" hardly exists, having been subsumed into user interface and presentation. In such a context, it would be appropriate to abandon our insistence on metaprogramming facilities as a part of the programming languages they enrich.

## 4 Toward a metaprogramming facility

In this section we present a high-level recipe for adding an embedded metaprogramming facility to a programming language. As we shall see, there are pragmatic reasons why some languages are more suitable for this than others, though, in principle, the recipe is generally applicable.

- Embed a model of translation for the language in the language. In the crudest terms, write a compiler for the language in the chosen metalanguage, perhaps the same language it compiles. The model may be idealized in various ways, in that it need not reflect the actual details of the "real" compiler to the extent that these may be suppressed without compromising needed expressive power. In particular, we may choose to make the target language the same as the source language so as to suppress the details of compilation into a lower-level language.

- Provide a "hook" into the compiler, i.e. a mechanism by which the user can declare a new syntactic form and provide a fragment of code that drives the translation of its instances. This includes static semantics and the generation of a (dynamic-) semantically equivalent program fragment ultimately expressible in terms of built-in primitives.

Note that static semantics is defined here operationally as "what the translator does." The static semantics of the translated program are reinterpreted as the dynamic semantics of the translator.

Crucial to the whole enterprise is the translation model, which becomes part of the advertised interface to the translator, i.e. *a part of the language.* This is what we mean by the term "embedded." As such, the model must be reasonably abstract, sufficiently so to serve as a perspicuous definition of the language's static semantics, or at least those parts constituting the unchangeable primitive basis upon which definitions must be built. This leads to the following observations:

1. The object language must be factorable into largely independently definable constructs with well-defined static interdependencies. The "meat" of the semantics must go into the definitions, as the model can provide little more than the "glue" to hold the definitions together without becoming both complicated and overcommitted. For example, the *eval* function serves as an adequate model for Lisp in this regard.

2. The metalanguage should permit the expression of definitions that are sufficiently declarative to serve as an acceptable means of specification while admitting an acceptably efficient imperative reading. Symbol processing tasks involving linked structures or terms must be adequately supported. Experience suggests that this implies that the metalanguage processor should implement a retention-oriented storage management policy with automatic garbage collection.

With regard to point (1), we shall later present a syntax-directed translation scheme, a simplification of attribute grammars, in which this constraint manifests itself in the requirement that (a) every instance of a given syntactic class (phylum) is constrained to play an equivalent role in static analysis, and (b) all translation-time information flow must mirror the syntactic structure of the program.

With regard to point (2), we observe that these two requirements are diametrically opposed, thus a compromise is necessary. Languages such as Lisp, ML, and Prolog appear to have made this compromise in a manner satisfactory for our purposes. Languages such as Pascal, C, and Algol choose to emphasize an efficient imperative interpretation at the expense of declarative force. While the object language and metalanguage may coincide (as they do in our work), it should be clear that this choice is feasible only for an object language of fairly high level.

Of even greater concern is the necessity to ensure some degree of harmonious cooperation between separately defined abstractions. Our approach is limited merely to erecting abstraction barriers that limit the degree of interaction possible. The compilation model must then present a rather restricted view of the compilation process in which the abstraction mechanism itself is protected from corruption (e.g. protecting the integrity of type-checking) while providing a useful degree of extensibility. It is this area in which the most sensitive engineering issues arise, and that leaves open the greatest research opportunities.

## 5 Lisp and computational macros

Before proceeding with the specifics of our proposal, it is instructive to examine the language Lisp, which serves as both the best example of the ideas discussed here in an existing and well-known programming language, and as the inspiration and point of departure for this work. In many ways, Lisp seems ideally suited for embedded metaprogramming:

1. Lisp programs are directly and meaningfully represented as Lisp data structures. As a consequence, metastructural access is trivial in Lisp. The macro facility provides a hook into the language processor through which user-supplied code can take control of the compilation process.

2. Lisp admits a simple, perspicuous model of its own semantics, in the form of the *eval* function, which is both declaratively significant and suitable for execution. From *eval*, a simple model of Lisp translation suitable for extension purposes can be derived.[2]

[2]Readers familiar with large, feature-laden Lisp dialects such as ZetaLisp [17] and Common Lisp [27] may doubt the truth of this. Given that the model need not be complete, but merely sufficient to support extension, we believe the claim is true even of these dialects. Nonetheless, the claim might be more accurately made of Scheme [21], a Lisp dialect whose design has placed a much higher value

3. Lisp is well-suited to the symbol-manipulation tasks that the user-supplied translation procedures must perform.

4. A simple but general syntactic framework is provided by the base language and is conveniently adopted by user-defined language constructs. Syntactic complexities such as parsing do not burden the programmer or obscure the semantic issues.

Why then don't we just abandon this project and program in Lisp? Lisp's static semantics are trivial. As such, there is little that the language definition has to say about the static properties of a well-formed Lisp program. Indeed, the notion of "well-formedness" is on shaky ground in languages that take a view of semantics as completely dynamic as that of Lisp. While some may argue that this is a virtue, in that the late-binding semantics of Lisp encourages exploratory programming and incremental program development, we argue that this is an environment issue and not a language issue at all. While we grant that it may be easier to construct an incremental implementation of a language with impoverished static semantics (e.g. with dynamic typing), there is no necessary connection between this property and incrementality. We contend that a language should be designed with an eye toward early binding of as many semantically-relevant constraints as possible. Such an approach facilitates reasoning about programs, and allows greater confidence to be expressed in their correctness prior to execution. A more precise statement of the programmer's intent allows the translator to provide a higher level of assistance to the programmer, detecting errors that might otherwise go unnoticed, as well as permitting more efficient code generation. Our program, then, is to devise a mechanism allowing us to realize the flexibility of Lisp's metaprogramming facilities within the context of a language with rich static semantics. As a consequence of our sensitivity to semantics, we shall additionally correct a few flaws in Lisp macros due to Lisp's essentially syntactic treatment of macro expansion.

## 6 Semantic macros

The traditional embedded metaprogramming facility is the macro. Lisp macros depart from most others in that macro expansion is performed by an arbitrary user-defined computation, rather than a fixed expansion algorithm driven by a simple parameterized template. All macros, however, suffer from a name capture problem due to their substitution semantics, whether the expansion is in textual or structural form. The problem is that a fragment of program structure, as represented by a

on semantic cleanliness and simplicity than others.

string of tokens or a fragment of an abstract syntax tree, does not necessarily mean anything in isolation. Only in the context of some environment providing bindings for its free identifier occurrences can a meaning be assigned. The environment in effect where the macro expansion ultimately appears may not be known at the time that the macro definition is written. Likewise, arguments passed to a macro are usually unaware of bindings that may be established by the macro expansion that may conflict with their own free identifiers. Note that this is just the so-called *funarg problem* of dynamically-scoped Lisp dialects in a slightly different context. A more useful definition would require that the free identifiers in the expansion refer to the same entities that were bound to those identifiers in the context in which the macro was defined (cf. generic parameters in Ada). Similarly, macro parameters should *usually* be closed in the environment of the macro call, not in the environment of the expansion. We qualify this statement because there are cases where a macro definition is intended to establish a new environment for its arguments, or where an unbound identifier is needed as a macro argument, e.g. in a binding construct.

Another deficiency of macros is that they require the meaning of the macro call be expressed entirely in terms of source code in the base language. We wish to add new static semantics, as well as perform syntactic transformations. There is no way to propagate semantic attributes between textually distinct macro calls other than to stash them in global expansion-time variables. There is little that can be done with purely local information; in particular, we cannot implement type-checking.

In the light of these considerations, we propose a somewhat different interpretation of a macro-like construct. Let us suppose that the static "meaning" of every well-formed program fragment is represented by a set of translation-time attributes. (We shall avoid deeper questions of semantics by operationally defining semantics in terms of the translator.) Such attributes will in general exhibit a functional dependence on attributes of the surrounding context, most notably the environment. To simplify things somewhat, we shall collect the attributes supplied by context, i.e. the inherited attributes, into a single value, in general a tuple, and likewise the synthesized attributes. We shall define the *static semantic value* of any program fragment to be a function from an inherited attribute tuple to a synthesized attribute tuple. In effect, we have eliminated inherited attributes in the traditional attribute-grammar sense by allowing synthesized attributes to range over a higher-order domain, i.e. one with functional values. This trick is a familiar one from denotational semantics, and in fact what we are suggesting is a denotational approach to *static* semantics, while expressing dynamic semantics operationally in terms of the base language. Having thus defined the meaning of the built-in language constructs in this way, it is clear that new constructs, analogous to macros, may be defined that compute over the domain of semantic values rather than abstract syntax trees. We call these constructs *semantic macros* because they respect the static semantics of the structures they manipulate, and by analogy to Leavenworth's syntax macros [11], so named because they respect the syntax of the language.

The signatures of the inherited and synthesized attribute tuples of a semantic value define an interface that must be known in order to make use of the value. To insure that semantic values are composed only in ways that are meaningful with respect to these interfaces, we must impose a type discipline on their use. A most natural discipline arises from the syntactic type (*phylum*, or nonterminal symbol) associated with the language construct possessing the semantic value in question. By adopting the convention that there is exactly one semantic value type associated with every syntactic form of a given phylum, we guarantee that syntactically well-formed programs give rise to type-correct construction of semantic values.

It may be helpful at this point to examine a simple example before proceeding. A Pascal-style `for` loop is easily defined as a semantic macro, and illustrates in a simple context most of the mechanisms that will be needed for more complex definitions. The notation used is that of the language XL, to be defined in detail in the following section. For the present, we shall be content to rely on the reader's intuition.

```
(syntax EXPR (for [var NAME] [initial EXPR] [final EXPR] [body EXPR])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((val [start (initial env)]
                 [limit (final env)]
                 [cbody (close-expr body env (list var))]))
        (unless (equiv-types? integer-type (expr-type start))
          (type-error integer-type (expr-type start) "in FOR initial value"))
        (unless (equiv-types? integer-type (expr-type limit))
          (type-error integer-type (expr-type limit) "in FOR limit value"))
        ((EXPR
          (let ((val [limit ,,limit])
                (fun [loop ([,var integer]) void
                       (unless (> ,(name->*expr var) limit)
                         ,(lambda ([e environment])
                            (let ((val [b (cbody e)]))
                              (unless (equiv-types? void-type (expr-type b))
                                (error "FOR body must be of VOID type"))
                              b))
                         (loop (+ ,(name->*expr var) 1))) ]))
            (loop ,,start)))
         denv)))))
```

The `syntax` declaration introduces a new syntactic form. The first line of the definition indicates that the new form belongs to the phylum EXPR, that of expressions, and is written as

(`for` *var initial-value final-value body*)

where *var* is a NAME, i.e. an identifier, and *initial-value*, *final-value*, and *body* are EXPRs. In XL, all forms follow this fully-parenthesized, Lisp-like syntax. In the second line, we capture the global environment in effect at the time that this definition is compiled, which is the appropriate environment in which to resolve names appearing literally in the macro expansion. The `lambda` expression beginning on the third line, a function of an environment, constitutes the functional semantic value of the `for` form. The type `environment` represents a compile-time description of the run-time environment in which the new form will execute. Its values, like all values manipulated in this definition, are meta-values, belonging to the execution of the compiler, not the user's program.

When invoked, the semantic value function first closes its argument expressions in the proper environments. The expressions for the initial and final index values are closed in the environment in which the use of `for` appears, i.e. the value of the parameter `env`. These expressions are represented by functions of the environment, thus they are simply applied to `env`. The body cannot yet be closed, as it must see the binding of *var* yet to be created, but it should be closed with respect to all other variables. The function `close-expr` partially closes body in the environment `env`, leaving it open with respect to the name *var*.

Next, the types of the initial and final value expressions are verified to be of integer type, represented by the type descriptor value named `integer-type`. The form "`(EXPR ...)`" is a *quasiquotation*, a means of denoting a semantic value by writing an XL syntactic form that would possess it. The keyword `EXPR` indicates that the quasiquotation should be interpreted as an instance of the phylum EXPR. Forms within the quasiquotation preceded by "`,`" or "`,,`" denote *unquotations*, which are first evaluated to obtain the semantic value they represent. The "`,`" indicates that the following form evaluates to the semantic value to be used in place of the unquotation in constructing the value of the quasiquotation. In this example, the "`,,`" form is a shorthand for "`,(lambda ([e environment])` *unquoted-value*`)`."

The expansion of the `for` construct is a simple tail-recursive function definition and invocation. The function `name->*expr` maps a `token` (name) value to the semantic value of an expression consisting of that name. Within the body of the expansion, we capture the expansion-time environment after it has been augmented with the binding of *var*, using the form "`(lambda ([e environment]) ...)`," and use the captured value to complete the closure of the body. In XL, statements are modelled as expressions that return a value of the trivial type `void`. Thus, once the body is fully closed, removing all functional dependence on the environment, we verify that the body is of void type.

This example should serve as a paradigm for the use of semantic macros. Although subsequent examples will be much more complex, most will have an essential resemblance to this one.

## 7 XL: A language with semantic macros

In order to illustrate the notion of semantic macros, we have implemented a simple language incorporating this mechanism. The language, XL, for eXperimental Language, is intended only as an experimental vehicle, thus it favors simplicity over utility in all aspects unrelated to its novel definitional facility. Even so, we have chosen to focus on metaprogramming proper, and have not included the provisions for visibility control and namespace management that would be required for the definition of secure abstract data types.

Much of the semantics of XL were borrowed from Scheme [21], with further influences from ML [16] and FX-87 [7]. Since the XL compiler generates Scheme as its object code, the similarity to Scheme simplified implementation considerably. Unlike Scheme, however, XL is statically-typed, and accommodations had to be made for the additional declarative structure. The syntax of XL is embedded within S-expressions as in Scheme, but is considerably more structured in that many distinct syntactic classes are distinguished. This makes the language more interesting for extension purposes at the expense of a more deeply-nested structure, resulting in yet more parentheses than is customary in Lisp-like languages. Since the focus of this work is on semantics, we make no apologies for this syntactic awkwardness.

XL is an expression-oriented language, supporting higher-order functions and encouraging a functional style of programming. Side-effects are permitted, but play a less significant role in XL programming than in the Algol-family languages. The XL system is conversational, presenting to its user an interactive command interpreter. A program is merely a sequence of commands, entered either interactively or loaded from a file. Commands are provided for defining types and variables, evaluating expressions, and extending the language, as well as for "environmental" actions such as loading files. Commands are part of the predefined infrastructure, and are not subject to extension.

At any given time, commands are interpreted relative to a *unit*, consisting of two *namespaces*, one representing the context in which object-level computations are to be interpreted, and the other representing the one in which compile-time computations are to be performed. Each namespace is further subdivided into two parts: an *environment*, associating identifiers with their referent entities, e.g.

types and variables, and a *keyword table*, associating syntactic keywords with their interpretations. The keyword table is used by the parser at the time code is read into the unit and the environment is used as the environment argument to the semantic values of expressions.

XL supports a variety of data types. Integers, characters, and strings are imported from Scheme with few changes. Scheme symbols are called tokens, and are denoted in XL as symbols preceded by a single-quote. Constructed types include assignable references, vectors (with assignable components), heterogenous tuples, records with named fields, tagged unions, and homogeneous lists. The record and union types are unusual in that they may be incrementally extended, i.e. additional fields and variants may be declared after the initial type definition and with restricted visibility.[3]

## 7.1 Syntax

The lexical syntax of XL is fixed, and is largely borrowed from the underlying Scheme system. The one exception to the fixed lexical syntax is that symbols may be removed from the class of names (identifiers) and added to the class of reserved keywords as new special forms are defined.

- *Special symbols* play a special syntactic role, often serving to abbreviate another form. The special symbols of XL are "`(`", "`)`", "`[`", "`]`", "`,`", "`@`", and "`;`". These characters cannot be used in names or keywords.

- *Keywords* are written as Scheme symbols, and serve to indicate syntactic structure. They have significance only to the parser, and have no semantics. Keywords are reserved, and cannot be used as names.[4]

- *Names* are also written as Scheme symbols, and represent the names of entities such as types and variables. In the event that a symbol is defined as a keyword, that interpretation takes precedence and the symbol may not be used as a name. Conversely, any symbol that is not a keyword is always taken to be a name. Note that the status of a symbol as a keyword or a name is relative to a particular unit. A symbol may be a keyword in one unit and a name in another.

[3]Such a mechanism was proposed in [5], where an efficient implementation was sketched. We implement records as association lists, reducing efficiency but allowing a straightforward implementation in our incremental compilation system.

[4]This restriction is actually not enforced by the present implementation. Indeed, it is often apparent from context whether a symbol should be interpreted as a keyword or a name. Where ambiguity arises, it is resolved in favor of interpretation as a keyword. This is the same convention used in most Lisp dialects.

- *Numbers* are written as Scheme integers, and represent integer literals. In the interest of simplicity, XL does not support any other numeric types.
- *Characters* are written as Scheme characters, and represent character literals.
- *Strings* are written as Scheme strings, and represent string literals.
- *Tokens* are written as Scheme symbols preceded by a single quotation mark ("'"). Unlike in Scheme, the quotation mark is considered part of the syntax of the token literal, and may not be used in other contexts.
- *Booleans* are written `#t` for true and `#f` for false.

New keywords may be introduced dynamically as language extensions are defined, altering the set of symbols available for use as names. As such definitions take effect only between top-level forms (commands), each form will be parsed with respect to a stable set of keywords. Case is not significant in keywords, names, or tokens.

**Examples**

| | |
|---|---|
| `if, let, &bind` | Keywords |
| `x, integer, +` | Names |
| `0, 15, 12345` | Numbers |
| `#\a, #\Z, #\space` | Characters |
| `"", "foobar", "This is a string."` | Strings |
| `'foo, 'This-Is-A-Token` | Tokens |
| `#t, #f` | Booleans |

A complete and well-formed syntactic unit is called a *form*. Every form is associated with a syntactic type, its *phylum*. The simplest forms consist of a single lexical token and are called *atomic forms*. Atomic forms belong to predefined phyla and have a semantic value of predefined type, according to the following table:

| Atomic Form | Phylum Name | Type of Semantic Value |
|---|---|---|
| Name | NAME | Token |
| Number | NUMBER | Integer |
| Character | CHARACTER | Character |
| String | STRING | String |
| Token | TOKEN | Token |
| Boolean | BOOLEAN | Boolean |

All syntactic forms other than these atomic forms are called *composite forms.* Such forms are written `([`*keyword*`] [`*form* ...`])`, that is, an optional keyword followed by zero or more subforms and surrounded by parentheses. The keyword serves to distinguish one composite form from another. In the case that the keyword is omitted, the form is treated as if the special keyword `&null` had appeared. Within a phylum, the keyword must be unique; however, the same keyword may be used in distinct phyla. As a consequence, it is possible for two textually identical form instances to belong to different phyla, as determined by their syntactic contexts, and thus have differing semantic interpretations.

Within the phylum of expressions (EXPR), we further distinguish between function applications, in which the keyword is omitted or is `&null`, and all other forms, called *special forms.* These two cases deserve special mention because in most applications the function expression is a single name, rendering the application visually indistinguishable from a special form. Only by accounting for which symbols have been reserved as keywords can we discern the difference. Semantically, however, the difference is crucial. All applications are processed identically, thus it is the function itself that is of primary interest. Special forms, on the other hand, are processed in an idiosyncratic manner, to which the programmer's attention should be directed. Furthermore, since keywords do not denote values, a special form name cannot be passed as a parameter, though some special forms do in fact behave in a manner otherwise appropriate for an application.

Brackets ("`[`" and "`]`") denote a simple abbreviation used in declarations and the formal parameter lists of functions. The form `[`*form* ...`]` is treated as if `(&bind` *form* ...`)` had been written instead. The comma character "`,`" preceding a form is read as if `(unquote` *form*`)` had appeared. The character "`@`" preceding a form is read as `(unquote-list` *form*`)`. Similarly, the double comma "`,,`" and double atsign "`@@`" preceding a form abbreviate `(unquote-reduced` *form*`)` and `(unquote-list-reduced` *form*`)` respectively. The `unquote`... operators are part of the syntax of quasiquotation, and will be further explained in that connection.[5] Comments begin with a semicolon ("`;`") and extend to the end of the line.

In the remainder of this section, we introduce the syntactic forms of XL and their semantics. All built-in syntactic forms are displayed like this:

`(if` *pred*: EXPR *cons*: EXPR *alt*: EXPR`)` EXPR

[5]It is unfortunate that the user is not allowed to define more abbreviations of this kind. These abbreviations are implemented using the reader macro facility of the Scheme system in which XL is implemented, which is an undisciplined mechanism which we do not wish to expose to the XL user. The need for these abbreviations is entirely a consequence of our choice of the verbose Lisp-like syntactic framework, and could be avoided if the parser were more powerful.

The subforms are given descriptive names, and their phyla are indicated. The phylum to which the entire form belongs appears to the right. Functions are displayed similarly, showing the syntax of a typical application of the function. Since all function calls and their arguments belong to phylum EXPR, we display their types instead:

(+ *x*: integer *y*: integer) integer

Note the typographic convention used here: phylum names are displayed in small capitals and type names in the normal Roman font. Some special forms simulate generic, or polymorphic, functions. In this case, it is more informative to display them like functions:

(cons *elt*: $\alpha$ *list*: list-of($\alpha$)) list-of($\alpha$)

The meta-variable $\alpha$ denotes an arbitrary type, indicating that the cons function is applicable to lists of any element type whatsoever. Note that any form of EXPR that accepts a variable number of arguments or an argument of a polymorphic type must be a special form. Special form names do not denote values, thus such polymorphic operations cannot be bound to variables, passed as arguments, or made components of data structures.

## 7.2 Expressions

Expressions in XL denote values and may also have side-effects. Every expression has a *type*, which determines the contexts in which it may legally be used. Sometimes an expression is useful only for its side-effects, and need not yield any value at all. Such expressions usually yield the trivial value `*void*`, of type `void`.

Expressions fall into four classes: *literals*, *variables*, *applications*, and *special forms*. Literals denote values of certain built-in types, and are evaluated trivially. Variables are names bound to values, and stand for the value so bound. Applications have the form (*funexpr* [ *argexpr* ... ]) and denote the result of applying the function denoted by *funexpr* to the arguments denoted by the expressions *argexpr*. Special forms have idiosyncratic syntax and evaluation rules, but are always composite forms with an explicit keyword.

Many special forms are closely associated with a particular data type, and will be described in conjunction with that type; the others are described here.

*literal*: NUMBER EXPR
*literal*: CHARACTER EXPR

*literal*: STRING EXPR
*literal*: TOKEN EXPR
*literal*: BOOLEAN EXPR

With the exception of NAME, any form of the atomic phyla described in the previous section may appear alone as an expression, denoting a value of the same type as its semantic value.

*variable*: NAME EXPR

A variable denotes the value to which it is currently bound. A compile-time error is signalled if no binding for the variable is in scope, or if the variable is not bound to a value, e.g. in the case it is bound to a type.

`(begin` *exprs*: EXPR ...`)` EXPR

The expressions *exprs* are evaluated in sequence from left to right. The value of the entire `begin` expression is that of the last expression. The values of the preceding forms are discarded, thus they are useful only for their side-effects. The trivial instance `(begin)` yields `*void*`. Certain other forms, e.g. `lambda`, `let`, `when`, and `unless`, have bodies consisting of a sequence of expressions that are processed as if they were enclosed in a `begin` form. The bodies of such forms are thus called *implicit* `begin`*s*.

(*fun*: EXPR *args*: EXPR ...) EXPR

An expression written as a composite form without a keyword (or with a keyword of `&null`) is a function application. The expression *fun* is first evaluated, and must yield a value of a functional type. The arguments *args* are then evaluated in an unspecified order, yielding a tuple of values. The function is applied to the argument values, yielding a result and possibly performing a side-effect. A compile-time error is signalled if the types of the arguments do not match those specified by the type of the function.

`(lambda` FPARM-LIST *body*: EXPR ...`)` EXPR
(FPARM ...) FPARM-LIST
[*arg*: NAME *type*: TYPE] FPARM

A lambda expression evaluates to a functional value. The resulting function, when invoked on arguments of types specified by *type*, returns the value of *body* evaluated in the environment at the point where the `lambda` expression occurs, augmented with the bindings of the actual parameters to the variables *args*. The body is an implicit `begin`.

(`if` *pred*: EXPR *cons*: EXPR) EXPR
(`if` *pred*: EXPR *cons*: EXPR *alt*: EXPR) EXPR

The expression *pred* is first evaluated, and must yield a boolean value. If the value is `#t`, then the result is obtained by evaluating *cons*, else the value of *alt* is returned. The types of *cons* and *alt* must be the same. If *alt* is omitted, the default value `*void*` of type `void` is used in its place.

(`when` *pred*: EXPR *body*: EXPR ...) EXPR
(`unless` *pred*: EXPR *body*: EXPR ...) EXPR

The expression *pred* is first evaluated, and must yield a boolean value. The `when` form evaluates its body as an implicit `begin` in the case that *pred* evaluates to `#t`, else it returns `*void*`. The `unless` form is similar, but evaluates its body when *pred* evaluates to `#f`. In either case, the value of the body is discarded, and the entire conditional yields `*void*`. The two-armed conditional `if` should be used except when the value of the conditional is to be ignored.

## 7.3 Declarations

Declarations define the meanings of names. Names are bound to objects called *entities*. The set of entities includes variables and named types, and may be extended by the user. XL is a lexically scoped language, in which the scope of a definition is textually determined. Redefinitions of a name within the scope of a previous definition temporarily shadow the old definition within the scope of the redefinition.

(`let` DECL-LIST *body*: EXPR ...) EXPR
(*decl*: DECL ...) DECL-LIST

The `let` construct introduces new bindings into the environment. The elements of the declaration list are processed sequentially, each within the context of the bindings established by the declarations preceding it as well as those of the scope enclosing the entire form. Any side-effects produced by the declarations (e.g. from embedded expressions) are guaranteed to take place sequentially. The expression *body* is then evaluated as an implicit `begin` in the context of the new environment and yields the value of the entire `let` form.

(`val` VALBIND ...) DECL
[*var*: NAME *val*: EXPR] VALBIND

Value declarations introduce one or more bindings of variables to values. The expressions *val* are evaluated in an unspecified order. The resulting values are bound pairwise to the corresponding variables *var*, and these bindings are then established simultaneously by the declaration. The types of the variables are taken from their defining expressions.

(type TYPEBIND ...) DECL
[*tvar*:NAME *type*:TYPE] TYPEBIND

Type declarations introduce names for one or more types. Unlike value declarations, type declarations give a mutually-recursive interpretation to the list of type variables *tvar* in which the variables being bound are made accessible within the defining type expressions *type*.

(fun FUNBIND ...) DECL
[*var*:NAME FPARM-LIST *rtype*:TYPE *body*:EXPR] FUNBIND
(*args*:FPARM ...) FPARM-LIST
[*arg*:NAME *type*:TYPE] FPARM

Function declarations provide a convenient way to define function bindings as an alternative to using a value binding with a lambda-expression as the defining expression. Additionally, function declarations allow recursive and mutually-recursive references within a single fun form. It is not possible to define recursive functions using value declarations. The formal parameters *args* are specified in the same manner as for lambda expressions. The result type *rtype* must be explicitly declared, and must match that of the expression *body*.

## 7.4 Types

Types represent compile-time descriptions of values to be created and manipulated at run-time. Their primary purpose is type-checking, a compile-time analysis intended to prevent misinterpretation of a value by the application of inappropriate operations to it. Types also serve as repositories of compile-time information about the expressions to which they are attached, such as the valid component names available for selection from a record. In general, two built-in types are considered equivalent if they are either identical ground types (i.e. non-constructed types) or were obtained by the application of the same constructor to equivalent types. Two exceptions to this structural type equivalence rule are record and union types. User-defined types follow their own type equivalence rules as defined by the user.

The types of the XL base language are described below. Discussion of certain types used primarily in writing semantic macros will be deferred to a later section.

*typevar*:NAME TYPE

A name bound to a type denotes a type. A compile-time error will be reported if *typevar* is unbound or bound to an entity other than a type.

### 7.4.1 Void

The void type has a single trivial value upon which no operations are defined. It is used as the type of expressions that are to be executed only for their effects.

| | |
|---|---|
| `void` | TYPE |
| `*void*` | EXPR |

The identifier `*void*` is predeclared to refer to the trivial value, of type `void`.

### 7.4.2 Integers

The integer data type and its semantics are inherited from the underlying Scheme implementation. As presently implemented, the size of representable integers is limited only by available memory.

| | |
|---|---|
| `integer` | TYPE |

| | |
|---|---|
| (~ $x$: integer) | integer |
| (+ $x$: integer $y$: integer) | integer |
| (- $x$: integer $y$: integer) | integer |
| (* $x$: integer $y$: integer) | integer |
| (/ $x$: integer $y$: integer) | integer |

The usual arithmetic operations of negation, addition, subtraction, multiplication, and division on integers are provided. The result of integer division always has the sign of the product of its arguments. Note the use of "~" to denote negation.

| | |
|---|---|
| (= $x$: integer $y$: integer) | boolean |
| (/= $x$: integer $y$: integer) | boolean |
| (< $x$: integer $y$: integer) | boolean |
| (<= $x$: integer $y$: integer) | boolean |
| (> $x$: integer $y$: integer) | boolean |
| (>= $x$: integer $y$: integer) | boolean |

The integer relational operators equality, inequality, less-than, less-than-or-equal-to, greater-than, and greater-than-or-equal-to are provided. All return a boolean result.

### 7.4.3 Characters

The character type is inherited from the underlying Scheme implementation. In the present implementation, it includes the full ASCII character set.

`character` TYPE

`(char=?` *x*: character *y*: character) boolean
`(char/=?` *x*: character *y*: character) boolean
`(char<?` *x*: character *y*: character) boolean
`(char<=?` *x*: character *y*: character) boolean
`(char>?` *x*: character *y*: character) boolean
`(char>=?` *x*: character *y*: character) boolean

A full set of relational operators on characters is provided. All comparisons are case-sensitive according to the ASCII collating sequence.

`(integer->character` *n*: integer) character
`(character->integer` *c*: character) integer
`(character->string` *c*: character) string

The functions `integer->character` and `character->integer` convert between characters and their ASCII integer character codes. The function `character->string` returns a string of length one containing the given character.

### 7.4.4 Strings

Strings are immutable, indexable sequences of characters. Strings are provided as a ground type for consistency with Scheme, though vectors of characters would serve much the same purpose..

`string` TYPE

`(string=?` *x*: string *y*: string) boolean
`(string/=?` *x*: string *y*: string) boolean
`(string<?` *x*: string *y*: string) boolean
`(string<=?` *x*: string *y*: string) boolean
`(string>?` *x*: string *y*: string) boolean
`(string>=?` *x*: string *y*: string) boolean

A full complement of string relational operators are provided. String ordering is lexicographic using case-sensitive character comparison.

`(string-length` *s*: string) integer

This function returns the length of a string. Empty strings of length zero are permissible.

(`string-append` *head*: string *tail*: string) string

The string concatenation function `string-append` returns a string consisting of the characters of string *head* followed by the characters of string *tail*.

(`string-ref` *s*: string *index*: integer) character

This function returns the *index*-th character of the string *s*. The characters of *s* are indexed left to right by ascending integers starting from zero for the first character.

(`substring` *s*: string *start*: integer *end*: integer) string

This function extracts a substring of the given string *s*, consisting of successive characters starting from index *start* (inclusive) and continuing through *end* (exclusive).

### 7.4.5 Booleans

Boolean values represent the logical truth values "true" (`#t`) and "false" (`#f`).

`boolean` TYPE

(`bool=` *x*: boolean *y*: boolean) boolean
(`bool/=` *x*: boolean *y*: boolean) boolean

Boolean equality and inequality.

(`not` *x*: boolean) boolean
(`and` *x*: boolean ...) boolean
(`or` *x*: boolean ...) boolean

Boolean connectives. The function `not` returns the logical complement of its argument. The special form `and` evaluates its arguments left to right, and returns `#t` if all evaluate to true. If any argument evaluates to false, evaluation of further arguments is aborted and `#f` returned. If no arguments are provided, `#t` is returned. The special form `or` evaluates its arguments left to right, and returns `#f` if all evaluate to false. If any argument evaluates to true, evaluation of further arguments is aborted and `#t` returned. If no arguments are provided, `#f` is returned.

### 7.4.6 Tokens

Token types are analogous to Scheme symbols. The only attributes of a token are its identity and its string name. All symbols with the same name are taken to be identical. While strings could be used in the place of tokens, the restricted semantics of tokens encourages an efficient implementation in which tokens are represented compactly as pointers and quickly tested for identity using pointer equality.

`token` TYPE

(`eq?` *x*: token *y*: token) boolean
Compare two token values, returning true if they both denote the same (identical) token object.

(`token->string` *t*: token) string
Return the name of a token as a string.

(`string->token` *s*: string) token
Return the unique token with the given string name.

(`generate-token` *prefix*: string) token
Return a new, unique token, guaranteed not to be `eq?` to any other token. The exact form of the token's string name is unspecified, but will include the given string *prefix* as its initial substring.

### 7.4.7 Functions

Functional types describe mappings from a tuple of values to a single result value (possibly the trivial value `*void*`). Functions are created by `lambda` expressions or `fun` declarations. The only operation defined on functions is application.

(`fun` *types*: TYPE-LIST *rtype*: TYPE) TYPE
(*argty*: TYPE ...) TYPE-LIST
The type of a function mapping arguments of type *argty* to a result of type *rtype*.

### 7.4.8 References

References are assignable storage locations. With the exception of vectors, they constitute the only mutable objects in XL.

(**ref-to** *reftype*: TYPE) TYPE

Create a reference type whose instances are restricted to holding values of type *reftype*.

(**ref** *value*: $\alpha$) ref-to($\alpha$)

Create a reference value whose initial content is given by the expression *value*.

(**get** *reference*: ref-to($\alpha$)) $\alpha$
(**set!** *reference*: ref-to($\alpha$) *value*: $\alpha$) void

The function **get** fetches the current content of *reference*. The content of a reference may be altered with **set!**.

(**same?** *r1*: ref-to($\alpha$) *r2*: ref-to($\alpha$)) boolean

Returns true if references *r1* and *r2* are the same object, i.e. were created by the same invocation of **ref**.

### 7.4.9 Vectors

Vectors are homogeneous indexable aggregates of objects. The length of a vector is determined dynamically when it is created and is not considered a part of its type.

(**vector-of** *elttype*: TYPE) TYPE

Create a vector type whose instances contain elements of type *elttype*.

(**vector** *eltval*: $\alpha$ ...) vector-of($\alpha$)

Create a vector consisting of the elements *eltval*, which must all be of the same type. There must be at least one such expression, else it would be impossible to determine the type of the resulting vector. To create an empty vector, use **make-vector**.

(**make-vector** *size*: integer *initval*: $\alpha$) vector-of($\alpha$)

Create a vector consisting of *size* copies of the value of *initval*. The size must be a non-negative integer. An empty vector may be specified by a size of zero, but note that an initial value is still required, from which the type of the resulting vector will be determined.

`(vector-length` *vector*: vector-of($\alpha$)) integer

Returns the number of elements in *vector*.

`(vector-ref` *vector*: vector-of($\alpha$) *index*: integer) $\alpha$
`(vector-set!` *vector*: vector-of($\alpha$) *index*: integer *value*: $\alpha$) void

The function `vector-ref` selects the *index*-th element of *vector*. The elements of a vector are numbered left to right by ascending integers starting from zero. The value of a vector element may be altered with `vector-set!`.

### 7.4.10 Lists

Lists are immutable homogeneous sequences of objects.

`(list-of` *elttype*: TYPE) TYPE

Create a list type whose instances are restricted to elements of type *elttype*.

`(cons` *elt*: $\alpha$ *list*: list-of($\alpha$)) list-of($\alpha$)

Construct a new list in which *elt* is prepended to the elements of *list*.

`(car` *list*: list-of($\alpha$)) $\alpha$
`(cdr` *list*: list-of($\alpha$)) list-of($\alpha$)

The function `car` returns the first element of a list, signalling a run-time error if the list is empty. The function `cdr` returns the tail of the list, i.e. all but the first element, and also signals a run-time error if the list is empty.

`(list` *eltval*: $\alpha$ ...) list-of($\alpha$)

Create a list consisting of the elements *eltval*, which must all be of the same type. There must be at least one such expression, else it would be impossible to determine the type of the resulting list. To create an empty list, use `empty-list`.

`(empty-list` $\alpha$: TYPE) list-of($\alpha$)

Return an empty list of the specified element type.

`(length` *list*: list-of($\alpha$)) integer
`(null?` *list*: list-of($\alpha$)) boolean

The function `length` returns the number of elements in a list. To determine if a list is empty, `null?` provides a concise alternative to comparing the length with zero.

`(append` *head*: list-of($\alpha$) *tail*: list-of($\alpha$)`)` list-of($\alpha$)

The list concatenation function **append** returns a list consisting of the elements of list *head* followed by the elements of list *tail*. Both lists must have the same element type, else a compile-time error is reported.

`(reverse` *list*: list-of($\alpha$)`)` list-of($\alpha$)

Return a list consisting of the elements of *list* but in reverse order.

### 7.4.11 Tuples

Tuples are the simplest heterogeneous data structure supported by XL. The fields are indexed positionally. Tuples with named components may be provided by extension, so we provide only this simple form as a primitive.

`(tuple-of` *ftype*: TYPE `...)` TYPE

Create a tuple type with components of the given types.

`(make-tuple` *fval*: EXPR `...)` EXPR

Construct a tuple value.

`(tuple-ref` *element*: NUMBER *tval*: EXPR`)` EXPR

Select an element from a tuple. The elements are numbered left to right starting with zero.

### 7.4.12 Records

Records are immutable heterogeneous aggregates of named components. It is not necessary to declare all of the components of a record type when it is first introduced. Additional components, whose visibility may be restricted to a smaller region of the program, may be declared incrementally. The declaration of these extensions takes the form of a type declaration, but is best considered as declaring an expanded view of the extended type. The additional components are only visible within the scope of the extension, but are taken to be a permanent part of all objects of the (unextended) base type. Values that already exist at the time an extension is made are modified retroactively (in effect) to contain the new components, using default values supplied by the extension declaration. Conceptually, all objects of the base record type, or any extension thereof, contain, from the moment of their creation, a complete set of all components declared or ever to be declared in extensions of that type.

`(record-of` *tag*:NAME FIELDSPEC ...`)` TYPE
`[`*fname*:NAME *ftype*:TYPE`]` FIELDSPEC

Create a base record type with components named *fname* of the type indicated by the corresponding *ftype*. Each occurrence of the **`record-of`** constructor introduces a new record type, distinct from any others including those with an identical set of components. The name *tag* is used as an identifying label in the printed representation of the type.

`(extend-record` *rtype*:TYPE EXTFIELDSPEC ...`)` TYPE
`[`*fname*:NAME *fval*:EXPR`]` EXTFIELDSPEC

Declare additional components belonging to the record type *rtype*. A default initial value *fval* must be provided for each new component, which will serve as the value of the component for record instances created in a scope in which the new component was not visible. The types of the components are taken from those of the initial value expressions. The type returned by **`extend-record`** is equivalent to *rtype* and all other extensions thereof for type-checking purposes, but will reveal an extended set of components to **`select`** and **`update`**. This works because any missing components will be supplied from the default values if an attempt is made to access them.

`(make-record` *rtype*:TYPE FIELDINIT ...`)` EXPR
`[`*fname*:NAME *fval*:EXPR`]` FIELDINIT

Create an instance of the record type *rtype*. A value must be provided for all components defined by *rtype*.

`(select` *rtype*:TYPE *fname*:NAME *rval*:EXPR`)` EXPR

Return the value of the component named *fname* from the record *rval*. A compile-time error is signalled if no component by that name is defined by *rtype*. The type *rtype* must be the same as the type of *rval* or an extension thereof.

`(update` *rtype*:TYPE *rval*:EXPR FIELDINIT ...`)` EXPR
`[`*fname*:NAME *fval*:EXPR`]` FIELDINIT

Create a copy of the record value *rval* in which the named components are replaced with the indicated values and all other components remain the same. Note that this is a shallow copy, i.e. the components are not copied and retain their original identity. All components named must be defined by *rtype*, which must the be same as the type of *rval* or an extension thereof.

### 7.4.13 Unions

Unions are immutable objects consisting of a single component drawn from a finite number of typed variants, and a label indicating to which variant the component of the union belongs. Like record components, additional union variants may be declared with possibly restricted visibility.

(`union-of` *tag*: NAME VARSPEC ...) TYPE
[*vname*: NAME *vtype*: TYPE] VARSPEC

Create a union type with variants named *vname* of the type indicated by the corresponding *vtype*. Each occurrence of the **`union-of`** constructor introduces a new union type, distinct from any others including those with an identical set of variants. The name *tag* is used as an identifying label in the printed representation of the type.

(**`extend-union`** *utype*: TYPE EXTVARSPEC ...) TYPE
[*vname*: NAME *vtype*: TYPE] EXTVARSPEC

Declare additional variants belonging to the union type *utype*. The identifier *vname* is the variant tag, and *vtype* is the required type of the component when the variant is in effect.

(`is?` *utype*: TYPE *vname*: NAME *uval*: EXPR) EXPR
(`inject` *utype*: TYPE *vname*: NAME *value*: EXPR) EXPR
(`project` *utype*: TYPE *vname*: NAME *uval*: EXPR) EXPR

The variant enquiry predicate `is?` returns true if the union value *uval* belongs to the variant *vname*, else it returns false. Union values are created by `inject`, which creates a union value whose variant is *vname* and component value is *value*. The type of the component value must agree with that declared for the variant. The component of a union is extracted with the `project` operation, which must specify the expected variant. A run-time error is signalled if the variant *vname* is not valid for the value *uval*. For `is?` and `project`, the type of *uval* must be equivalent to *utype* or an extension thereof. In all cases, the variant *vname* must be defined by the type *utype*.

### 7.4.14 Options

An option type is similar to a union of two variants in which one of the variants is empty, i.e. of type `void`. Unlike unions, however, option types follow a structural rule of type equivalence, as do lists, vectors, and tuples. Options exist primarily for representing the semantic values of optional constituents in `syntax` declarations, but are often useful for "out of band" signalling, e.g. of failure. In Lisp, many functions signal failure by returning `nil` instead of the expected value. We can model this convention in XL by returning an empty option.

`(option-of` *basety*: TYPE`)` TYPE
Create an option type with base type *basety*.

`(make-option` *value*: $\alpha$`)` option-of($\alpha$)
`(empty-option` $\alpha$: TYPE`)` option-of($\alpha$)
A non-empty option is created by `make-option`. The operator `empty-option` yields an empty option of the specified base type.

`(empty?` *oval*: option-of($\alpha$)`)` boolean
`(value` *oval*: option-of($\alpha$)`)` $\alpha$
The predicate `empty?` returns true if the option *oval* is empty, else it returns false. The component value of a non-empty option may be extracted with `value`. A run-time error is signalled if the option is empty.

`(if-present` *var*: NAME *val*: EXPR*cons*: EXPR *alt*: EXPR`)` EXPR
The expression *val* must evaluate to a value of an option type. If the option is not empty, the expression *cons* is evaluated with *var* bound to the contents of the option, else the expression *alt* is evaluated with *var* unbound. The result is that of the evaluated subexpression. The types of *cons* and *alt* must be equivalent. This construct provides a convenient shorthand for dealing with option types.

## 7.5 Commands

`(define` *declaration*: DECL ...`)` COMMAND
Process a a list of declarations, giving each a scope enclosing the declarations immediately following it and all subsequent commands within the unit in which it appears. When used interactively at the top-level prompt, the scope is the remainder of the interactive session. A brief message is displayed summarizing the definitions introduced.

`(eval` *expression*: EXPR`)` COMMAND
Evaluate an expression in the current global environment and display the value and type of the result returned.

`(load` *filename*: STRING`)` COMMAND
Read the contents of a file into the current unit as if typed interactively at the top-level prompt.

(`verbose` *verbose?*: BOOLEAN) COMMAND

When loading from a file, by default, the messages displayed by `define` and `eval` are abbreviated. This behavior is controlled by the verbose option, which can be set by the `verbose` command. When verbosity is set true, the full message is displayed as in interactive use, else the abbreviated display is presented.

(`debug` *debug?*: BOOLEAN) COMMAND

Enable or disable display of debugging information, most notably the Scheme code produced for each compiled XL form. This command is intended for use by the implementor, and requires some knowledge of the compiler internals in order to understand the display.

(`scheme`) COMMAND

Enter a Scheme break loop for debugging. Typing "`(xl-reset)`" at the Scheme prompt returns to XL.

(`quit`) COMMAND

Exit the XL system.

## 7.6 Metaprogramming facilities

The facilities of XL thus far presented have been concerned primarily with object-level computations, that is, computations about the problem we are trying to solve. We now turn to the special facilities of XL that support computations about the program itself. These metaprogramming facilities, provided within the strongly-typed framework of the XL base language, constitute the novel aspects of this work.

When metaprogramming in XL, we do not manipulate uninterpreted syntactic structures as in conventional macro expansion; instead, we manipulate directly the semantic values that would be associated with these structures by the interpretation functions of a translation semantics. Alternately, we can say that we are manipulating *interpreted* syntactic structures, in which the nodes of the abstract syntax tree are annotated with the function that will perform the portion of a syntax-directed translation associated with the node's phylum. Both views are correct, as representational details are suppressed so that it is impossible to recover the syntactic structure from the values our metaprograms manipulate, allowing access only on the semantic level.

Every XL form is interpreted with respect to a *keyword table* and an *environment*. The keyword table governs the internalization of the textual form of the program as a semantic value. The interpretation of a type, declaration, or expression will in general depend on the referents of names occurring free in that form. Any identifiers that are free in a top-level form must be resolved with respect to the appropriate global environment, which maps identifiers to their referent entities. The object-level environment describes entities belonging to the object program. In particular, identifiers in the object-level environment may be bound to descriptions of variables that will not exist until the object program is executed. Clearly, compile-time metaprograms must be interpreted with respect to an environment in which the entities described will exist at compile time, i.e. "meta-run-time." We thus provide a *meta-level environment* or *meta-environment* to be used in compiling metaprograms. Likewise, if we wish to internalize additional metaprogram text, we must interpret it with respect to a meta-level keyword table. These components, a keyword table and an environment, make up the context, or *namespace*, of the meta-level and object-level components of a unit.

It should be made clear that the terms "object-level" and "meta-level" are relative. When processing the object program, the bodies of semantic macro definitions and their supporting declarations will be viewed as metaprograms. However, we may wish to extend the language in which metaprograms are written, in which case the extensions will be internalized into the former meta-environment as object-level code, and a new meta-environment created in which to compile these extensions. Any given unit, i.e. an interactive session or its transcript in a file, has direct access only to two adjacent levels in the hierarchy of "meta-ness." To load extensions into the meta-namespace of a unit, the `meta-load` command is used, which loads the contents of the given file treating the meta-namespace of the invoking unit as the object-namespace for the loaded code and creating a new meta-namespace as a copy of the standard meta-namespace.[6]

As each form is read, it is converted immediately into a semantic value whose type is dependent on the phylum to which it belongs. All semantic values arising from instances of a given phylum have the same type. This allows us to guarantee that syntactically correct programs give rise only to type-correct construction of semantic values.

The three most important predefined phyla for extension purposes are given below. Each instance of these phyla is represented as a value of the type given in the following table:

[6]This is a rather weak level-shifting mechanism, and places too much significance on files. It may be difficult to remove this unwelcome intrusion of files into the semantics of the definitional mechanism due to fact that syntax bindings are resolved at read time.

| Phylum | Type of Semantic Value |
|---|---|
| EXPR | **expr = environment* $\rightarrow$ *expression* |
| TYPE | **type = environment* $\rightarrow$ *description* |
| DECL | **decl = environment* $\rightarrow$ *declaration* |
| | where *declaration = *expr* $\rightarrow$ *expression* |

EXPRs take an environment as an argument and return an *expression*, a fully name-resolved and type-checked code fragment whose type may be queried. TYPEs map an environment to a *description*, a compile-time type descriptor. DECLs map an environment to a *declaration*, a function from an unclosed expression (i.e. a function from an environment to an expression) to a closed expression.

The semantic value types of each of these phyla are given names in the standard meta-environment: `*expr`, `*decl`, and `*type`. In addition, the semantic value types of the other predefined phyla are also named using the convention of prepending an "`*`" to the phylum name. The structure of these other semantic value types is hidden, and values of such types may only be used to build other semantic values as arguments to quasiquotations.

(`meta-define` *declaration*: DECL ...) COMMAND
(`meta-eval` *expression*: EXPR) COMMAND
(`meta-load` *filename*: STRING) COMMAND

These operations perform analogously to `define`, `eval`, and `load`, but shift up a level in the hierarchy of "meta-ness." More precisely, each is executed with respect to a unit in which the object namespace is the the meta-namespace of the unit from which the command is invoked and the meta-namespace is a newly-created copy of the standard meta-namespace. The `meta-define` command is used primarily to define auxiliary definitions needed by the semantic values of syntax definitions. The `meta-load` command can be used to load libraries of language extensions into the current unit. The `meta-eval` command is helpful when debugging definitions in the meta-namespace.

(`meta-level-listener`) COMMAND

Enter a new command interpreter ("listener") shifted up a level. The previous meta-namespace is treated as the object namespace, and a new meta-namespace is created, as in `meta-load`. Typing the end-of-file character (Control-D) returns to the previous command interpreter.

## 7.7 Syntax extensions

To define new language facilities using semantic macros, it is necessary to declare the structure of new syntactic forms, their syntactic roles, and their semantic values.

New syntactic classes are introduced with `phylum` declarations. New syntactic forms are introduced with `syntax` and `atom-syntax` declarations, which specify to which phyla the new forms belong.

(`phylum` *phylum*: NAME *semtype*: TYPE) COMMAND
(`phylum` *phylum*: NAME *semtype*: TYPE *reducer*: EXPR) COMMAND

The `phylum` command introduces a new phylum. It has two effects:

1. A new phylum is added to the keyword table of the object namespace. Subsequent commands may add new syntactic forms to this namespace belonging to the new phylum or containing instances of it as constituents. If a phylum of the same name has previously been defined in the same namespace, the old definition is shadowed, and new references to the phylum name refer to the new definition. Syntax declarations processed when the previous definition was in effect retain their meanings.[7]

2. The phylum-specific quasiquotation form is created and declared as a form of the EXPR phylum in the meta-namespace. If a special form using the same keyword was previously defined, whether due to a previous phylum declaration or not, the old definition is shadowed and is no longer accessible.

The optional *reducer* argument is significant only to quasiquotations belonging to the new phylum and will be explained in the sequel.

(**syntax** *phylum*: NAME *pat*: ⟨pattern⟩ *semval*: EXPR) COMMAND
(⟨item⟩ ...) ⟨pattern⟩
(*keyword*: NAME ⟨item⟩ ...) ⟨pattern⟩
! ⟨item⟩
& ⟨item⟩
[*var*: NAME *phylum*: NAME] ⟨item⟩

Syntax declarations introduce new composite forms. Usually, an explicit keyword is provided, but if omitted, it defaults to the keyword `&null`. Normally, each bracketed argument specification matches a single constituent of the specified phylum *phylum*, binding its semantic value to the variable *var*. The special symbols "`!`" and "`&`" indicate optional and iterated (repeated) constituents respectively. The pseudo-phyla "⟨pattern⟩" and "⟨phylum⟩" are used only to decompose

[7]In general, no declaration can have a retroactive effect on the interpretation of previously declared names. Thus the scoping discipline for the XL top-level is like that of ML rather than Lisp. Note, however the unusual retroactive modification of objects that takes place during record and union extension.

a lengthy syntactic description. They are part of the idiosyncratic syntax of syntax patterns, and do not have semantic values.

Any constituents specified to the right of a "`!`" are optional, and the semantic value bound to the pattern variable *var* is of type `(option-of` *semvaltype*`)`, where *semvaltype* is the semantic value type of the matched phylum. If an optional constituent is present, the option contains its semantic value, else the option is empty. Note that if any of a group of optional constituents are provided, they all must be. For example, the pattern `(foo ! [a NUMBER] [b NUMBER])` matches `(foo)` and `(foo 1 2)`, but not `(foo 1)`. To allow all three to match, use the pattern `(foo ! [a NUMBER] ! [b NUMBER])`. The type of the value bound to the pattern variable of an optional constituent does not depend on the number of "`!`" symbols that appear to its left.

The symbol "`&`" must be followed by only one constituent, and indicates that zero or more occurrences will be matched. The semantic value bound to *var* is of type `(list-of` *semvaltype*`)`, where *semvaltype* is the semantic value type of the matched phylum. An iterated constituent is always considered optional, as the list of occurrences may always be empty. If governed by a "`!`", the interpretation as an iterated constituent supersedes the interpretation as an optional one.

When an instance of the declared syntactic form is read, the expression *semval* is evaluated in the context of the bindings of the pattern variables *var* to yield the semantic value for the form. The expression *semval* is parsed and closed with respect to the meta-namespace of the unit in whose object-namespace the syntax declaration appears. If a phylum is extended with a new form having the same keyword as an existing one, the old definition is shadowed.

## Example

```
(syntax EXPR (increment [exp EXPR])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      ((EXPR (+ ,,(exp env) 1)) denv))))
```

The intent of this definition might more easily be captured by a function definition, but we use a `syntax` definition for the sake of example. Note the use of `(definition-environment)` to capture the object-level environment in effect at the time the syntax definition is processed, so as to provide an appropriate environment in which to close the expansion. This assures that the interpretation of identifiers appearing free in the quasiquoted expansion "`(EXPR ...)`," in this case only "`+`", depends only on the context of the definition, not the context of use. This is analogous to the closure of `lambda` expressions in their contexts of definition.

(**atom-syntax** *phylum*: NAME ⟨atompat⟩ *semval*: EXPR) COMMAND
[*var*: NAME *atomic-phylum*: NAME] ⟨atompat⟩

The simplest syntactic forms are the built-in atomic phyla: NAME, NUMBER, CHARACTER, STRING, TOKEN, and BOOLEAN. The **atom-syntax** command declares a coercion from an atomic phylum to another (non-atomic) phylum. Instances of the designated atomic phylum *atomic-phylum* are then permitted to appear in contexts where an instance of the target phylum *phylum* is required. The semantic value expression *semval* defines the semantics of the coercion. It is evaluated in an environment in which the pattern variable *var* is bound to the semantic value of the form as an instance of *atomic-phylum*, and must yield a value of the type demanded by the target phylum. The semantic value is parsed and closed with respect to the meta-namespace (keyword table and environment) of the unit in which the command is invoked.

The function of **atom-syntax** may be made somewhat clearer by an example. The following might be part a metacircular definition of the built-in phylum EXPR, which we will call MYEXPR:

```
(atom-syntax MYEXPR [v NAME]
  (lambda ([env environment])
    (lookup-variable v env)))

(atom-syntax MYEXPR [n NUMBER]
  (lambda ([env environment])
    (make-integer-literal n)))
```

If a phylum is extended using **atom-syntax** with an atomic phylum already defined for it, the old definition is shadowed and all subsequent instances of the phylum will be parsed according to the new definition. The meanings of previously parsed forms, however, remain unchanged.

## 7.8 Quasiquotation

In metaprogramming, we explicitly manipulate semantic values. While some operations constructing semantic values do not correspond to any source-level construct of the object language, most do. Thus it is convenient to notate these values using the same syntax as is normally employed in the object language.

For each phylum, a phylum-specific *quasiquotation* operator is introduced which denotes the semantic value of a given instance of the phylum as an expression. In general, the quasiquotation operator has the form (*phylum-name form*) where *phylum-name* is the name of the phylum, and *form* is an instance of the phylum in which one more constituents has been replaced by an *unquoting operator*. The unquoting operators belong to every phylum and denote an instance of the phylum demanded by their context. The argument to an unquoting operator is an expression, whose value becomes the semantic value of the constituent for which the unquotation stands. Unquoting operators are legal only as constituents of a quasiquotation, and will be rejected as a syntax error in any other context.

`(unquote` *semval*: EXPR`)` ⟨any phylum⟩
When used in a syntactic context within a quasiquotation, this operator denotes an instance of the phylum expected in that context and whose semantic value is given by the expression *semval*. A type error is reported if the type of this expression is not equivalent to that required of the semantic value of the expected phylum. The abbreviation "`,`*semval*" is normally used in preference to `unquote`.

`(unquote-list` *semval*: EXPR`)` ⟨any phylum⟩
This unquoting operator is similar to `unquote`, but when used where an iterated constituent is required, it stands for the entire list matched by that parameter. The semantic value must be a list of the semantic values appropriate for the parameter. If `unquote-list` is preceded or followed by other unquoted expressions, the list will be spliced in, analogous to `unquote-splicing` in Scheme. The abbreviation "`@`*semval*" is normally used in preference to `unquote-list`.

The semantic values for most major phyla, including the important built-in phyla EXPR, DECL, and TYPE, exhibit a functional dependence on the environment. Each has a type of the form *environment* $\rightarrow \tau$ for some type $\tau$. More generally, though not in the XL base language, there may be multiple contextual parameters. The semantic attributes of the phyla (e.g. the type of an EXPR) are associated with the values of $\tau$. These values, obtained by applying the functional semantic value to one or more context parameters such as the environment, are said to be the *reduced* semantic values for their respective phyla. Likewise, the application of such a functional semantic value so as to make its semantic attributes accessible, fully-resolved and independent of context, is called *reduction*.

Metaprogram code performing semantically-sensitive program manipulations must generally deal with reduced semantic values, though unquotations using `unquote` or `unquote-list` will demand the unreduced values. It is straightforward to coerce a reduced semantic value to an unreduced value by enclosing it in a `lambda`-expression with dummy context parameters. The situation arises so often, however, that it is worth a bit of syntactic sugar to avoid excessive clutter.

`(unquote-reduced` *semval*: EXPR`)` ⟨any phylum⟩
`(unquote-list-reduced` *semval*: EXPR`)` ⟨any phylum⟩
These forms are similar to `unquote` and `unquote-list`, but are applicable only

to phyla for which a reducer was specified in the `phylum` declaration, or to one of the built-in phyla EXPR, DECL, and TYPE. In this case, the reducer is applied to the argument expression before it is used in the normal manner of `unquote` or `unquote-list`. The reducer should coerce the reduced semantic value to its unreduced form, generally by "wrapping" its argument (the reduced value) in a `lambda`-expression so as to provide a semantically-vacuous "dependence" on the context parameters. The reducer should be coded to perform the coercion to the declared semantic value type of the phylum to which it belongs. The extension to lists of such semantic values, needed by `unquote-list-reduced`, will then be generated automatically. The abbreviations "`,,`*semval*" and "`@@`*semval*" are normally used in preference to `unquote-reduced` and `unquote-list-reduced`.

## 7.9 The standard meta-namespace

With the exception of quasiquotation, the facilities described so far are a part of the object language, in which computations about the problem domain are expressed. A few commands make reference to the immediately superior meta-namespace, but the command names themselves belong to the object namespace. In this section, a number of types, functions, and special forms are described that concern computations over representations of concepts of the XL language itself. These *metaprogramming* facilities are available in the the meta-namespace of every unit. In fact, every namespace is the meta-namespace of *some* unit with the exception of the root namespace, i.e. the object namespace of the unit accessible from the interactive command interpreter. Thus these definitions are available in all namespaces but the root one.

The standard meta-namespace, from which all other meta-namespaces derive their initial contents, consists of all the object-level definitions previously discussed and, additionally, further definitions as described below.

We will occasionally make reference to run-time errors signalled by certain functions defined in the meta-namespace. Note that what we consider run-time when discussing metaprogramming features is compile-time from the standpoint of the object program, during whose compilation the metaprogram will be executed. Thus all run-time errors in metaprogram execution will be reported during compilation of an object program that exercises the offending metaprogram code.

### 7.9.1 Contours and environments

Values of type `environment` represent the lexical (textual) context with respect to which expressions, types, and declarations are interpreted. In the base language, this context consists of a mapping from names to their referent entities; however, other sorts of contextual information may be incorporated into the environment by type extension. We shall see an example of such an extension in Section 8.3.

`entity` TYPE
`contour` TYPE
`environment` TYPE

Values of type **`entity`** model compile-time properties of named entities, such as types and variables. The entity type is a union and may be extended using **`extend-union`** declarations. Initially, variants representing variables and type names exist, but are hidden from the user. Values of type **`environment`** are records, and may be extended using **`extend-record`**. Initially, environments contain a single hidden component representing a mapping from names (of type **`token`**) to entities. Values of type **`contour`** represent lexical binding contours, mappings from names to entities in which each bound name is associated with exactly one entity. To augment the bindings of an environment, a contour containing the new bindings is first constructed, then a new environment is created in which the contour is "nested" within the previous environment. The new bindings shadow any previous bindings of the same names.

`(empty-contour)` contour
`(add-binding` *name*: token *entity*: entity *contour*: contour`)` contour
`(bound-in-contour?` *name*: token *contour*: contour`)` boolean

The nullary function **`empty-contour`** returns a lexical binding contour containing no bindings. The function **`add-binding`** creates a new contour differing from *contour* only in that the name *name* is bound the entity *entity*. A run-time error is signalled if the name is already bound in the contour. The predicate `bound-in-contour?` tests whether a given name is bound in a contour.

`(empty-environment)` environment
`(add-contour` *contour*: contour *env*: environment`)` environment
`(bound-in-environment?` *name*: token *env*: environment`)` boolean
`(binding` *name*: token *env*: environment`)` entity

The nullary function **`empty-environment`** returns an environment containing no bindings. The function **`add-contour`** augments the bindings of *env* with those of *contour*, in which new bindings shadow any existing bindings involving the same name. The predicate `bound-in-environment?` tests whether a given name is bound in an environment. The function `binding` retrieves the entity associated with the name *name* in the environment *env*, signalling a run-time error if no binding exists.

`(current-environment)` environment

The nullary function **`current-environment`** returns the global environment of the object-level namespace of the current unit. Note that this function is only visible within the unit's meta-namespace, so it is impossible for code executed at object-level to have access to its own environment.

`(definition-environment)` environment

The special form **`definition-environment`** returns the value of `(current-environment)` at the time the form was compiled. It is used to capture a suitable environment for closing quasiquotations within syntax declarations.

`(merge-environments` *env1, env2*: environment *vars*: list-of(token)`)` environment

Yields an environment identical to *env1* except that for every name in the list *vars* for which a binding exists in *env2*, the binding in the result is that from *env2*. This function is useful for constructing environments in which to partially close expressions that must be left open with respect to a specified list of names. The function `close-expr`, defined below, partially closes the unclosed expression *exp* (of type `*expr`) in the environment *denv*, allowing it to capture the names *vars* in the context of use.

```
(meta-define
  (fun [close-expr ([exp *expr]
                    [denv environment]
                    [vars (list-of token)]) *expr
       (lambda ([env environment])
         (exp (merge-environments denv env vars)))]))
```

The function `close-expr`, as well as similar functions for declarations and types, are predefined in XL, and will be described in the sequel.

### 7.9.2 Expressions

Values of type `expression` denote fully type-checked expressions in which all free identifiers have been resolved and all semantic attributes computed. Values of type `*expr` are functions from an `environment` to an `expression`.

`expression` TYPE

`(expr-type` *expr*: expression`)` description

The type expression is a record type, with no fields initially visible. The function `expr-type` is provided to query the type attribute of an expression. Note that if this component were made directly visible, it would be possible to subvert the type-checking mechanism by altering the type component indiscriminately with `update`.[8]

(`name->*expr` *name*: token) *expr
(`number->*expr` *number*: integer) *expr
(`character->*expr` *char*: character) *expr
(`string->*expr` *string*: string) *expr
(`boolean->*expr` *boolean*: boolean) *expr
(`token->*expr` *token*: token) *expr

These functions effectively "unquote" values of the atomic phyla. That is, they return the semantic value that an expression consisting of an atomic phylum instance would have, given that the instance had the specified value.

(`close-expr` *expr*: *expr *env*: environment *vars*: list-of(token)) *expr
(`close-expr-list` *expr*: list-of(*expr) *env*: environment *vars*: list-of(token)) list-of(*expr)

The function `close-expr` partially closes an expression in a given environment, leaving it open with respect to the variables enumerated in *vars*. The function `close-expr-list` just maps `close-expr` over a list of `*expr` values.

## 7.10 Declarations

Values of type `declaration` are functions from an unclosed expression of type `*expr` to a closed expression of type `expression`.

`declaration` TYPE

(`close-decl` *decl*: *decl *env*: environment *vars*: list-of(token)) *decl
(`close-decl-list` *decl*: list-of(*decl) *env*: environment *vars*: list-of(token)) list-of(*decl)

The function `close-decl` partially closes a declaration in a given environment, leaving it open with respect to the variables enumerated in *vars*. The function `close-decl-list` just maps `close-decl` over a list of `*decl` values.

[8]The present implementation, for historical reasons, implements the `expression` type as a ground type, not an extensible record, precluding further extension of the set of expression attributes.

### 7.10.1 Types

Types impose an interpretation on expressions, restricting the contexts in which they can occur to those operations that are prepared to operate meaningfully upon them. Additionally, they serve as repositories for compile-time properties of expression values, such as their "shapes" as data structures.

Because of the use of the word "type" in connection with the phylum TYPE, we call the reduced semantic value of that phylum by the name `description`. Values of (meta-) type `description` designate the types of expressions belonging to the object level.

| | |
|---|---|
| `description` | TYPE |

| | |
|---|---|
| `void-type` | description |
| `integer-type` | description |
| `character-type` | description |
| `string-type` | description |
| `boolean-type` | description |
| `token-type` | description |
| `entity-type` | description |
| `contour-type` | description |
| `environment-type` | description |
| `expression-type` | description |
| `description-type` | description |

The ground types are made available via variables for convenience, though they would otherwise still be accessible via quasiquotation.

| | |
|---|---|
| (`equiv-types?` *t1*: description *t2*: description) | boolean |

Return true if types *t1* and *t2* are equivalent, else return false.

| | |
|---|---|
| (`void-type?` *ty*: description) | boolean |
| (`integer-type?` *ty*: description) | boolean |
| (`character-type?` *ty*: description) | boolean |
| (`string-type?` *ty*: description) | boolean |
| (`boolean-type?` *ty*: description) | boolean |
| (`token-type?` *ty*: description) | boolean |
| (`entity-type?` *ty*: description) | boolean |
| (`contour-type?` *ty*: description) | boolean |
| (`environment-type?` *ty*: description) | boolean |
| (`expression-type?` *ty*: description) | boolean |

| | |
|---|---|
| (`description-type?` *ty*: description) | boolean |
| (`function-type?` *ty*: description) | boolean |
| (`function-type-argtypes` *ty*: description) | list-of(description) |
| (`function-type-resulttype` *ty*: description) | description |
| (`reference-type?` *ty*: description) | boolean |
| (`reference-type-basetype` *ty*: description) | description |
| (`vector-type?` *ty*: description) | boolean |
| (`vector-type-elttype` *ty*: description) | description |
| (`list-type?` *ty*: description) | boolean |
| (`list-type-elttype` *ty*: description) | description |
| (`tuple-type?` *ty*: description) | boolean |
| (`tuple-type-elttypes` *ty*: description) | list-of(description) |
| (`record-type?` *ty*: description) | boolean |
| (`record-type-fieldnames` *ty*: description) | list-of(token) |
| (`record-type-fieldtypes` *ty*: description) | list-of(description) |
| (`union-type?` *ty*: description) | boolean |
| (`union-type-varnames` *ty*: description) | list-of(token) |
| (`union-type-vartypes` *ty*: description) | list-of(description) |
| (`option-type?` *ty*: description) | boolean |
| (`option-type-basetype` *ty*: description) | description |

The predicates *type*-`type?` test the type for membership in the type or family of constructed types indicated. Operations for extracting the attributes of each constructed type are provided. The `description` values associated with these type can be constructed using quasiquotation, so functions are not provided for this purpose. Note that the attributes of record and union types refer to the components or variants visible. Other equivalent descriptions may have more or fewer visible components.

(`new-type` *attrty*:TYPE *equiv?*:EXPR *print*:EXPR) EXPR

Introduces a new class of constructed types. The type *attrty* is the type of a value that represents the attributes of the types that are instances of the new constructed type, e.g. the element type of a vector type or the component names and types of a record type. The expression *equiv?* must be a predicate on two arguments of type *attrty* that returns true if the types possessing those attributes are to be considered equivalent. For example, for a record type, it might check that corresponding field types are equivalent. The expression *print* must denote a function from one argument of type *attrty* to a result of type `void`, and should display a suitable printed representation of the type when invoked. This function is used by the command interpreter when displaying the type of a value of the new type. (There is unfortunately no way in XL to specify a display function for the *values* themselves.)

The result is a tuple of five functions: *create*, *test*, *attributes*, *seal*, and *unseal*. The names are shown in italics to indicate that they are for documentation purposes only, and in fact are anonymous functional values.

(*create refines*: description *attrs*: *attrty*) description
(*test ty*: description) boolean
(*attributes ty*: description) *attrty*
(*seal exp*: expression *attrs*: *attrty*) expression
(*unseal exp*: expression) expression

The function *create* creates a new instance type of the constructed type, whose instances are to be represented by objects of the type *refines*. The predicate *test* returns true if its argument was created by a call to *create*. The function *attributes* recovers the attribute value passed to *create*. It is a run-time error to invoke *attributes* on a type not produced by *create*.

Two types created by user-defined constructors are equivalent if and only if they were created by the same *create* function, yield true when passed to the *equiv?* function, and were refined from equivalent types. In this way, creation of new types can add further constraint to the usage of a type, but cannot remove constraints imposed by the type in terms of which it is defined, i.e. *refines*.

In order to allow manipulation of values of user-defined types, XL provides a type-safe means for viewing these values as instances of the type in terms of which the user-defined type was defined. The function *seal* maps an expression to an otherwise identical expression in which the type has been replaced by a refinement of its prior type using *create* with the given attribute value. The function *unseal* strips off one such refinement and returns the modified expression. Thus for any expression *e* and attribute *a*, (*unseal* (*seal e a*)) = *e*, and (expr-type (*unseal* (*seal e a*))) = (expr-type *e*).

(name->*type *name*: token) *type

Unquote a token as a type name, that is, return a *type that when applied to an environment returns the type value associated with *name*.

(close-type *type*: *type *env*: environment *vars*: list-of(token)) *type
(close-type-list *type*: list-of(*type) *env*: environment *vars*: list-of(token)) list-of(*type)

The function close-type partially closes a type in a given environment, leaving it open with respect to the variables enumerated in *vars*. The function close-type-list just maps close-type over a list of *type values.

### 7.10.2 Miscellaneous

(`write-char` *ch*: character) void
(`write-string` *str*: string) void
(`write-integer` *int*: integer) void
(`write-description` *desc*: description) void

Writes an object of the given type to the terminal. The function `write-description` is particularly useful in writing display functions for new types.

(`error` *msg*: string) void
(`type-error` *expect*: description *found*: description *msg*: string) void

When executed, `error` causes an error message to be generated on the interactive display. The error message includes the given string *msg*, as well as an indication of the phylum and form associated with the most recent *functional* semantic value invoked. Note that error contexts follow dynamic, not lexical, scoping rules.

The `type-error` function produces a somewhat more informative error message for situations in which a particular type is expected and another one is found. The message string in this case is appended to "`Type mismatch` " for display.

# 8 Using semantic macros

To demonstrate the utility of semantic macros, we present four illustrative examples. While the semantic macro facility can be used in casual, ad-hoc ways specific to an application program, these examples all represent significant language extensions of the type that would compose an extension library. Each is intended to serve as an exemplar of a larger family of related definitions illustrating the metaprogramming techniques involved.

The first example is a definition of a record type with named components, implemented as a refinement of the built-in tuple type. The second example is a control structure in which user-defined actions are performed upon the transitions of a finite state machine. The third example implements an iterator data type and its associated control structure. The fourth and final example is perhaps the most interesting, introducing a new named entity: a handle providing access to a cached resource over a region of the program where the resource is guaranteed to be in a consistent state. This example shows most clearly how type-specific special forms can be used to maintain program invariants that cannot be guaranteed when ordinary functions are used to provide ADT operations.

## 8.1 A heterogeneous data type

In this section, we present the complete XL definition of the data type generator `structure-of`, defining a family of types called *structures* whose values are heterogeneous tuples of named components. Unlike the built-in tuple type, structure components have symbolic labels, and unlike the built-in record type, structures obey a structural type-equivalence rule. Two structure types are considered equivalent if they possess the same number of fields, with the same names, in the same order, and of pairwise equivalent types.

In the example below, we define a type representing a point in planar Cartesian coordinates and a function computing the point at a specified distance above another.

```
(define
  (type
       [point (structure [x integer] [y integer])])
  (val
       [my-position (make-structure [x 10] [y 15])])
  (fun
       [move-up ([start point] [distance integer]) point
         (make-structure [x (structure-ref x start)]
                         [y (+ (structure-ref y start) distance)])]))
```

We begin the definition with a type `struct-attr`, the values of which will represent the compile-time attributes of the generated structure types. These attributes consist of a list of `token` values designating the component names, and a parallel list of `description` values designating their corresponding types. We also define a few auxiliary functions to make manipulations of the `struct-attr` tuples a little more perspicuous.

```
(meta-define
  (type
       [fieldnames  (list-of token)]

       [fieldtypes  (list-of description)]

       [struct-attr (tuple-of fieldnames fieldtypes)])
  (fun
       [make-struct-attr ([fn fieldnames] [ft fieldtypes]) struct-attr
         (make-tuple fn ft)]
```

```
    [get-fieldnames ([a struct-attr]) fieldnames
      (tuple-ref 0 a)]

    [get-fieldtypes ([a struct-attr]) fieldtypes
      (tuple-ref 1 a)]))
```

We must define the type equivalence predicate on structure types as a predicate on their associated `struct-attr` values. We require that corresponding fields have the same name and equivalent types. We also provide a display function that will be used by the XL system when displaying the new type, e.g. when reporting the type of an object displayed by the `eval` command.

```
(meta-define
  (fun [struct-equiv? ([a1 struct-attr] [a2 struct-attr]) boolean
       (let ((fun [eq-aux ([fn1 fieldnames] [fn2 fieldnames]
                           [ft1 fieldtypes] [ft2 fieldtypes]) boolean
                    (if (null? fn1)
                        (null? fn2)
                      (if (null? fn2)
                          #f
                        (and (eq? (car fn1) (car fn2))
                             (equiv-types? (car ft1) (car ft2))
                             (eq-aux (cdr fn1) (cdr fn2)
                                     (cdr ft1) (cdr ft2)))))]))
         (eq-aux (get-fieldnames a1) (get-fieldnames a2)
                 (get-fieldtypes a1) (get-fieldtypes a2)))])

   (fun [struct-print ([a struct-attr]) void
          (let ((fun [tp-aux ([fn fieldnames] [ft fieldtypes]) void
                       (unless (null? fn)
                         (write-string " [")
                         (write-string (token->string (car fn)))
                         (write-string " ")
                         (write-description (car ft))
                         (write-string "]")
                         (tp-aux (cdr fn) (cdr ft)))]))
            (write-string "(STRUCTURE-OF")
            (tp-aux (get-fieldnames a) (get-fieldtypes a))
            (write-string ")"))]))
```

We now invoke `new-type` to create the set of functions needed to create and use type descriptors for structure types. This action also "registers" the above definitions with the XL type-checker, which can now type-check function applications and other generic operations involving the values of structure types.

```
(meta-define
  (val
     [newtype (new-type struct-attr struct-equiv? struct-print)])
  (val
     [make-struct-type       (tuple-ref 0 newtype)]
     [struct-type?           (tuple-ref 1 newtype)]
     [struct-attributes      (tuple-ref 2 newtype)]
     [struct-seal            (tuple-ref 3 newtype)]
     [struct-unseal          (tuple-ref 4 newtype)]))
```

The structure type constructor `structure-of` requires an auxiliary phylum to represent the list of fields. The semantic value of this phylum is a tuple of two parallel lists giving their names and types.

```
(meta-define
   (type
        [*struct-field (tuple-of token *type)]

        [struct-field-list (list-of *struct-field)])
   (fun
        [make-struct-field ([fn token] [ft *type]) *struct-field
          (make-tuple fn ft)]

        [struct-field-name ([sf *struct-field]) token
          (tuple-ref 0 sf)]

        [struct-field-type ([sf *struct-field]) *type
          (tuple-ref 1 sf)]))

(phylum STRUCT-FIELD *struct-field)

(syntax STRUCT-FIELD [ [fieldname NAME] [fieldtype TYPE] ]
  (make-struct-field fieldname fieldtype))
```

With these preliminaries out of the way, we can now define the structure type constructor itself.

```
(syntax TYPE (structure-of & [fields STRUCT-FIELD])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((fun [mkfields ([fields struct-field-list]
                            [fnames fieldnames]
                            [ftypes fieldtypes]) description
```

```
                    (if (null? fields)
                        (make-struct-type
                         ((TYPE (tuple-of @@(reverse ftypes))) denv)
                         (make-struct-attr (reverse fnames)
                                           (reverse ftypes)))
                        (let ((val [f (car fields)])
                              (val [fname (struct-field-name f)]
                                   [ftype ((struct-field-type f) env)]))
                          (mkfields (cdr fields)
                                    (cons fname fnames)
                                    (cons ftype ftypes))))]))
      (mkfields fields (empty-list fieldnames) (empty-list fieldtypes))))))
```

To create values of a structure type, we introduce a structure value constructor **make-structure**. Given a list of field names and values, this form yields a value of the appropriate structure type with the indicated field names and values. Again, we need some preliminary definitions, this time for the list of component values and their names.

```
(meta-define
  (type
       [*struct-init (tuple-of token *expr)]

       [struct-init-list (list-of *struct-init)])
  (fun
       [make-struct-init ([fn token] [fi *expr]) *struct-init
         (make-tuple fn fi)]

       [struct-init-name ([si *struct-init]) token
         (tuple-ref 0 si)]

       [struct-init-expr ([si *struct-init]) *expr
         (tuple-ref 1 si)]))

(phylum STRUCT-INIT *struct-init)

(syntax STRUCT-INIT [ [fieldname NAME] [fieldval EXPR] ]
  (make-struct-init fieldname fieldval))
```

Structure values are represented as tuples. The function **struct-seal**, obtained from the call to **new-type**, labels the tuple-valued expression representing the structure value as possessing a structure type with the indicated attributes.

```
(syntax EXPR (make-structure & [fields STRUCT-INIT])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((fun [dofields ([f struct-init-list]
                            [fnames (list-of token)]
                            [fvals  (list-of expression)]
                            [ftypes (list-of description)]) expression
                   (if (null? f)
                       (struct-seal
                        ((EXPR (make-tuple @@(reverse fvals))) denv)
                        (make-struct-attr (reverse fnames)
                                          (reverse ftypes)))
                       (let ((val [fname (struct-init-name (car f))]
                                  [fval  ((struct-init-expr (car f)) env)])
                             (val [ftype (expr-type fval)]))
                         (dofields (cdr f)
                                   (cons fname fnames)
                                   (cons fval fvals)
                                   (cons ftype ftypes))))]))
        (dofields fields
                  (empty-list token)
                  (empty-list expression)
                  (empty-list description))))))
```

The structure reference operator unseals the structure-valued expression with `struct-unseal`, revealing the underlying tuple value, and uses the structure field name list to determine which tuple element to select.

```
(meta-define
  (fun [field-index ([fn token] [rt description]) integer
         (let ((fun [f-i-aux ([idx integer] [fnames fieldnames]) integer
                      (begin
                       (when (null? fnames)
                         (error (string-append
                                 "Attempt to select nonexistent field - "
                                 (token->string fn))))
                       (if (eq? fn (car fnames))
                           idx
                         (f-i-aux (+ idx 1) (cdr fnames))))]))
           (f-i-aux 0 (get-fieldnames (struct-attributes rt))))]))
```

```
(syntax EXPR (structure-ref [fn NAME] [structval EXPR])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((val [sval  (structval env)])
            (val [stype (expr-type sval)]))
        (unless (struct-type? stype)
          (error "Attempt to apply STRUCTURE-REF to non-structure"))
        ((EXPR (tuple-ref ,(field-index fn stype)
                          ,,(struct-unseal sval))) denv)))))
```

This completes the definition of structure types. A completely satisfactory definition would encapsulate the various auxiliary definitions, particularly the `struct-seal` and `struct-unseal` functions. We wish only to augment the existing phyla TYPE and EXPR. As remarked earlier, however, XL in its present form does not address modularity concerns.

## 8.2 A fancy control structure

Finite state machines (FSMs) are a convenient means to express many computations. In this example, we implement a control structure in which user-defined actions, represented by arbitrary XL expressions, may be executed as transitions are taken along the arcs of an FSM. The FSM is specified by a straightforward enumeration of its states, its arcs, and the labels and actions associated with the arcs. For example, the following FSM recognizes strings over the alphabet $\{a, b, c\}$ matching the regular expression $ab^*c$, where the functions **succeed** and **fail** signal success and failure respectively:

```
(fsm character char=? s1 next-char
     (s1
          ((#\a)       s2)
          ((#\b #\c)   (exit)  (fail)))
     (s2
          ((#\b)       s2)
          ((#\a)       (exit)  (fail))
          ((#\c)       (exit)  (succeed))))
```

The first line indicates that the arc labels are of type `character`, for which `char=?` is to be used as the equality predicate, that the initial state is `s1`, and that the nullary function `next-char` (not defined here) fetches successive characters from the input stream. The form `(exit)` denotes a distinguished final state, a transition to which causes the FSM to exit returning the value `*void*`. Multiple actions are permitted, in which case they form an implicit `begin`. All actions must have type void. A run-time error will be reported if there is no valid transition for the current state and input symbol.

Our implementation strategy will be to translate the FSM into a set of mutually tail-recursive functions, one for each state, as shown below:

```
(let ((val [%next ⟨next-symbol⟩]
           [%eqpred ⟨equality-predicate⟩])
      (fun [⟨state-name⟩ ([%item ⟨symbol-type⟩]) void
             (if (or (%eqpred %item ⟨label⟩) ... more labels ...)
                 (begin ⟨arc-action⟩ ... ⟨go-next⟩)
                 (if ... more arcs ...
                     (error "No applicable FSM transition")))]
           ... more states ...))
  (⟨start-state⟩ (%next)))
```

where ⟨go-next⟩ is "(⟨next-state⟩ (%next))" for a named state, or "*void*" for an (exit). It is apparent from this translation scheme that the arc labels will be tested in sequential order. While the user should be discouraged from using expressions other than literals as arc labels, to require this within the existing framework would restrict the symbol type to one of the built-in types for which literals are defined.[9]

We will avoid inadvertent capture of names appearing free in the arc actions by closing each action expression in the proper environment. We must be careful, however, that state names in the macro expansion do not conflict with the various names prefixed with "%" above. Our strategy will be to map each user-supplied state name to a newly-created unique token. The mapping process will also provide a convenient way to check that there are no duplicate state labels, and that every label appearing as the next state in an arc is in fact the label of some state. A tokenmap is a function that when applied to a token *from* yields an option which contains the token *to* to which it maps, or is empty if no mapping is defined. The function map-token takes a token map and a new pair of tokens *from* and *to* and returns an updated map.

```
(meta-define
  (type
       [tokenmap  (fun (token) (option-of token))])
  (val
       [empty-map (lambda ([tok token]) (empty-option token))])
```

[9]To avoid this difficulty would require introducing a distinction, not made in XL, between static (compile-time) and non-static (run-time) expressions.

```
(fun
     [map-token ([from token] [to token] [tmap tokenmap]) tokenmap
       (lambda ([tok token])
         (if (eq? tok from) (make-option to) (tmap tok)))]))
```

The arc labels are a list of expressions of the FSM symbol type. The semantic value for the ARC-LABELS phylum takes the environment in which the labels are to be evaluated and the FSM symbol type and returns an `expression` which evaluates to `#t` at run-time if any of the arc labels match the input.

```
(phylum ARC-LABELS (fun (environment description) *expr))

(syntax ARC-LABELS (& [labels EXPR])
  (lambda ([env    environment]
           [symty  description])
    (let ((fun [loop ([labs  (list-of *expr)]
                      [tests (list-of *expr)]) *expr
                 (if (null? labs)
                     (EXPR (or @(reverse tests)))
                     (let ((val [sym ((car labs) env)]))
                       (unless (equiv-types? symty (expr-type sym))
                         (type-error symty (expr-type sym) "in FSM arc label"))
                       (loop (cdr labs)
                             (cons (EXPR (%eqpred %item ,,sym)) tests))))]))
      (loop labels (empty-list *expr)))))
```

Each arc specifies a next state, which is either a state name or the distinguished final state `(exit)`. Note the use of smap, the state map.

```
(phylum FSM-NEXTSTATE (fun (tokenmap) *expr))

(atom-syntax FSM-NEXTSTATE [state NAME]
  (lambda ([smap tokenmap])
    (let ((val [ostate (smap state)]))
      (when (empty? ostate)
        (error
         (string-append "Undefined next FSM state - " (token->string state))))
      (EXPR (,(name->*expr (value ostate)) (%next))))))

(syntax FSM-NEXTSTATE (exit)
  (lambda ([smap tokenmap])
    (EXPR *void*)))
```

The semantic value for phylum FSM-ARC takes an environment in which the state labels and actions are to be closed, a state map, and the symbol type, and yields a "wrapper" function which will later generate the code for the arc. The wrapper will generate the expression

```
    (if (or (eq? ...) ...) (begin actions ...) alt)
```

when applied to an expression *alt*. The wrappers for each state will be composed to generate the body of the state's transition function.

```
(meta-define
  (type
        [arc-wrapper (fun (*expr) *expr)]

        [*fsm-arc    (fun (environment tokenmap description) arc-wrapper)]))

(phylum FSM-ARC *fsm-arc)

(syntax FSM-ARC ([labels ARC-LABELS] [nstate FSM-NEXTSTATE] & [actions EXPR])
  (lambda ([env    environment]
           [smap   tokenmap]
           [symty  description])
    (let ((val [test    (labels env symty)])
          (fun [do-arcs ([actions (list-of *expr)]
                         [actcode (list-of expression)]) *expr
                (if (null? actions)
                    (EXPR (begin @@(reverse actcode) ,(nstate smap)))
                    (let ((val [action ((car actions) env)]))
                      (unless (equiv-types? void-type (expr-type action))
                        (error "FSM arc action must yield VOID"))
                      (do-arcs (cdr actions) (cons action actcode))))]))
      (lambda ([alt *expr])
        (EXPR (if ,test ,(do-arcs actions (empty-list expression)) ,alt))))))
```

The semantic value for phylum FSM-STATE takes the environment and symbol type as before, to be passed on to FSM-ARC. A pair of values is returned: the state label and a suspension, a function of type `stategen`, that completes the processing of the state when applied to a state map. This suspension is necessary because we must collect all of the labels and build the state map before generating any of the functions for the states.

Suspensions are a standard trick in XL metaprogramming for cases in which insufficient information is available when needed — just abstract the offending expression with respect to the unknown variables and pass it on until the information becomes available. This technique can be used in many cases where it might appear at first glance that multiple passes are required.

```
(meta-define
 (type
      [stategen   (fun (tokenmap) *funbind)]

      [*fsm-state (fun (environment description) (tuple-of token stategen))]))

(phylum FSM-STATE *fsm-state)

(syntax FSM-STATE ([label NAME] & [arcs FSM-ARC])
  (lambda ([env   environment]
           [symty description])
    (make-tuple
     label
     (lambda ([smap tokenmap])
       (let ((fun [do-arcs ([arcs (list-of *fsm-arc)]) *expr
                    (if (null? arcs)
                        (EXPR (error "No applicable FSM transition"))
                        (((car arcs) env smap symty) (do-arcs (cdr arcs))))]))
         (let ((val [state   (value (smap label))]))
           (FUNBIND [,state ([%item ,,symty]) void
                             ,(do-arcs arcs default)])))))))
```

The processing required for the top-level FSM form itself is rather involved, so we factor out a pair of auxiliary functions. The function `collect-states` processes a list of FSM-STATE instances to obtain a list of the `stategen` functions. It also constructs a state map in which each FSM state label is mapped to a unique new token, and checks for duplicate state labels. The function `check-types` verifies that the types of the equality predicate and the next symbol function are correct given the symbol type.

```
(meta-define
  (fun [collect-states ([states (list-of *fsm-state)]
                        [env    environment]
                        [symty  description])
                       (tuple-of (list-of stategen) tokenmap)
         (let ((fun [loop ([states (list-of *fsm-state)]
                           [smap   tokenmap]
                           [genfns (list-of stategen)])
                          (tuple-of (list-of stategen) tokenmap)
```

```
                    (if (null? states)
                        (make-tuple genfns smap)
                        (let ((val [state ((car states) env symty)])
                              (val [label (tuple-ref 0 state)]
                                   [genfn (tuple-ref 1 state)]))
                          (unless (empty? (smap label))
                            (error (string-append "Duplicate FSM state - "
                                                  (token->string label))))
                          (loop (cdr states)
                                (map-token label (generate-token "S") smap)
                                (cons genfn genfns))))]))
          (loop states empty-map (empty-list stategen)))]))

(meta-define
  (fun [check-types ([stype  description]
                     [eqpred expression]
                     [next   expression]
                     [denv   environment]) void
       (let ((val [eqpred-type ((TYPE (fun (,,stype ,,stype) boolean)) denv)]
                  [next-type   ((TYPE (fun () ,,stype)) denv)]))
         (unless (equiv-types? eqpred-type (expr-type eqpred))
           (type-error eqpred-type (expr-type eqpred)
                       "in FSM equality predicate"))
         (unless (equiv-types? next-type (expr-type next))
           (type-error next-type (expr-type next)
                       "in FSM next-token function")))]))
```

The fsm construct first verifies that the equality predicate equalp and the next symbol function getnext agree with the declared symbol type symtype. The states are then processed, by collect-states to accumulate the stategen functions and the state map. Each stategen function is then invoked to yield the code for each state, and the resulting function bindings are incorporated into a let form binding the equality predicate and next symbol functions, and invoking the start state's transition function.

```
(syntax EXPR (fsm [symtype TYPE] [equalp EXPR]
                  [start NAME] [getnext EXPR] & [states FSM-STATE])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((val [stype  (symtype env)])
            (val [eqpred (equalp  env)])
            (val [next   (getnext env)]))
(check-types stype eqpred next denv)
(let ((val [stateinfo (collect-states states env stype)])
      (val [genfuns   (tuple-ref 0 stateinfo)]
           [statemap  (tuple-ref 1 stateinfo)])
      (fun [loop ([genfns (list-of stategen)]
                  [fbinds (list-of *funbind)]) expression
             (if (null? genfns)
                 (let ((val [ostate (statemap start)]))
                   (when (empty? ostate)
                     (error (string-append "Invalid FSM start state - "
                                           (token->string start))))
                   ((EXPR (let ((val [%next ,,next]
                                     [%eqpred ,,eqpred])
                                (fun @(reverse fbinds)))
                            (,(name->*expr (value ostate))(%next))))
                    denv))
                 (loop (cdr genfns)
                       (cons ((car genfns) statemap) fbinds)))]))
  (loop genfuns (empty-list *funbind))))))
```

## 8.3 Iterators

An iterator is a function-like object that yields not a single value, but a sequence of values generated on demand. Our formulation of iterators follows roughly that of CLU [15].

An iterator is constructed with the `iterator` special form. The argument list is identical in form to that of a `lambda` expression, but the result type must be explicitly indicated. The body of an iterator is an expression of type void. The body returns each value with the `yield` form, which may appear only within an iterator. To terminate the sequence of values, the body returns normally. An iterator is invoked with the `for-each` form, which repeatedly evaluates its body with successive values yielded by the iterator bound to a variable.

A simple use of iterators mimics the Pascal for loop:

```
(define
  (val [up-to  (iterator ([start integer] [end integer]) integer
                 (let ((fun [loop ([i integer]) void
                              (unless (> i end)
                                (yield i)
                                (loop (+ i 1)))]))
                   (loop start)))]))
```

The iterator call

```
(for-each x (up-to 1 5)
  (begin
    (write-integer x)
    (write-string " ")))
```

displays "`1 2 3 4 5 `". A more unusual iterator enumerates the characters of a string in left-to-right order:

```
(define
  (val [in-string (iterator ([str string]) character
                   (let ((val [strlen (string-length str)])
                         (fun [loop ([i integer]) void
                                (when (< i strlen)
                                  (yield (string-ref str i))
                                  (loop (+ i 1)))]))
                     (loop 0)))]))
```

The invocation

```
(for-each x (in-string "foobar")
  (begin
    (write-char x)
    (write-string " ")))
```

displays "`f o o b a r `". More exotic applications include traversals of trees and other complex data structures. Iterators allow an abstract treatment of iteration over the components of an abstract data type, permitting us to hide its representation. They also allow us to represent "infinite" sequences as manipulable data.

Our strategy for compiling iterators is to represent them as functions taking an extra hidden functional argument. When the iterator is called, the body of the `for-each` will be abstracted with respect to the bound variable and supplied as the value of this parameter. To yield a value, the iterator need only invoke this function. The following transformations illustrate the method:

```
(iterator (⟨args⟩ ...) ⟨resulttype⟩
    ... (yield ⟨result⟩) ...)
  ⟹
(lambda ([%yield (fun (⟨resulttype⟩) void)] ⟨args⟩ ...)
    ... (%yield ⟨result⟩) ...)

(for-each ⟨var⟩ (⟨iter-exp⟩ ⟨args⟩ ...) ⟨body⟩)
  ⟹
(⟨iter-exp⟩ (lambda ([⟨var⟩ ⟨resulttype⟩]) ⟨body⟩) ⟨args⟩ ...)
```

In the actual expansion, a unique token generated by `generate-token` will replace the name `%yield` above to prevent naming conflicts.

We begin the definition of iterators with the type `iter`, analogous to the `fun` type possessed by functions. The attributes of an iterator type include its result type and the types of its arguments.

```
(meta-define
  (type
      [iter-attrs (structure-of [restype  description]
                                [argtypes (list-of description)])]))
```

Two iterator types are equivalent if their result types are equivalent and if the types of corresponding arguments are equivalent.

```
(meta-define
  (fun
      [iter-equiv? ([a1 iter-attrs] [a2 iter-attrs]) boolean
        (let ((fun [eq-aux ([ats1 (list-of description)]
                            [ats2 (list-of description)]) boolean
                     (if (null? ats1)
                         (null? ats2)
                         (if (null? ats2)
                             #f
                             (and (equiv-types? (car ats1) (car ats2))
                                  (eq-aux (cdr ats1) (cdr ats2)))))]))
          (and (eq-aux (structure-ref argtypes a1)
                       (structure-ref argtypes a2))
               (equiv-types? (structure-ref restype a1)
                             (structure-ref restype a2))))]))
```

As for any new type, we need a function that prints a human-readable representation of the type descriptor. Iterator types are displayed in the form `(iter (`*argtypes...*`)` *resulttype*`)`.

```
(meta-define
  (fun
      [iter-print ([a iter-attrs]) void
        (let ((fun [ip-aux ([ats (list-of description)]) void
                     (unless (null? ats)
                       (write-description (car ats))
                       (unless (null? (cdr ats))
                         (write-string " "))
                       (ip-aux (cdr ats)))]))
```

```
          (write-string "(ITER (")
          (ip-aux (structure-ref argtypes a))
          (write-string ") ")
          (write-description (structure-ref restype a))
          (write-string ")"))]))
```

We now invoke new-type to create the new type constructor and its operations.

```
(meta-define
  (val
     [newtype (new-type iter-attrs iter-equiv? iter-print)])
  (val
     [iter-create        (tuple-ref 0 newtype)]
     [iter-type?         (tuple-ref 1 newtype)]
     [iter-attributes    (tuple-ref 2 newtype)]
     [iter-seal          (tuple-ref 3 newtype)]
     [iter-unseal        (tuple-ref 4 newtype)]))
```

The yield form, permitted only within the body of an iterator construct, must have some way to determine its context. All communication between distinct EXPRs must take place via the environment. We extend the type environment, an extensible record type, to include a new component iter-yield. In an environment representing the context within an iterator body, this component, an option, will be non-empty and will have a value of type token. This token is the name of the function that the expansion of yield will call to return a value from the iterator. We hide the representation of iterator contexts with the auxiliary functions yield-context and yield-function, which update and query environments in a style compatible with the primitives for accessing their bindings component.

```
(meta-define
  (type [iter-body-env
         (extend-record environment [iter-yield (empty-option token)])])

  (fun  [yield-context ([yname token] [env environment]) environment
         (update iter-body-env env [iter-yield (make-option yname)])]

        [yield-function ([env environment]) (option-of token)
         (select iter-body-env iter-yield env)]))
```

The iterator type constructor iter offers no surprises.

```
(meta-define
  (type [*it-type-list (fun (environment) (list-of description))]))

(phylum IT-TYPE-LIST *it-type-list)

(syntax IT-TYPE-LIST (& [types TYPE])
  (lambda ([env environment])
    (let ((fun [loop ([itypes (list-of *type)]
                      [descrs (list-of description)])
                     (list-of description)
                 (if (null? itypes)
                     (reverse descrs)
                     (loop (cdr itypes)
                           (cons ((car itypes) env) descrs)))]))
      (loop types (empty-list description)))))

(syntax TYPE (iter [argty IT-TYPE-LIST] [resty TYPE])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((val [argtypes (argty env)]
                 [restype  (resty env)])
            (val [refines  ((TYPE (fun (,,restype @@argtypes) void)) denv)]
                 [attr     (make-structure [restype restype]
                                           [argtypes argtypes])]))
        (iter-create refines attr)))))
```

The parameters of the iterator form are represented by the phyla IPARM and IPARM-LIST. No semantic processing is done at this time. We just collect the parameter names and types for later examination.

```
(meta-define  (type [*iparm (tuple-of token *type)]))

(phylum IPARM *iparm)

(syntax IPARM [ [n NAME] [t TYPE] ]  (make-tuple n t))

(phylum IPARM-LIST (list-of *iparm))

(syntax IPARM-LIST (& [ip IPARM]) ip)
```

We must verify that parameter names are not duplicated in iterator parameter lists. The function member? tests a token for membership in a list of tokens.

```
(meta-define
  (fun  [member? ([tok token] [lst (list-of token)]) boolean
         (if (null? lst)
             #f
             (if (eq? tok (car lst))
                 #t
                 (member? tok (cdr lst))))]))
```

The auxiliary function `make-fparm-list` processes a list of iterator parameters, constructing the list of parameters for the `lambda` expression and checking for duplicate parameter names. The parameter types are closed in the given environment.

```
(meta-define
  (fun [make-fparm-list ([env    environment]
                         [iparms (list-of *iparm)]
                         [pnames (list-of token)]
                         [fparms (list-of *fparm)]
                         [itypes (list-of description)])
                        (tuple-of (list-of *fparm)
                                  (list-of token)
                                  (list-of description))
         (if (null? iparms)
             (make-tuple (reverse fparms) (reverse pnames) (reverse itypes))
             (let ((val [iname (tuple-ref 0 (car iparms))]
                        [itype ((tuple-ref 1 (car iparms)) env)]))
               (when (member? iname pnames)
                 (error (string-append
                        "Duplicate name in iterator parameter list - "
                        (token->string iname))))
               (let ((val [fparm (FPARM [,iname ,,itype])]))
                 (make-fparm-list env
                                  (cdr iparms)
                                  (cons iname pnames)
                                  (cons fparm fparms)
                                  (cons itype itypes)))))]))
```

Most of the `iterator` definition is straightforward, passing the buck to `make-fparm-list`. The construction of the environment for the iterator body is somewhat tricky, however. A name yname is invented to represent the yield function, i.e. the first parameter of the `lambda` expression comprising the expansion. We capture the expansion-time environment inside that `lambda` expression with the form "`(lambda ([e ...]) ...)`." This environment is then augmented with the yield context, indicating the yield function to be used, before being passed to the body. Note that it would not have been correct to simply replace "denv" with "(yield-context yname env)," as "*void*" would then be closed in the wrong environment. Observe also that the types appearing within fparms have already been reduced, otherwise they would also be closed incorrectly.

```
(syntax EXPR (iterator [parms IPARM-LIST] [rtype TYPE] [body EXPR])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((val [resty  (rtype env)])
            (val [ytype  ((TYPE (fun (,,resty) void)) denv)])
            (val [ptemp  (make-fparm-list
                          env parms (empty-list token)
                          (empty-list *fparm) (empty-list description))])
            (val [fparms (tuple-ref 0 ptemp)]
                 [fnames (tuple-ref 1 ptemp)]
                 [ftypes (tuple-ref 2 ptemp)])
            (val [yname  (generate-token "YIELD")]))
        (iter-seal ((EXPR (lambda ([,yname ,,ytype] @fparms)
                            ,(lambda ([e environment])
                               (body (yield-context yname e)))
                            *void*))
                    denv)
                   (make-structure [restype resty]
                                   [argtypes ftypes]))))))
```

The yield form simply invokes the yield function indicated by the current yield context, or reports an error if the context is empty.

```
(syntax EXPR (yield [e expr])
  (lambda ([env environment])
    (let ((val [yield (yield-function env)]))
      (when (empty? yield)
        (error "YIELD not inside ITERATOR"))
      ((EXPR (,(name->*expr (value yield)) ,,(e env))) env))))
```

The for-each form just passes on the bound variable and body to the semantic value of the iterator call, a subform of the for-each. Most of the work here is collecting and closing (reducing) the argument expressions. Note the use of the same trick seen in iterator to capture the expansion-time environment after it has been augmented with additional bindings in the expansion. Here we check that the for-each body, evaluated in the context of the bound variable *var*, has type void.

```
(phylum ITER-CALL (fun (environment token *expr) expression))

(syntax ITER-CALL ([iter EXPR] & [args EXPR])
  (lambda ([env environment] [var token] [body *expr])
    (let ((fun [do-args ([argexprs (list-of *expr)]
                         [argparms (list-of expression)]
                         [argtypes (list-of description)])
                        (list-of expression)
                 (if (null? argexprs)
                     (reverse argparms)
                     (let ((val [aparm ((car argexprs) env)]))
                       (unless (equiv-types? (car argtypes) (expr-type aparm))
                         (type-error (car argtypes) (expr-type aparm)
                                     "in argument to iterator"))
                       (do-args (cdr argexprs)
                                (cons aparm argparms)
                                (cdr argtypes))))])
          (val [ival     (iter env)])
          (val [itype    (expr-type ival)])
          (val [rtype    (structure-ref restype  (iter-attributes itype))]
               [argtys   (structure-ref argtypes (iter-attributes itype))])
          (val [argparms (do-args args (empty-list expression) argtys)]))
      ((EXPR (,,(iter-unseal ival)
              (lambda ([,var ,,rtype])
                ,(lambda ([e environment])
                   (let ((val [bexpr (body e)]))
                     (unless (equiv-types? (expr-type bexpr) void-type)
                       (error "FOR-EACH body type must be VOID"))
                     bexpr)))
              @@argparms))
       env))))

(syntax EXPR (for-each [var NAME] [icall ITER-CALL] [body EXPR])
  (lambda ([env environment])
    (icall env var body)))
```

## 8.4 Resource management

By restricting the use of a data type to a special binding form, we can guarantee that proper initialization and finalization actions precede and follow its use in the body of the binding form. This restriction is particularly useful for a type representing a handle on a resource that must be explicitly put into a consistent state before use and then released when no longer needed, e.g. file descriptors and cached disk-resident structures. In this example, we define a simple example of such a managed resource, the type vmspace. A value of type vmspace is an indexed collection of pages, each of a fixed size and consisting of integer-valued cells. Each cell is addressed by a ⟨*page, offset*⟩ pair. We assume the following definitions, eliding the details of their implementation with "...":

```
(define
  (type [vmspace  ... ]
        [handle   ... ]))

(define
  (fun [acquire ([space vmspace] [page integer]) handle
         ... ]
       [release ([hdl handle] [writeback? boolean]) void
         ... ]
       [handle-fetch ([hdl handle] [idx integer]) integer
         ... ]
       [handle-store ([hdl handle] [idx integer] [datum integer]) void
         ... ] ))
```

We are not concerned with how vmspace values are created, but we assume that the storage for the pages is allocated on disk. (XL in fact has no disk access primitives, but we ask the reader to use his imagination.) The function acquire, given a vmspace and a page number, returns a handle value, representing a buffer initialized with the contents of the page. The release function frees the buffer for re-use, writing its possibly-altered contents out to disk if writeback? is true. The functions handle-fetch and handle-store allow read and write access to the contents of the buffer.

We require that the following invariants be maintained:

- The handle argument to handle-fetch and handle-store must designate a properly initialized buffer, i.e. one returned from a call to acquire.
- Every handle must eventually be released. Ideally, a handle should remain allocated to a page only while a series of closely-spaced accesses are being performed, in order to minimize total buffer usage (space) while avoiding excessive buffer management overhead (time).
- If a buffer is modified, it must be written back out to disk before being released.

We would like to enforce these constraints statically, without run-time overhead. Our strategy will be to conceal the above definitions, except for the type vmspace itself, and make their functionality accessible only through new syntactic forms whose syntax and static semantics imply the satisfaction of the constraints. The key idea is to introduce a new form of declaration, a `handle` declaration, which introduces a local name through which the page buffer may be accessed in a restricted way by the special forms `fetch` and `store`. This name does not designate a value, but a new kind of entity representing an initialized page buffer. It is not possible to refer to the buffer outside of the scope of the name, hence it is safe to release the buffer before exiting that scope. The user expresses his intention to write into the buffer at the time that the handle is declared. It is a static semantic error to use `store` with a handle that has not been declared `(read-write)`. The following schemata illustrate the usage of these language constructs:

```
(let ((handle handle space pageno (read-only)))
  ...
  (fetch handle offset)
  ... )

(let ((handle handle space pageno (read-write)))
  ...
  (fetch handle offset)
  ...
  (store handle offset datum)
  ... )
```

We begin our language extension with some auxiliary definitions. We first capture the `description` value associated with the type `vmspace`. Values of type `handle-info` represent the semantic attributes of a handle entity: a flag indicating whether the handle is writable, and the name of a variable that will actually hold the buffer at runtime. The `entity` type, a union type, must also be extended to include handles.

```
(meta-define
  (val  [vmspace-type ((TYPE vmspace) (current-environment))]))

(meta-define
  (type [handle-info
         (structure-of [writable? boolean] [bufname token])]))

(meta-define
  (type [handle-entity (extend-union entity [handle handle-info])]))
```

The auxiliary phylum ACCESS represents the access mode for the handle. Its semantic value is a boolean, true if the buffer is to be written.

```
(phylum ACCESS boolean)

(syntax ACCESS (read-only)  #f)

(syntax ACCESS (read-write) #t)
```

The handle declaration wraps its scope in a `let` which binds a newly-generated variable to the buffer returned for the specified space and page number by `acquire`. The body is closed in an environment in which the handle name is bound to a handle entity. The generated code releases the buffer after the body is evaluated, writing the page out to disk if the handle was declared for read-write access.

```
(syntax DECL (handle [h NAME] [s EXPR] [p EXPR] [a ACCESS])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (let ((val [vmspace (s env)]))
        (unless (equiv-types? vmspace-type (expr-type vmspace))
          (type-error vmspace-type (expr-type vmspace)
                      "in handle declaration"))
        (let ((val [page (p env)]))
          (unless (equiv-types? integer-type (expr-type page))
            (type-error integer-type (expr-type page)
                        "in handle declaration"))
          (let ((val [h-name   (generate-token "HANDLE")])
                (val [h-info   (make-structure [writable? a]
                                               [bufname h-name])])
                (val [h-entity (inject handle-entity handle h-info)])
                (val [contour  (add-binding h h-entity (empty-contour))])
                (val [body-env (add-contour contour env)]))
            (lambda ([exp *expr])
              ((EXPR (let ((val [,h-name
                                 (acquire ,,vmspace ,,page)])
                           (val [%result
                                 ,(close-expr exp body-env (list h-name))]))
                       (release ,(name->*expr h-name) ,(boolean->*expr a))
                       %result))
               denv))))))))
```

The `fetch` form just invokes `handle-fetch` on the buffer after checking the validity of its handle argument and retrieving the buffer variable name (bound by the expansion of the handle declaration) from its semantic attributes. The `store` form is nearly identical, but invokes `handle-store` and verifies that the handle was declared writable.

```
(syntax EXPR (fetch [h NAME] [i EXPR])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (unless (bound-in-environment? h env)
        (error (string-append "Unbound handle name - " (token->string h))))
      (let ((val [entity (binding h env)]))
        (unless (is? handle-entity handle entity)
          (error (string-append "Handle name required - " (token->string h))))
        (let ((val [h-info (project handle-entity handle entity)])
              (val [h-name (structure-ref bufname h-info)])
              (val [index  (i env)]))
          (unless (equiv-types? integer-type (expr-type index))
            (type-error integer-type (expr-type index) "in FETCH index"))
          ((EXPR (handle-fetch ,(name->*expr h-name) ,,index))
           (merge-environments denv env (list h-name))))))))

(syntax EXPR (store [h NAME] [i EXPR] [d EXPR])
  (let ((val [denv (definition-environment)]))
    (lambda ([env environment])
      (unless (bound-in-environment? h env)
        (error (string-append "Unbound handle name - " (token->string h))))
      (let ((val [entity (binding h env)]))
        (unless (is? handle-entity handle entity)
          (error (string-append "Handle name required - " (token->string h))))
        (let ((val [h-info (project handle-entity handle entity)])
              (val [h-name (structure-ref bufname h-info)])
              (val [index  (i env)]))
          (unless (equiv-types? integer-type (expr-type index))
            (type-error integer-type (expr-type index) "in STORE index"))
          (let ((val [datum (d env)]))
            (unless (equiv-types? integer-type (expr-type datum))
              (type-error integer-type (expr-type datum) "in STORE datum"))
            (unless (structure-ref writable? h-info)
              (error "Attempt to STORE via read-only handle"))
            ((EXPR (handle-store ,(name->*expr h-name) ,,index ,,datum))
             (merge-environments denv env (list h-name)))))))))
```

Many variations on this theme are possible. A very simple example, which would be useful if XL supported multi-threaded execution, would be a construct for protecting critical regions. Given the synchronization primitives P and V, we might implement the following transformation, where ⟨t1⟩ and ⟨t2⟩ are temporary variables:

```
(critical-region ⟨semaphore⟩ ⟨body⟩)
  ⟹
(let ((val (⟨t1⟩ ⟨semaphore⟩)))
  (P ⟨t1⟩)
  (let ((val (⟨t2⟩ ⟨body⟩)))
    (V ⟨t1⟩)
    t2))
```

## 8.5 Embedded languages

The examples presented have all been extensions to the XL language. The mechanisms used, however, may be also be employed to implement translators for embedded languages quite different from their XL host, provided, of course, that XL suffices for expressing the translated result. Extensions largely preserve the character of the base language, as the immutable framework represented by the semantic value types of the predefined phyla must be respected by all extensions. Embedded languages, on the other hand, may do as they please.

Interesting possibilities include:

- A Prolog-like logic programming language that could interface smoothly with "evaluable predicates" written in the base language.
- A "patch-panel" language for describing digital signal processing algorithms, such as music synthesis.
- An embedded database query language. Note that the data model, including heterogeneous tuples, might be smoothly interfaced with the base language type system for convenient database access from application code.
- Embedded parser and lexical analyzer generators.

As XL is intended only as a vehicle to explore a novel approach to macro processing, we do not claim that it is suitable for all of these proposals. What we hope to have demonstrated is that mechanisms such as we have illustrated could in principle be used in these ways when incorporated into an "industrial strength" language and its compiler.

# 9 Remarks on the design of XL

In the design of XL, various design choices had to be evaluated and tradeoffs made, some in the interest of technical merit, and others as concessions to the requirement that the project be kept within a manageable scope. Here we pause to reflect on some of these choices, attempting to point out which were essential and which were incidental, with an eye toward identifying fruitful alternative choices that might serve as topics of further research.

## 9.1 Choice of metalanguage

XL serves as its own metalanguage. User-defined compile-time computations are described in the same language as run-time computations. This is quite appropriate in the case of XL, as XL is well-suited for the symbolic processing involved in metaprogramming. Using the same language for both purposes minimizes the number of additional concepts that must be added to the language to support metaprogramming, and assures that the expressive power of the metalanguage is as great as that of the object language. Note that Lisp takes this approach as well.

If we had chosen to use a more conventional object language, e.g. one similar to Pascal or C in its semantics, it would probably not have been satisfactory to use this same language as the metalanguage. Conventional imperative languages are not well-adapted to symbolic processing, and explicit storage management operations clutter up the code. Additionally, most conventional languages have insecurities, i.e. unchecked language rules, the violation of which can result in embarrassingly non-robust behavior such as core dumps. This behavior is not appropriate for a compiler, which should recover gracefully from all errors. (But note that it is possible for the compiler to go into an infinite loop in a metaprogram. There is not much we can do about this beyond providing adequate debugging facilities, and perhaps a timeout for unattended batch compilations.)

There is no essential connection between the basic structure of the metalanguage and the object language it manipulates. We could provide a different set of meta-types and their operations specific to another object language (especially the quasiquotation operators for the object language phyla) and write metaprograms over that language. Thus it would be possible to support a variety of object languages in an XL-like metaprogramming system using a common metalanguage.

## 9.2 Syntactic framework

The syntactic framework adopted for XL is a fully-parenthesized notation closely following that of Lisp. Such a notation allows easy extension without fear of introducing ambiguity or undue parsing complexity. Furthermore, by requiring a keyword to be associated with each syntactic form, we can treat ⟨*keyword, phylum*⟩ pairs in much the same way as ordinary identifiers in that we need only guarantee that each such pair is uniquely defined at all points in the program following any scoping discipline we wish to implement. While we have not chosen to exploit this fact, the XL syntactic framework can easily support syntax declarations with a local restricted scope, analogous to `let` declarations.

It would be interesting to investigate richer syntactic frameworks, e.g. various subclasses of the context-free grammars. The difficulty with these is that alterations to a grammar intended to have local significance may have adverse and unexpected global effects, such as rendering the grammar ambiguous. Additionally, efficient parsing algorithms such as the LR(k) methods require grammars to be in a particular form, usually defined with respect to a parse table construction algorithm. The acceptable grammars, and more significantly, the kinds of acceptable local modifications, are difficult to characterize in an intuitively appealing way. It appears necessary either to accept a broader class of grammars than is customarily used in programming language definition, or to adopt a significantly more restrictive syntactic framework than context-free grammars.

## 9.3 Quasiquotation

In XL, the internal structure of a quasiquotation is parsed at the same time as the quasiquotation itself, and is effectively expanded at read-time into an expression that will evaluate to the semantic value of the quasiquoted form.[10] Thus any syntactic errors will be detected as early as possible, when the quasiquotation is read, not when it is evaluated. In this way, we avoid the possibility of syntax errors detected during macro expansion.

One consequence of this choice is that no semantic information, static or otherwise, is present to guide the parse, thus the syntactic class expected of an unquoted form must be determined from surrounding context. It is a property of XL that the parser can always predict the expected syntactic class of a constituent without examining any of its symbols, modulo detection of an omitted `&null` keyword and the syntactic coercions introduced by `atom-syntax`. In the latter case, an explicit conversion using one of the "`(`*literal*`->*expr)`" functions is required. It would be possible (and desirable) to postpone the parsing of the quasiquotation until it is compiled (i.e. after it is read), but before execution. In this case, the types of the quasiquotation arguments might be used to resolve the expected syntactic class of the unquoted expressions, avoiding the need for the conversion operators while preserving the property that expansion-time syntax errors are not possible. Note that it would be necessary in this case to insure that each type is associated with

[10]This is analogous to the treatment of quasiquotation ("backquote") in Common Lisp, in which quasiquotations are expanded at read-time. Compare with Scheme, in which the backquote read-macro simply expands into a `quasiquote` form, which is then expanded during the normal macro-expansion phase.

at most one of the syntactic classes from among which we must distinguish. The present design would have to be altered, as the type **token** is associated with two syntactic classes coercible to EXPR, namely the atomic phyla NAME and TOKEN. That this more desirable approach was not taken is largely a historical accident, frozen early in the design before its consequences were appreciated.

## 9.4 Suppression of syntax in the metalanguage

In the design of XL, we chose to keep syntactic issues out of the metalanguage to the greatest extent possible. Thus keyword tables are hidden in the operation of XL parser and the primitive syntax extension commands. It is this choice that forces those commands to be primitive and non-extensible. To permit extension of the set of defining forms would require that we provide hooks giving user-defined metaprograms some form of access to the parser and its tables.

The decision to deal with program fragments entirely in terms of their semantic values is crucial to the "flavor" of XL metaprogramming. From the user's point of view, there simply are no uninterpreted syntactic forms. Every form manipulated in a metaprogram is inseparably connected with its syntactic class and semantic value and, furthermore, the structure of the form is not accessible except to the extent revealed by the semantic value. Compare the XL examples with Lisp macros in which the expander function traverses the expression to be expanded with structure accessors such as **car**, **cdr**, etc.

There is, unfortunately, a severe negative consequence of our peculiar treatment of macro "expansion." Since the resulting semantic value gives us no hint of how it came into being, an erroneous macro definition can be exceedingly difficult to debug. Indeed, since we are not constrained to construct the expansion by instantiating quasiquotation templates, we are not even guaranteed that a semantically-equivalent base-language program text (i.e. the usual sense of macro expansion) exists at all! In principle, the situation is no worse than in ordinary object-level programming, particularly when programming extensively with higher-order functions. However, the debugging leverage provided by syntactic macro expansion regimes illustrates a significant area in which semantic macros fall uncomfortably short of current practice.

## 9.5 Scope issues and modularity

It is clear that some sort of module mechanism is needed in order to organize collections of language extensions and to encapsulate the abstractions they define. This issue has not been addressed in this work, as we have chosen to focus our attention elsewhere. It appears likely that, given the current syntactic framework, a conventional module scheme could be adapted for this purpose by treating phylum names and ⟨*keyword, phylum*⟩ pairs in the same way as ordinary identifiers.

### 9.6 Type system

The type system of XL is unsophisticated compared to that of, say, ML [16]. This choice was made almost entirely for the sake of expedience, as type systems *per se*, as opposed to statically-typed languages, were not the intended object of this research. In retrospect, an ML-style polymorphic type inferencer seems nearly as easy to construct, and would have made the language much easier to use. Furthermore, many XL operations are special forms only because they need to be polymorphic. If ordinary expressions could be given polymorphic types, these operations could have been imported directly from Scheme. Nonetheless, using the simpler type system did permit the definition of an interesting record type. Incorporation of types with named components into an ML-like type system is a tricky business, and no entirely satisfactory solution has been found.[11]

## 10 Implementation status

A prototype implementation of XL has been written in Scheme. We used the $R^3RS$ Scheme environment provided as a subset of **T** [22], and imported a few facilities from **T** missing from $R^3RS$ Scheme, such as macros. Each XL command is processed completely before the next command is read. Forms are read as S-expressions by the Scheme reader, and then analyzed to produce their semantic values. While in principle a parser could produce the semantic values directly without the intermediate S-expression representation, it was convenient to use the existing Scheme reader. Evaluations and definitions are translated first into Scheme code, and then passed on to the underlying Scheme system for further processing. Since the dynamic semantics of XL are nearly identical to Scheme, this translation is trivial, thus the compiler is concerned nearly exclusively with the aspects of compilation peculiar to XL. The code consists of approximately 3000 lines divided among 14 source files. The compiler is written in a style reminiscent of XL extensions. A version of `phylum`, `syntax`, and `atom-syntax` are provided as macro extensions to Scheme, and are used to define the built-in XL forms. The interfacing of user-defined semantic values to those manipulated by the compiler is greatly simplified,

[11]Essentially, one must either admit the occasional necessity of using an explicit type declaration to resolve a typing ambiguity (as in Standard ML [16]), or introduce the concept of subtyping and type inclusion, which significantly increases the complexity of the tvpe inference algorithm. See [9] and [29] for examples of such extensions.

indeed made trivial, by the use of the same representations internally as are seen by the user.

## 11 Comparison with related work

This work was motivated primarily by a desire to adapt the macro mechanism of Lisp to statically-typed languages. The influence of Lisp should thus be clear, as has been elucidated previously. Here we summarize other notable research relevant to our program, and examine the similarities and contrasts between such work and our own.

### 11.1 Extensible languages

In the decade from 1965 to 1975, so-called "extensible languages" were a popular topic of investigation. Standish [26] counted 27 proposals for such languages. Solntseff and Yezerski [25] provide an excellent survey. Most extensible language proposals rely on some form of macro-processing, usually at the lexical or syntactic level. Of particular interest are *syntax macros* [3] [11], in which a syntactic type is specified for each macro argument and for the expanded result. In such a scheme, macro calls are syntactically indistinguishable from built-in forms.

Nearly all of the proposals provide for limited expansion-time computation, such as arithmetic and conditional replacement. In most cases, however, the compile-time metalanguage is relatively weak compared to the base language. In his seminal paper on definitional facilities [4], Cheatham proposes the use of an embedded syntax macro processor for extension purposes, but restricts the "semantics" of definitions to an expansion string and a few built-in macro-time operations for manipulating tables (e.g. the symbol table). He dismisses more general semantic processing as infeasible.

A radically different position is taken by Irons [8]. In describing the IMP language, he claims that "nothing short of a general programming language" would be adequate to express the required translation-time processes. His scheme, while based on the translator generator model of Cheatham's proposal, provides for the use of the entire IMP language, an Algol-like language with list-processing features, for expressing the semantics of definitions. He states that "the semantic part of an extension is in fact not a macro shell but a computation which is evaluated as a part of the translation process." Here we find a clear statement of the view of macro-processing taken in Lisp and XL. Unfortunately, IMP's realization of this insight is defective. Despite the provision of list-processing facilities, including pattern matching and substitution primitives, IMP is an imperative language in the

Algol style. Semantic attributes (e.g. types) are communicated between definition instances in global variables, with semantic processing occurring at fixed times during multiple parse tree traversals in a fixed order. Multiple semantic actions may be specified, each labelled with a number indicating the time (traversal number) at which it is to be executed. In the resulting style, semantic actions cooperate harmoniously only by careful adherence to systemwide programming conventions.[12] The interface between cooperating definitions is completely unconstrained, and there is no guarantee of commonality in the behavior of semantic actions of multiple definitions belonging to the same syntactic class (which should be, on syntactic grounds, interchangeable).

Our approach differs from that of Irons mainly in that we use a typed functional metalanguage, and replace imperative semantic actions with functional semantic values. By extending the type discipline to these values in such a way that each syntactic class is associated with a single semantic value type, we guarantee that every definition belonging to a given syntactic class will conform to a common interface to the extent that can be captured in the type. While the type discipline cannot assure us that semantic values are defined with the meanings intended, it does provide the means by which abstraction barriers can be enforced by the compiler. In XL, we need not worry that a correct definition will be rendered erroneous by a change to an apparently unrelated one. A further difference can be found in our treatment of data types. We provide a rich set of compile-time operations on type descriptors, by which complex structured types can be expressed. Iron's type system is rather simple, modelled on that of Algol-60. Although it would be possible to formulate complex type descriptions in terms of IMP list structures, it is not clear that such descriptions could be made accessible to the existing base language type system. In contrast, our system provides a "hook" into the underlying type system, so that type extensions function as built-in types with respect to the built-in operations.

## 11.2 Semantics-directed compiler generation

There have been a number of attempts to generate compilers from semantic specifications in the denotational style. Early systems merely expanded the program into a lambda-calculus expression representing its denotation, which was then either directly interpreted, as in Mosses' SIS [18], or compiled into SECD machine code, as in Paulson's compiler generator [19]. No distinction was made between

[12]Witness the example in [8], page 35, 2nd column. Here a convention is described whereby an expression can determine whether it is being used to the left or the right of an assignment, and can adjust its behavior accordingly.

static and dynamic semantics in the language specification, though a compile-time beta-reducer could signal static errors before execution time.

Lee and Pleban have taken a somewhat different approach in MESS [12] [13] [20]. They factor the semantic specification into two parts, a *macrosemantics*, which makes no reference to the run-time model (e.g. details such as whether stores or continuations are used), and a *microsemantics*, which supplies this model. The static semantics are captured entirely in the macrosemantics, whereas the microsemantics specifies a language, comparable to the IR in a conventional compiler, into which programs are translated. By replacing the microsemantic specification with an actual code generator, a realistic compiler may be generated.[13] In practice, the macrosemantics metalanguage serves as a high-level notation for writing directly executable specifications of compiler front-ends.

Unlike our system, MESS is an off-line compiler generation system, not an extensible compiler. Furthermore, our target language is the source language itself, not an IR as represented by the microsemantics in MESS. Nonetheless, our approach was influenced by the observation that in MESS the techniques of denotational semantics, applied to static semantic analysis only, could directly yield an acceptable implementation. In XL, we make no pretense that language extensions are declarative specifications, but the purely functional style of denotational definitions and the notion of higher-order semantic domains contribute much toward the character of XL metaprogramming.

## 11.3 Syntactic closures

In [2], Bawden and Rees present a proposed solution to the scoping problems associated with Lisp macros, including the inadvertant capture of free identifiers in macro arguments by variables bound locally in the expansion, and assuring the closure of the macro expansion in the context of the macro definition. Our treatment of partially-closed expressions, e.g. those produced by `close-expr`, is borrowed directly from this paper, adapted only to account for the fact that we deal with semantic values instead of uninterpreted S-expressions. One advantage of our formulation is that a distinguished "syntactic closure" type is not needed, as we can represent all expressions uniformly as ordinary functions. Bawden and Rees leave open the question of an appropriate "pattern matching" language for specifying the syntax of macro definitions. Our syntax declarations might be adapted to this purpose, with the removal of static typing to accommodate the dynamic type discipline of Lisp.

[13]The purist may object, we believe with justification, that the claim to being "semantics directed" is lost when an ad-hoc code generator is added in this way, though the work of Appel [1] might be applied to make the derivation of the code generator somewhat better grounded in the theory.

## 11.4 Reflective programming

Brian Smith introduced the idea of reflective programming in [24]. The idea has been further explored by Friedman and Wand in [6] and [30]. In a reflective programming language, code appearing textually at the object level can be "lifted" into the context of the interpreter, effectively adding new code. In reality, that interpreter need not be the "real" one, but the illusion is maintained that the user program is interpreted by a metacircular interpreter (i.e. written in the same language it interprets) that can be so extended. A crucial idea in Smith's formulation (but see [6]) is the "reflective tower", a conceptually infinite tower of metacircular interpreters, each interpreting the interpreter below, or, in the case of the lowermost one, the user's program. Given the reflective tower, code that has already been reflected up one level can reflect again to the level above, and so on as needed. At each level, the reflected code has access to an effective model of the computation taking place one level below. Furthermore, the model is "causally connected" with that computation in that changes made to the model are immediately reflected in the actual state of the computation. In Smith's language, 3-Lisp, the model provides access to both the (dynamic) environment and the continuation. Constructs that must be special forms in Lisp such as `lambda`, `if`, and `catch`, are definable as reflective procedures in 3-Lisp.

Friedman and Wand observed in [6], an exploration of a simpler, "towerless" form of reflection, that macro facilities constitute a form of compile-time reflection, but failed to elaborate further on this point. Historically, their insight motivated the early work on XL; however, we abandoned the analogy with Smith's work early in the design when it became unclear how to develop the analogy into a satisfactory definitional mechanism. We thus took a more conventional source-to-source transformation approach, in which a macro expansion *replaces* the macro call, as source code, rather than *implementing* the macro call, as object code in some underlying implementation language. We believe that the latter approach is what would be demanded by a faithful adaptation of run-time reflection to an analogous compile-time variant.

We conjecture that a notion analogous to the reflective tower of interpreters could be constructed using compilers instead. Such an approach would allow us to treat our semantic macros as true compiling transformations, translating into an object language distinct from the source language, though isomorphic to it. Semantic values to be incorporated in a macro expansion (i.e. the semantic value of the macro call) would belong to the object language. To create the semantic value of, say, an expression, we would have to provide both a (dynamic-) semantically equivalent expression in the object language and a type attribute representing the interpretation we wish to give it at the source level.

Since every source expression would be fully translated into an expression in a distinct language, the source expression would never inherit any static semantic attributes from its expansion. Furthermore, the namespaces of the source program and the expansion would be completely disjoint, avoiding the need to carefully manage the environment with respect to which the expansion is closed, e.g. `definition-environment` and the partial closure operators would no longer be needed. Note that this situation is the usual case in a compiler, and nothing odd would be going on at all if not for the use of an object language isomorphic to the source language. In this case, there is no semantic "lowering" during compilation, and we must imagine the same translation process applied to the object program as well, and so on ad infinitum. Thus we imagine an infinite tower of compilers in which each compiler translates the output of the compiler below it into a program in essentially the same language. The reflection primitives would "add lines" to the compiler that processes them analogous to reflection in the 3-Lisp interpreter. At some level, as we ascend the tower, we will presumably arrive at some level beyond which the program we are compiling does not reflect, as to require unbounded reflection is an error analogous to unbounded recursion. Above this level, all the compilers are compiling the same language (more correctly, isomorphic languages) so we can recognize this condition dynamically and close off the potential infinite regress, appealing to the primitive implementation of the base language. This is precisely the manner in which Smith's interpreter works.

Suffice it to say that we are not claiming (necessarily) that such macros would be any easier to write or prettier to look at, but that the translation model for extensions of existing phyla would more closely resemble that of embedded languages. We find the need for the partial closure operations aesthetically unsatisfying, feeling that it is cleaner to rigorously maintain an "implements" relation between levels in the sense that quasiquotations should represent phrases in the implementation language, not the source language. To maintain this view uniformly, while retaining a language isomorphic to the source language as the target, requires us to imagine that the built-in forms are also being compiled in this way, though the implementation mapping is trivial.

## 12 Future research

As it stands, the XL definitional facility makes it possible to define significant extensions of the same degree of generality permissible by Lisp, but with full static semantic analysis in terms of the same model used by the built-in constructs. Furthermore, the mechanisms are provided whereby embedded languages (i.e. systems of extensions loosely coupled to the base language) can implement completely arbitrary compile-time processing.

Unfortunately, the price to be paid is difficulty in use. While it is true that the major examples in this paper are of a kind likely to be written by users of considerable sophistication (e.g an expert systems programmer or a programming group's "toolsmith"), the mechanisms of XL make even simple definitions more difficult than they should be. The main culprit is the treatment of environments. Facilities are provided for assuring that every expression is closed in an appropriate environment, but they constitute a low-level "toolkit" providing little guidance for their use. Indeed, the Lisp experience shows that programmers are often sloppy with macro closure issues, relying on the choice of obscure names in preference to the reliable but more verbose technique of using system-generated globally unique names ("gensyms"). The most severe obstacle to the practical use of an XL-style definitional facility appears to be this clumsy treatment of closure. A completely adequate solution would make macro closure completely automatic, with no more thought required of the programmer than that required by the ordinary lexical scope rules applying to functions. We do not believe that syntactic closures, either as proposed in [2] for Scheme or as adapted by us for XL, represent an adequate solution for either language.

XL has chosen to treat environments as part of static semantics, following the traditional treatment in compilers (e.g. the symbol table), and in doing so has had to treat the "meaning" of most syntactic forms, as represented by the semantic value, as a function of the surrounding environment. However, important semantic properties, such as type, are not well-defined until this environmental dependence has been removed. While a departure from most current practice, it might be fruitful to explore an approach in which naming is subsumed into the syntactic component of the language, and in which names represent purely structural references to the defining forms as syntactic instances of declarations. Some support for this approach is given by the observation that all scoping disciplines definable in terms of the XL primitives reduce to those expressable by a purely syntactic macro expansion modulo variable renaming and restrictions on variable visibility.

While we have claimed that the definitional facilities of XL can be used to encapsulate data and control abstractions, we have not provided the visibility control mechanisms that would be needed to enforce the integrity of these abstractions. Ideally, we would like to be able to build libraries of useful definitions and import them freely as needed without concern for their internals. The presence of syntactic extensions, beyond the definition of new names, presents problems that deserve further investigation.

A richer type system, supporting polymorphism, would contribute much to the usability of the language. An ML-style type inference system would go a long way toward eliminating the notational clutter due to explicit type declarations for function arguments and results. There does not appear to be any problem in principle in making this change, but it would require a complete reformulation of the compile-time type-checking primitives.

A less restrictive syntactic framework would ideally be preferred to the one currently adopted, yet abandoning the uniform, fully-parenthesized notation would complicate any scheme for program modularity, as well as introducing a new level of complexity into the parser. We believe that the choice to stay within the Lisp experience on this point has served us well in this project.

## 13 Conclusions

We believe that we have demonstrated that a macro-like definitional mechanism can be incorporated into a statically-typed language in a manner which respects the integrity of the base language while permitting definitions a substantial degree of access to static semantic attributes not normally accessible. One of the simplest but most important consequences of this power is that our macros can report semantic errors in terms of the macro call, not its expansion. Our definitions, while consisting of program extensions to the compiler itself, cannot compromise its type-correctness. Furthermore, extensions are compartmentalized in such a way that the translation of instances of correctly-written extensions (as well as of the built-in constructs) cannot be affected in a catastrophic manner by erroneous definitions elsewhere in the program. The possible interactions between distinct definitions are tightly constrained and intellectually manageable. Our examples illustrate the value of macro definitions as an abstraction mechanism, and how semantic attributes can be exploited to enforce useful constraints, such as restricting the macro to arguments of certain types. On the negative side, XL programs are much too verbose, and the "cure" proposed for the traditional macro scoping problems seems almost worse than the disease. We have provided a few directions for future research which promise to ameliorate this deficiency.

## References

[1] Andrew W. Appel. Semantics-directed code generation. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 315–324, 1985.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 86–95, 1988.

[3] W. R. Campbell. A compiler definition facility based on the syntactic macro. *Computer Journal*, 21(1):35–41, 1975.

[4] T. E. Cheatham, Jr. The introduction of definitional facilities into higher level programming languages. In *Proceedings of the 1966 Fall Joint Computer Conference*, pages 623–637, 1966.

[5] David R. Cheriton and Michael E. Wolf. Extensions for multi-module records in conventional programming languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 296–306, 1987.

[6] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 348–355, 1984.

[7] David K. Gifford et al. FX-87 reference manual. Technical Report MIT LCS TR-409, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1987.

[8] Edgar T. Irons. Experience with an extensible language. *CACM*, 13(1):31–40, January 1970.

[9] Lalita Jategaonkar and John Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 198–211, 1988.

[10] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Bell Laboratories, Murray Hill, New Jersey. Reprinted in *Unix Programmer's Manual: Supplementary Documents*, distributed with 4.2BSD, 1984.

[11] B. M. Leavenworth. Syntax macros and extended translation. *CACM*, 9:790–793, 1966.

[12] Peter Lee and Uwe Pleban. On the use of LISP in implementing denotational semantics: A progress report. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 233–248, 1986.

[13] Peter Lee and Uwe Pleban. A realistic compiler generator based on high-level semantics: Another progress report. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 284–295, 1987.

[14] M. E. Lesk and E. Schmidt. Lex – a lexical analyzer generator. Bell Laboratories, Murray Hill, New Jersey. Reprinted in *Unix Programmer's Manual: Supplementary Documents*, distributed with 4.2BSD, 1984.

[15] Barbara Liskov et al. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[16] Robin Milner. The Standard ML core language. *Polymorphism*, 2(2), October 1985. An earlier version of this paper appeared in *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming.*

[17] David Moon, Richard Stallman, and Daniel Weinreb. *LISP Machine Manual.* MIT Artificial Intelligence Lab, fifth edition, January 1983.

[18] P. Mosses. SIS – Semantics Implementation System. Technical Report DAIMI MD-30, Computer Science Department, Aarhus University, August 1979.

[19] Lawrence Paulson. A semantics-directed compiler generator. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 224–233, 1982.

[20] Uwe F. Pleban and Peter Lee. An automatically generated, realistic compiler for an imperative programming language. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 222–232, 1988.

[21] Jonathan Rees and William Clinger (editors). Revised$^3$ report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[22] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual.* Computer Science Department, Yale University, fourth edition, January 1984.

[23] Mary Shaw, editor. *Alphard: Form and Content.* Springer-Verlag, New York, 1981.

[24] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[25] N. Solntseff and A. Yezerski. *A Survey of Extensible Programming Languages*, volume 7 of *Annual Review in Automatic Programming*, pages 267–307. Pergammon Press, 1974.

[26] Thomas A. Standish. Extensibility in programming language design. In *Proceedings of the National Computer Conference*, pages 287–290, Montvale, New Jersey, 1975. AFIPS Press.

[27] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 1984.

[28] Mitchell Wand. Complete type inference for simple objects. In *1987 IEEE Symposium on Logic in Computer Science*, pages 37–44. IEEE Computer Society Press, 1987.

[29] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 298–307, 1986.