# STELLA - A Lisp-like Language for Symbolic Programming with Delivery in Common Lisp, C++, and Java

**Hans Chalupsky** **Robert M. MacGregor**

USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

**Abstract**

We describe STELLA,[1] a strongly typed, object-oriented, Lisp-like language, designed to facilitate symbolic programming tasks in artificial intelligence applications. STELLA preserves those features of Common Lisp deemed essential for symbolic programming such as built-in support for dynamic data structures, heterogeneous collections, first-class symbols, powerful iteration constructs, name spaces, an object-oriented type system with a meta-object protocol, exception handling, and language extensibility through macros, but without compromising execution speed, interoperability with non-STELLA programs, and platform independence. STELLA programs are translated into a target language such as C++, Common Lisp, or Java, and then compiled with the native target language compiler to generate executable code. The language constructs of STELLA are restricted to those that can be translated directly into native constructs of the intended target languages, thus enabling the generation of highly efficient as well as readable code.

## Introduction

From its inception about 40 years ago, Lisp was intended specifically to support the writing of artificial intelligence (AI) software (McCarthy 1981), and it has been one of the most popular AI programming languages ever since. However, despite its high level of maturity, for example, the existence of standards for major dialects such as Common Lisp and Scheme and the emergence of an international standard for ISLISP, it has not become a mainstream programming language such as C, C++ or, lately but quickly, Java. This is one of the reasons why it is generally difficult to deliver Lisp libraries or applications that smoothly interoperate with standard non-Lisp software, such as, for example, GUI tools, commercial off-the-shelf software, tool libraries, etc. Other reasons are the technical nature and size of Lisp implementations (in particular for feature-rich dialects such as Common Lisp), as well as the lack of Lisp knowledge in the general non-AI programming community. Balzer (1990) conjectures that AI's lack of impact on real world software engineering is due to its isolationist technology and approaches manifested by its use of idiosyncratic languages (Lisp), idiosyncratic environments, and idiosyncratic hardware (now extinct Lisp machines).

From the standpoint of AI this is very unfortunate, since, on the one hand, Lisp is extremely well suited for symbolic programming tasks commonly found in AI. For example, Shrobe (1996) argues that the AI community still needs Lisp because of its dynamic nature and rich development environments that facilitate rapid prototyping, as well as for its unique support for creating and embedding new domain or problem-specific languages. On the other hand, the success of AI as perceived outside of the field is more and more linked to the successful fielding of AI technology in non-AI settings, and, since this success depends heavily on the technical and commercial viability of the programming languages used, we claim that Lisp is an increasingly less suitable choice.

When the second author embarked on the project of developing a large knowledge representation system that had to be delivered in C++, he was faced exactly with the dilemma described above: the symbolic programming nature of the project strongly favored Common Lisp while "real world" constraints demanded C++. The solution was the development of a new language called STELLA, which preserves those features of Common Lisp deemed essential for symbolic programming, but without compromising execution speed, interoperability with non-STELLA programs, and platform independence.

The motivation for the development of the language Dylan (Shalit 1996) was very similar, namely, to preserve the best features of Common Lisp without compromising the ability to generate tight and efficient application programs. Scott Fahlman who has been involved in the development of Common Lisp as well as Dylan writes that "if Dylan becomes popular for mainstream applications, it will free AI and expert system programmers from having to choose between life in the Lisp ghetto or the C++ minefield" (Shrobe *et* al. 1996, p.12). But the question is: will it become popular?

We do not venture a bet on the answer to this question. Instead, STELLA takes a safer approach by using a translation scheme that can deliver STELLA programs in three already established and widely accepted languages. Even if STELLA does not become a mainstream language – which is quite likely – it can still deliver on its promise of making AI applications available to the programming mainstream in a way that was not previously possible.

[1] "STELLA" is an acronym for **S**trongly **T**yp**E**d **L**isp-like **LA**nguage.

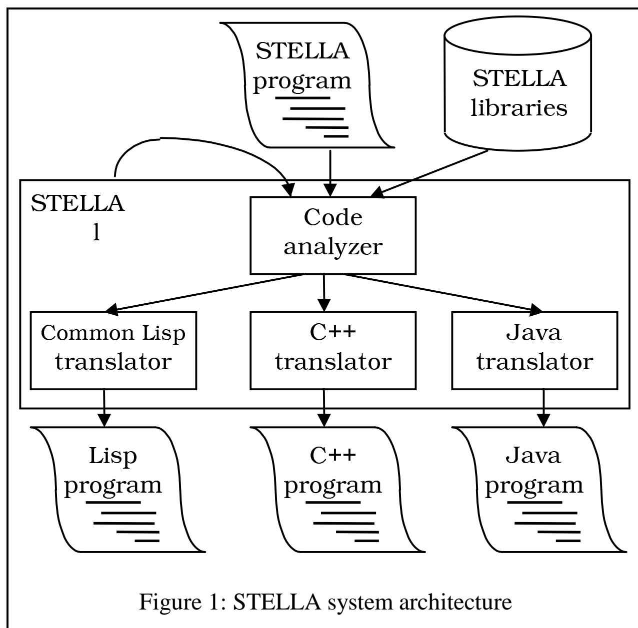


Figure 1: STELLA system architecture

## An Overview of STELLA

STELLA is a strongly typed, object-oriented, Lisp-like language. STELLA programs are first translated into either Common Lisp, C++, or Java, and then compiled with any conventional compiler for the chosen target language to generate executable code. Figure 1 gives an overview of the STELLA system architecture. Over 95% of the STELLA system is written in STELLA itself, which is the reason for the circular arc emanating from the translator.

The design of STELLA borrows from a variety of programming languages, most prominently from Common Lisp (Steele 1990), and to a lesser degree from other object-oriented languages such as Eiffel (Meyer 1992), Sather (Stoutamire & Omohundro 1996), and Dylan (Shalit 1996). Since STELLA has to be translatable into C++ (Stroustrup 1991) and Java (Gosling, Joy, & Steele 1996), various restrictions of these languages also influenced its design.

In the following, we assume that the reader is familiar with basic Common Lisp concepts, and has at least some familiarity with C++ or Java. Let us start with a cursory overview of STELLA's main features:

**Syntax:** STELLA uses a parenthesized, uniform expression syntax similar to Lisp. Most definitional constructs and control structures are similar to their Common Lisp analogues with variations to support types.

**Type system:** STELLA is strongly typed and supports efficient static compilation similar to C++. Types are required for the arguments and return values of functions and methods, for global variables, and for slot definitions. Local, lexically scoped variables can be typed implicitly by relying on type inference.

**Object system:** Types are organized into a single inheritance class hierarchy. Restricted multiple inheritance is allowed via mixin classes. Dynamic method dispatch is based on the runtime type of the first argument (similar to C++ and Java). Slots can be static (native) or dynamic. Dynamic slots can be defined at runtime and do not occupy any space until they are filled. Slots can have both initial and default values, and demons can be triggered by slot accesses. A meta-object protocol allows the control of object creation, initialization, termination, and destruction.

**Control structure**: Functions and methods are distinguished. They can have multiple (zero or more) return values and a variable number of arguments. Lisp-style macros are supported to facilitate syntax extensions. Expressions and statements are distinguished. Local variables are lexically scoped, but dynamically scoped variables (specials) are also supported. STELLA has an elegant, uniform, and efficient iteration mechanism plus a built-in protocol for iterators. An exception mechanism can be used for error handling and non-local exits.

**Symbolic programming:** Symbols are first-class objects, and extensive support for dynamic datatypes such as cons-trees, lists, sets, association lists, hash tables, extensible vectors, etc., is available. A backquote mechanism facilitates macro writing and code generation. Interpreted function call, method call, slot access, and object creation is supported, and a restricted evaluator is also available.

**Name spaces:** Functions, methods, variables, and classes occupy separate name spaces (i.e., the same name can be used for a function and a class). A hierarchical module system compartmentalizes symbol tables and supports large-scale programming.

**Memory management:** STELLA relies on automatic memory management via a garbage collector. For Lisp and Java the native garbage collector is used. For the C++ version of STELLA we use the Boehm-Weiser conservative garbage collector (Boehm 1993) with good results. Various built-in support for explicit memory management is also available.

The Common Lisp features most prominently absent from STELLA are anonymous functions via `lambda` abstraction, lexical closures, multi-methods, full-fledged `eval` (a restricted evaluator is available), optional and keyword arguments, and a modifiable readtable. STELLA does also not allow dynamic re/definition of functions and classes, even though the Lisp-based development environment provides this facility (similar to Dylan). The main influences of C++ and Java onto STELLA are the strong typing, limited multiple inheritance, first-argument polymorphism, and the distinction between statements and expressions.

## Translation Instead of Compilation

Maybe the most important characteristic of STELLA that distinguishes it from other approaches to the problem of Lisp-based application generation and interoperability is its translation philosophy. One of the main design goals has been to allow a direct translation into *readable, conventional,* and *efficient* code of the various target

languages that can be compiled with conventional compilers, since that achieves the highest degree of platform independence and interoperability with non-STELLA programs without sacrificing efficiency. Readability is of primary concern, since it enables a STELLA-illiterate application programmer to effectively and efficiently integrate some mainstream application with a piece of AI technology written in STELLA. Conventional Lisp-to-C translators are an inferior alternative, since they use C more like an assembly language which makes it difficult to access and understand underlying data and control structures. Integration approaches based on foreign function interfaces or protocols such as CORBA sacrifice efficiency, since they require data conversions or various layers of protocol.

When a STELLA program is translated into Common Lisp, C++, or Java, classes are mapped onto classes, slots onto slots, methods onto methods, functions onto functions (or static methods), etc. Native datatypes are used whenever possible, for example, STELLA strings become Lisp or C++ strings, integers are mapped onto integers, etc. This is illustrated by Figure 2 which shows a very simple STELLA function and its translations into Common Lisp and C++. Both translations are very direct and straightforward. In the Common Lisp translation all functions used from the Lisp package are qualified with the `CL` package prefix. Note, that for Lisp all type information is dropped (at higher optimization levels some type information is retained – sacrificing some readability - to provide optimization hints to the Lisp compiler). The C++ translation is even more similar to the original STELLA code (the C++ pretty-printing in this and all following examples has been changed slightly to save vertical space).

**STELLA:**

```
(defun (fact INTEGER) ((n INTEGER))
  (if (= n 0)
    (return 1)
    (return (* n (fact (1- n))))))
```

**Common Lisp:**

```
(CL:defun fact (n)
  (CL:if (CL:= n 0)
         1
         (CL:* n (fact (CL:1- n)))))
```

**C++:**

```
int fact(int n) {
  if (n == 0) {
    return (1);}
  else {
    return (n * fact(n - 1));}}
```

Figure 2: A simple STELLA function and its translations

Note, that both the return value and the parameter of the STELLA function are explicitly typed. Since STELLA distinguishes between statements and expressions (similar to C++), the STELLA `if` does not return a value; hence, function values are returned via explicit calls to `return`.

The direct mapping between STELLA and the target languages not only serves readability but also efficiency. For example, in the C++ translation function calls do not require an extra indirection through a function cell, method calls use the very efficient C++ v-table mechanism, and slot access is almost as efficient as the access to local variables. These are all areas where the expressiveness and dynamic nature of Common Lisp has to be paid for with a loss of efficiency.

## The STELLA Type System

The most fundamental difference between STELLA and Common Lisp is that STELLA is strongly typed. While Common Lisp does have a full-fledged and complex type system, type declarations for functions and variables are completely optional. Moreover, a Common Lisp compiler is free to ignore any user provided type declarations, which discourages programmers to provide them in the first place (if they are not discouraged already by the somewhat arcane declaration syntax). Most Lisp programmers regard the untyped nature of Common Lisp as a feature that facilitates rapid prototyping and only provide type information if they hope for a particular optimization by the compiler.

Since STELLA is aimed to be directly translatable into strongly-typed languages such as C++ or Java, it has to have a type system that can be mapped more or less directly onto the type systems of these target languages. Strong typing might seem unpalatable to many Lisp programmers accustomed to a programming style that does not rely on explicit typing; however, we will show that STELLA's sophisticated type system combined with its type inference mechanism allows one to get away with a minimum of explicit type declarations while still reaping the safety and performance benefits of a strongly typed language.

Types in STELLA can be categorized into *literal* types such as `INTEGER` or `STRING`, and *object* (or non-literal) types such as `OBJECT`, `LIST`, or `SYMBOL`. This corresponds roughly to the distinction between primitive and reference types in Java. In the simplest case, a type corresponds to a class. For example, the following class definition creates the type `PERSON`:

```
(defclass PERSON (OBJECT)
  :documentation "The class of people."
  :slots ((name :type STRING)
          (age :type INTEGER)
          (father :type PERSON)
          (mother :type PERSON)))
```

By convention, we upcase type names, even though this usually does not matter, since by default STELLA is case-insensitive. The class definition shown above uses a nested keyword/value syntax similar to the `defclass` macro of the Common Lisp Object System (CLOS), but with a different set of option keywords. It defines the new class (or type) `PERSON` as a subclass of the class `OBJECT`. In fact, `OBJECT` is the top of the STELLA

object type hierarchy, thus, every object type used by STELLA is a subtype of it. Every instance of PERSON has exactly the storage slots shown above with their respective types (there are no slots inherited from OBJECT). Here is the C++ version of this class to again illustrate the readability of the translation:

```
class Person : public Object {
// The class of people.
public:
   char* name;
   int age;
   Person* father;
   Person* mother;
public:
   virtual Surrogate* primary_type(); };
```

Every class definition usually also generates various auxiliary functions and methods such as the primary_type method shown above which facilitates runtime type determination.

Apart from simple types such as PERSON, STELLA also has *parameterized* and *anchored* types. Parameterized types instantiate parameterized classes. This mechanism provides a kind of polymorphism that is commonly used to implement generic containers, such as, for example, a list datatype that can take arbitrary elements of some parameter type *T*. Instead of having to provide multiple definitions for the container class and its associated functions and methods – one for each individual element type that is needed by a particular application – it suffices to provide a single parameterized definition. For example, the STELLA kernel class LIST which implements a dynamic, singly-linked list datatype is defined as follows:

```
(defclass LIST (SEQUENCE)
  :documentation "The class of singly-linked lists."
  :parameters ((any-value :type OBJECT))
  :slots ((the-cons-list
            :type (CONS OF (LIKE (any-value self)))
            :initially NIL)))
```

The :parameters keyword is used to parameterize a class definition. In the given example, the class LIST has one parameter with name any-value. Its type, OBJECT, serves as a constraint on the parameter types that can legally instantiate the class. Parameterized types have the syntax ($T$ OF $P_1 \ldots P_n$) where $T$ is called the base type and the $P_i$ are called the parameter types. Each parameter type has to be a subtype of the type that constrains the corresponding parameter. For example, the type (LIST OF PERSON) is a legal instantiation of LIST. The base type $T$ can also be used by itself, in which case the $P_i$ are assumed to be of the type of the corresponding class parameter; thus, the type LIST is equivalent to the parameterized type (LIST OF OBJECT). A parameter type can itself be parameterized, therefore allowing arbitrarily nested type expressions.

STELLA lists simply add a header element to a more low-level Lisp-style CONS-list which is stored in the slot the-cons-list (the somewhat complicated type specification of that slot will be explained shortly). Armed with all that, we can now extend the PERSON class defined above to also account for a person's siblings:

```
(defclass PERSON (OBJECT)
  :documentation "The class of people."
  :slots ((name :type STRING)
          (age :type INTEGER)
          (father :type PERSON)
          (mother :type PERSON)
          (siblings :type (LIST OF PERSON))))
```

Providing such detailed type information not only documents the intent of the programmer, it also enables the STELLA translator to verify the integrity of accesses to a particular person's siblings slot. For example, whenever a new sibling is added via a call to the STELLA method insert, the translator can verify at compile time whether the inserted element is indeed of type PERSON. Conversely, whenever a sibling element is retrieved from the list, the STELLA translator knows that it has to be of type PERSON, and it can use that knowledge for type inference and to generate the necessary type conversions in the target language.

The concept of anchored types was borrowed from Eiffel (Meyer 1992). Anchored types can be used to provide a type by pointing to the type of some other typed entity (called the anchor) instead of explicitly duplicating the anchor's type information (this mechanism is somewhat similar to symbolic links in a file system). Anchored types serve two main functions: (1) to avoid redundant duplication of type information (which facilitates code maintenance), and (2) to declare dependencies between types (which assists type inference).

Anchored types have the syntax (LIKE *anchor*) where the syntax of *anchor* depends on the context of the declaration. For example, in the definition of the class LIST given above, the type of the slot the-cons-list uses an anchored type as a parameter type. The anchor (any-value self) refers to the type of the class parameter any-value (self is a special keyword that refers to the class in whose context the definition occurs). Since the anchor points to a class parameter, the parameter type of the-cons-list gets effectively linked to the parameter type of the LIST class. For example, when the STELLA translator analyzes the expression (the-cons-list (siblings p)) which accesses the slot the-cons-list of the siblings list of some person p, it can infer that its type must be (CONS OF PERSON).

## Explicit, Implicit, and Inferred Types

In STELLA all globally visible type information of a translation unit such as a class, function, method, or global variable has to be provided explicitly. This contractual information visible to the outside is often called an entity's *signature*. For example, all of a function's argument and

return values have to be typed explicitly, all slots of a class have to be typed, etc.

The types of local variables, on the other hand, can be provided implicitly or by relying on type inference. If a local variable is not typed explicitly, it is assumed to be of the type of its initialization argument. For example: let us assume that a person's siblings are ordered from oldest to youngest, and that a sufficiently fine-grained age representation is used. Then the predicate below returns true if its argument is a family's first-born child:

```
(defun (first-born? BOOLEAN) ((p PERSON))
  :documentation "True if 'p' is a first-born child."
  (let ((s (siblings p))
        (os (value (the-cons-list s))))
    (return (or (null? os)
                (> (age p) (age os))))))
```

Both local variables `s` and `os` introduced by the `let` statement above are untyped, hence, STELLA implicitly types them from their initialization arguments (note: the STELLA `let` initializes its variables sequentially similar to the Common Lisp `let*`). The type of `s` is assumed to be `(LIST OF PERSON)`, since that is the type of the `siblings` slot of a `PERSON`. The determination of the type of `os` is slightly more complicated. A STELLA `CONS`-cell has two slots, `value` and `rest`, which correspond to the CAR and CDR of a Lisp cons-cell. If nothing else is known, the `value` slot of a `CONS` cell is of type `OBJECT`. However, in this case STELLA can actually infer a narrower, more specific type than that. Remember, that the type of the slot `the-cons-list` of the class `LIST` defined above was anchored to the type of the class parameter; hence, the type of the expression `(the-cons-list s)` is inferred to be `(CONS OF PERSON)`. The `value` slot of a CONS cell is typed as `(LIKE (any-value self))`, that is, it also is anchored to the parameter of its class; thus, the type of the expression `(value (the-cons-list s))` is inferred to be `PERSON` which is the type STELLA assumes for the variable `os`. Here is how these types manifest themselves in the C++ translation:[2]

```
boolean first_bornP (Person* p) {
  // True if 'p' is a first-born child.
  {List* s = p->siblings;
   Person* os = ((Person*)(s->the_cons_list->value));
   return ((os == NULL) || (p->age > os->age)); }}
```

In the translation of `s` the parameter type information is dropped. Instead, type information inferred by STELLA but not available to C++ or Java is communicated via explicit, static type casts as used in the initialization of `os`. This cast is necessary, since the type of the `value` slot as known by C++ or Java is `OBJECT`. In C++ these casts do not incur any runtime overhead, since we can translate STELLA classes into a single, `virtual` inheritance C++ class hierarchy. Unfortunately, Java is not quite as trusting and checks casts at runtime for their validity (somewhat similar to the runtime type checking in Lisp). We do not yet know how much overhead these runtime type checks actually incur, however, since Java lacks parametric types, every use of a generic collection data structure such as Java's built-in vectors forces the programmer to use explicit casts. For this reason, Java translations of STELLA code should not behave significantly different in this respect than manually written Java code.

For the sake of the example above, we exposed the representation of the underlying list representation. A better and more elegant implementation would use only a single local variable and STELLA's generic `first` method to access the oldest sibling. This would then allow us to change the underlying representation of the `siblings` slot, for example, to use a vector instead of a list, without having to perform any maintenance on the function `first-born?` at all. Besides relieving the programmer from the burden of explicit typing, one of the most important benefits of implicit typing and type inference is its support for automatic software maintenance. Since fewer types are actually materialized in the code, changing a type will syntactically affect fewer places. In traditional languages, the same insulation can only be achieved with help of user-defined "logical" types. The caveat is that type changes can also indicate semantic changes, in which case implicit typing can make it more difficult to find all the affected places. Implicit typing and type inference is also a crucial ingredient of STELLA's elegant iteration facility described below.

Implicit typing is not always sufficient. For that case STELLA `let` declarations do take an optional type argument that can be used to override an inferred type, or to provide a type if a variable is initialized to the typeless `NULL` value.

## Other Benefits of the STELLA Type System

Besides aiding type checking and efficient translation, the STELLA type system also provides various other benefits. For example, in a language such as Java that does not have parameterized types, the type checks and explicit casts generated by the STELLA translator would have to be performed manually by the programmer, which is tedious, error prone, and leads to inefficient and hard to maintain code (see (Myers, Bank, & Liskov 1997) for a discussion of parameterized types, and how they could be added to Java). In C++ we could have used templates, but those usually implement genericity by code duplication, e.g., by generating a different version of the `first` method for each different parameter type it is used on. STELLA can achieve the same effect more elegantly with only a single implementation of `first`.

Even Common Lisp could benefit from parameterized types, since there it is the sole responsibility of the programmer to verify that the type contract of the

[2] In the translation of the function name characters that are illegal in C++ identifiers were replaced by legal substitutes.

siblings slot does not get violated. If s/he was not careful in doing so, s/he will eventually be disciplined by the Lisp debugger. As an added benefit, STELLA's type system facilitates the generation of more efficient Lisp code. As mentioned above, at higher optimization levels the STELLA-to-Lisp translator retains some type information to enable optimizations by the Lisp compiler. Lisp array access and integer arithmetic can benefit greatly from such declarations, in particular, since the translator also types intermediate expressions which even type-aware Lisp programmers often overlook.

The type system also supports automatic type conversions that would otherwise have to be done tediously by hand. For example, literals such as numbers or strings have to be wrapped (objectified) before they can be stored in generic containers such as vectors or lists. Because of the available type information, STELLA can perform the necessary conversions fully automatically.

## Iteration

STELLA has a powerful iteration facility that provides efficient and extensible iteration over arbitrary collections in a uniform syntax that is inspired by Common Lisp's `loop` macro. For example, the following is a revised version of the `first-born?` function that does not rely on any ordering of the `siblings` slot:

```
(defun (first-born? BOOLEAN) ((p PERSON))
   (foreach s in (siblings p)
       where (> (age s) (age p))
       do (return FALSE))
   (return TRUE))
```

This version is highly preferable over the previous one, since it does not expose the representation of the `siblings` slot. The STELLA `foreach` uses a uniform syntax for iteration regardless of the type of the underlying collection. Such collections can be lists, association lists, strings, vectors, integer intervals, STELLA or user-defined iterators, etc. This means that we could change the definition of the `PERSON` class to, for example, use a vector instead of a list representation for the `siblings` slot, without having to adapt the function above. In this respect the Common Lisp `loop` macro is clearly inferior, since it exposes the datatype of the underlying collection by requiring different keywords to indicate the collection type. This is a case where the lack of type information in Common Lisp compromises elegance as well as maintainability. The uniformity of `foreach` does not affect the efficiency of the generated code. For example, here is the C++ translation of the function above:

```
boolean first_bornP (Person* p) {
  { Person* s = NULL;
    Cons* iter_001 = p->siblings->the_cons_list;
    while (!(iter_001 == NIL)) {
      { s = ((Person*)(iter_001->value));
        iter_001 = iter_001->rest; }
      if (s->age > p->age) {
        return (FALSE); }}}
  return (TRUE); }
```

The STELLA translator optimizes iteration over various commonly used collection data structures; hence, the generated iteration over the underlying `CONS`-list data structure is about as efficient as one could write it by hand. Note, how type inference is crucial here to automatically type the loop variable and the helper variable `iter_001`. STELLA also supports parallel iteration over two or more collections, collection of results into a result list, destructive modification of the underlying collection data structure, and the `foreach` variants `some`, `exists`, and `forall`. For example, here is another version of the function `first-born?`:

```
(defun (first-born? BOOLEAN) ((p PERSON))
   (return (not (exists s in (siblings p)
                     where (> (age s) (age p))))))
```

The generated iteration that implements the `exists` predicate above is as efficient as the previously shown `foreach` loop; however, since the generated loop is a statement rather than an expression, it cannot be an argument to the `return` operator which expects a value returning expression. To translate this, STELLA uses a special construct called `VRLET` (for value-returning `let`) which percolates the procedural code outwards, saves its result value in a temporary variable, and folds the return expression inside. The resulting translation is very efficient, but it is not quite as close to the source code as was the case in previous examples. This is one of the few cases where readability of the generated translation has been somewhat sacrificed for the sake of expressiveness.

STELLA uses *iterators* to facilitate iteration over arbitrary data structures. A variety of built-in iterators are provided, for example, the following loop iterates over all classes in a module:

```
(foreach cl in (all-classes NULL TRUE)
    do (print cl EOL))
```

Here is the corresponding C++ translation which uses a `next?` method (translated as `nextP`) to bump the iterator, and its `value` slot to access the current loop value (iterators can have more than one value):

```
{ Class* cl = NULL;
  Iterator* iter_029 = all_classes(NULL, TRUE);
  while (iter_029->nextP()) {
    { cl = ((Class*)(iter_029->value)); }
    cout << cl << endl; }}
```

The iterator protocol is part of the STELLA language definition which enables the user to use standard `foreach`-style iteration to iterate over arbitrary user-defined data structures. If a `foreach` is used on a collection that is neither an iterator nor one of the built-in STELLA collections, a standard method `allocate-iterator` is called on it to convert it into an iterator. This method can be specialized by the user to allocate iterators on any user-defined data structures.

## Runtime Type Inference

Not always is it possible or convenient to rely on static type information alone. In particular, the handling of heterogeneous collections is a somewhat thorny issue in statically typed languages such as C++, while it is rather effortless in Common Lisp. Since such collections are sometimes needed, e.g., to store the results of reading and parsing user input, STELLA provides a runtime type system that can be used to determine object types and subtype relationships at runtime. A Lisp-like `typecase` construct makes it particularly convenient to encode specialized processing based on an object's type. Purists of the object-oriented programming persuasion might frown upon such programming style, but we believe that the resulting code is often cleaner and easier to understand than equivalent code that relies solely on standard object-oriented means such as dynamic method dispatch. It also allows one to avoid high-level catch-all methods which in C++ are somewhat space inefficient, because of the nature of its method dispatch mechanism.

## Rapid Prototyping

Since evolutionary development is such an essential ingredient of AI programming, an important question to answer is how well STELLA supports this particular programming style. One important contribution is how it minimizes the need for code maintenance. For example, its uniform iteration syntax allows one to experiment with different collection representations without having to maintain associated iteration code. Its uniform syntax for function call, method call, and slot access makes it possible, for example, to change a method into a function or a storage slot into an access method without having to change any of the call sites of the function or slot (provided, the basic signature has not changed). In C++ this is not the case, since it exposes the type of an entity by the syntax that is used. Type inference also plays an important role by automatically maintaining the types of local variables. STELLA places minimal restrictions on the placement of declarations by utilizing a two-pass translation scheme. This enables the programmer to position definitions where they most naturally belong, without having to use redundant forward declarations. The biggest support for rapid prototyping, however, comes from leveraging existing Lisp development environment technology which provide powerful hypercode environments for incremental code development. In such environments it *is* possible to incrementally define and redefine STELLA functions and classes by piggybacking on the dynamic nature of Lisp, even though this feature is not directly addressed by the STELLA language itself. STELLA provides a set of Lisp macros for the definition of functions, classes, etc. which call the STELLA translator behind the scenes and then send the resulting translation to the underlying Lisp system for evaluation or compilation. This allows one to use the exact same incremental code development process as is common for standard Lisp code, such as evaluation or compilation from an editor buffer, changing definitions on-the-fly during debugging, looking up source code, automatic code indentation, etc. Once development has completed, one can "push a button" to generate a final C++ or Java production version of the program.

## Discussion

To date we have written approximately 70000 lines of STELLA code and successfully released Common Lisp and C++ versions of the knowledge representation system that motivated the development of STELLA. We feel that STELLA development is almost as effortless as Lisp development, and that it would have been vastly more difficult and tedious to write the system directly in C++ or Java. Maybe the most convincing evidence for the readability of the translated STELLA code is that we are able to use standard Lisp, C++ and Java debuggers and inspectors to debug STELLA code and inspect its data structures. The safety benefits of the type system have proven to outweigh by far the additional burden it puts on the programmer. Being both veteran Lisp programmers, we now miss the STELLA type system when we write the occasional Lisp program directly. On average, the C++ translation of a STELLA program runs about three to five times faster than its Lisp translation which executes roughly isomorphic code. This speed difference is mainly explained by the somewhat less efficient slot access and method dispatch of CLOS. In fairness to Lisp we have to say that the nature of the Lisp translations generated by STELLA put it somewhat at a disadvantage compared to programs that were written for Lisp directly. The Java translations run about as fast as Lisp, but the Java translator has been completed only recently, and we expect to be able to significantly improve the performance of the generated code. The implementation of STELLA is fairly complete, but a few features still need finishing, and a few language issues still need to be tidied up.

## Conclusion

STELLA demonstrates that it is possible to smoothly integrate strong typing and the Lisp programming paradigm. Apart from facilitating the translation into other strongly-typed languages, the STELLA type system combined with its type inference facility also proves to be

an important tool to aid the programmer in the process of rapid development and maintenance of AI software. Many of Common Lisp's features were found essential to symbolic programming and also made their way into STELLA. Most prominently absent is Lisp's support for dynamic redefinition, which mainly aids code development (which is where it is also exploited for STELLA), but rarely is needed in a finished application.

## Acknowledgements

The development of STELLA was sponsored by the Defense Advanced Research Projects Agency under contract N00014-94-C-0245. Thanks go to Eric Melz for the development of the STELLA-to-C++ translator and to Tom Russ for the development of the STELLA-to-Java translator.

## References

Balzer, R. 1990. AI and software engineering: will the twain ever meet? In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1123–1125. Cambridge, MA: MIT Press.

Boehm, H. 1993. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. SIGPLAN Notices 28, 6 (June 1993), 197-206.

Gosling, J.; Joy, B.; and Steele, Jr, G. L. 1996. *The Java™ Language Specification*. Reading, MA: Addison Wesley.

McCarthy, J. 1981. History of LISP. In Wexelblat, R., ed., *History of Programming Languages*, ACM Monograph Series. New York: Academic Press. 173–197.

Meyer, B. 1992. *Eiffel: The Language*. Object-Oriented Series. New York, NY: Prentice Hall.

Myers, A. C.; Bank, J. A.; and Liskov, B. 1997. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 132–145.

Shalit, A. 1996. *The Dylan™ Reference Manual*. Reading, MA: Addison Wesley.

Shrobe, H.; Fahlman, S. E.; Stroustrup, B.; and Steele, Jr, G. L. 1996. The future of Lisp. *IEEE Expert* 11(1):10–16.

Steele, Jr, G. L. 1990. *Common Lisp*. Bedford, MA: Digital Press, second edition.

Stoutamire, D., and Omohundro, S. 1996. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA.

Stroustrup, B. 1991. *The C++ Programming Language*. Reading, MA: Addison Wesley, second edition.