From Lisp S-expressions to Java source code

Article *in* Computer Science and Information Systems · December 2008

DOI: 10.2298/CSIS0802019L · Source: DBLP

1 author:

António Menezes Leitão
Technical University of Lisbon
37 PUBLICATIONS **78** CITATIONS

UDC 004.439:00.55

From Lisp S-Expressions to Java Source Code

António Menezes Leitão

Instituto Superior Técnico / INESC-ID
Lisboa, Portugal
antonio.leitao@dei.ist.utl.pt

Abstract. The syntax of Lisp languages is based on S-expressions, an extremely simple form of structured data representation that is nevertheless fundamental to the development of Lisp syntactic extensions. By adopting a more conventional syntax, the Java language placed itself in a difficult position in regard to user-defined syntax extensions. In spite of the many efforts to provide mechanisms for such extensions, they continue to be more difficult to use than S-expression-based ones. In this paper, we will describe the use of the S-expression syntax in a Java code generation environment. By providing an S-expression based program representation for Java source code, we are able to reuse and extend Lisp macro-expansion techniques to significantly simplify the construction of Java programs.

Keywords: S-expressions, Macros, Common Lisp, Java.

1. Introduction

S-expressions (*Symbolic Expressions*) were invented by John McCarthy in the late 50s as a notation for both programs and data in the language Lisp [16].

S-expressions are usually expressed as fully-parenthesized prefix notation (also known as Cambridge Polish notation). For example, the mathematical expression 1+2×3 is represented in S-expression notation as `(+ 1 (* 2 3))`. Data can also be easily represented and there are old and recent proposals for its standardized use [17] [20].

McCarthy's original idea also suggested the use of a different, more ALGOLesque, format named M-expressions (*Meta Expressions*), that would be translated into S-expressions. In M-expression notation, the previous expression looks like `+[1, *[2, 3]]`. However, programmers started to use and appreciate the S-expression format and M-expression never caught up.

Homoiconicity is one of the fundamental ideas behind the S-expression notation. A language is said to be homoiconic when the primary representation of a program source code is implemented using a primitive type of the language itself. Thus, in a homoiconic language, a program can be constructed, analyzed, and evaluated using the programming language itself.

Lisp is the best known example of a homoiconic programming languages but there are other examples such as Prolog, SNOBOL and Tcl.

The ability to write programs that write programs is the hallmark of *meta-programming* [12]. A meta-program is written in a meta-language and its execution generates programs written in an object-language. In the case of homoiconic languages, such as Lisp, where programs and data are represented using the same S-expression notation, the meta-language and the object-language can be the same.

From its inception Lisp was extensively used for meta-programming, including writing self-modifying programs. However, these programs were perceived as generally difficult to debug and with performance problems. As time went by, simpler and more standardized forms of meta-programming were developed and one, in particular, became well established: *macros* [19].

The word “macro” has a somewhat dubious reputation due to the problems of their use in the C programming language but that reputation is totally undeserved when we talk about Lisp. Macros in C are processed using a special program (a pre-processor) that operates as a text-replacement tool that does not understand the syntax of the language and that might create problems that are hard to debug.

In Lisp, a macro describes a program that accepts program fragments (as S-expressions) and computes a new program fragment (an S-expression) that is evaluated in place of the macro call. This means that Lisp macros do not deal with the program text but with the program syntax tree instead.

For performance reasons, the evaluation of the macro call and the evaluation of the new program fragment produced by that call occur at different times (called, respectively, *macro-expansion time* and *run time*) and, in most implementations, the new program fragment *replaces* the original macro-call so that there is only one macro-expansion for each call.

Macros have been used in Lisp languages for quite a long time (at least, since 1963 [11]) and are fundamental for syntactically extending the language. In dialects such as Common Lisp [2], a significant part of the language is implemented using macros.

As an example of the use of a macro, consider the following program fragment in Common Lisp that reads the first line of text contained in the given file:

```
(with-open-file (s "/tmp/file.txt")
  (read-line s))
```

The macro with-open-file is responsible for opening the file, executing the read-line operation requested and, in the end, closing the file and returning whatever was read. This behavior, however, is not implemented by the macro itself but by its macro-expansion, i.e., by the S-expression that is computed by the macro call. In fact, after macro-expansion time, what is really evaluated is the form:[1]

```
(let ((s (open "/tmp/file.txt")))
  (unwind-protect
      (read-line s)
    (when (streamp s)
      (close s))))
```

This sort of code generation can be very easy to do when we combine the Lisp macro system with a template-based *quasiquotation* approach [5]. Quasiquotation allows us to provide the following definition for the with-open-file macro:

```
(defmacro with-open-file ((f filename) form)
  `(let ((,f (open ,filename)))
     (unwind-protect
         ,form
       (when (streamp ,f)
         (close ,f)))))
```

The idea behind quasiquotation is that it operates as a parameterized version of quotation and is generally used to describe templates for code generation where some “unquoted” parts (those proceeded by commas) will be filled in by evaluating the corresponding expression. Given the fact that macros receive their arguments unevaluated (as S-expressions) and that they must compute an S-expression as result, it is extremely tempting to use quasiquotation in macros and, in fact, they are heavily used for that purpose. The macro/quasiquotation combination is one of the best features of Lisp and has been appropriately called “the ultimate CASE tool” [3].

In many cases, macros do nothing more than construct an S-expression from a template described by quasiquotation that incorporate the arguments to the macro call. The with-open-file example presented above shows this behavior, as can be seen in the `(read-line s)` argument that is carried over

[1]We present only a simplification of the expansion that is done in most implementations of this macro.

the expansion without any changes. In these cases, the arguments are treated as completely opaque objects.

In some other cases, however, it is necessary to look inside those arguments. For example, the extended loop macro provided in Common Lisp is used to express iterations and it expands into different forms depending on the presence of certain symbols in the macro arguments, such as from, in, and while, so that a `(loop for i from ... to ... do ...)` and a `(loop while ... do ...)` can produce different expansions.

As another example, consider the macro that increments a place: `(incf x)` is the same as `(setf x (+ x 1))`. However, if the macro argument is more complex, e.g., in `(incf (first (foo)))`, the naive application of the macro will generate the form `(setf (first (foo)) (+ (first (foo)) 1))` that will incorrectly evaluate `(foo)` twice. To solve this problem it is necessary to look inside the argument in order to generate a program that avoids duplicate evaluation.

In other cases, the situation is even more complex. Sophisticated syntactic extensions, such as the *Series* ([23]) and *Iterate* ([1]) packages, must extensively analyze and rearrange the source code that is passed as argument. For an even more extreme case, consider *Screamer* ([21]), a non-deterministic variant of Common Lisp that depends on a few macros to convert a program into *continuation-passing-style* and that also (incrementally) operates a whole-program analysis to distinguish deterministic and non-deterministic functions. The common theme across the previous examples is that macros, sometimes, must do a lot more than the simple template instantiation provided by quasiquotation.

In spite of its long history, there are still some problems associated with Lisp macros (unintended variable capture, out-of-order evaluation, etc) and some of these problems have been solved, e.g., by the *hygienic macros* [9] and *syntactic closures* [6] that have been proposed for the Scheme dialect of Lisp. Common Lisp, however, still uses the traditional model because it is simpler and it does not entail the same problems that it does in Scheme.

In the rest of this paper, for reasons that will be obvious, we will only consider the Common Lisp model.

2. The Linj Language

System Software maintenance is an highly difficult task, in particular, when the software is written in one programming language but the maintenance team prefers to develop in a different language. To deal with this problem, management tends to restrict the set of “acceptable” programming languages to the most widely used ones such as Java. This imposes a difficult constraint on developers that prefer to work in less mainstream languages such as Common Lisp.

Linj is a Common Lisp-like language intended to be translated into human-readable Java source code. The fundamental idea behind Linj is that it should

be possible to develop a program in a Common Lisp dialect but deliver it in Java just like if the program was originally written in Java. Linj was designed by carefully selecting some of the best features of Common Lisp and avoiding or slightly changing those that are difficult to translate into readable Java source code.

A preliminary version of Linj was described in [7] and a real example of its use was presented in [8]. The paper [13] explains the use of Linj in a reengineering setting and [14] discusses the round-trip process from Java to Linj. In this paper we will focus on the meta-programming capabilities of Linj for Java code generation, a topic that was only superficially addressed in the previous papers. We will now present a short overview of the Linj language.

As a first example of a Linj program, we will consider the typical factorial function:

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

The previous definition is written using the exact same syntax that would be used in the Common Lisp language. However, instead of being used by a Common Lisp evaluator, the previous definition is translated by the Linj compiler into the following Java source code:

```
public static Bignum fact(Bignum n) {
  if (n.compareTo(Bignum.valueOf(0)) == 0) {
    return Bignum.valueOf(1);
  } else {
    return n.multiply(fact(n.subtract(Bignum.valueOf(1))));
  }
}
```

It is important to note that, as a first approximation, the Linj compiler generates Java code that preserves the semantics of the original Common Lisp program. In this particular example, the presence of arithmetical operations allowed the compiler to infer that the factorial function accepts a number as argument and returns a number as result. However, it could not infer a more specific type for these numbers (such as integer or floating point)

and, consequently, the generated program will depend on a run-time library that implements a generic (arbitrary large) rational number called `Bignum`.[2]

It is possible, however, to include in the program more specific type information about the function parameter `n`. If, instead of writing the above fact function definition, the programmer writes

```
(defun fact (n/long)
  (if (= n 0)
    1
    (* n (fact (1- n)))))
```

then he is also asserting that the parameter `n` is a `long`. This allows Linj to make a more aggressive type inference to conclude that the function also returns `long`s. This extra aggressiveness is justified because it allows Linj to generate code that more closely resembles a human-written code fragment, as can be seen below:

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

As another example, consider the following Linj function that computes the biggest of three integer arguments:

```
(defun max-x-y-z (x/int y/int z/int)
  (let ((max-x-y (if (> x y) x y)))
    (if (> max-x-y z)
      max-x-y
      z)))
```

The Linj compiler translates the previous function into the following Java fragment:

[2]The compiler can be trivially configured to use different Java classes and primitive types in the generated code, e.g., BigInteger in the place of Bignum.

```
public static int maxXYZ(int x, int y, int z) {
  int maxXY = (x > y) ? x : y;
  if (maxXY > z) {
    return maxXY;
  } else {
    return z;
  }
}
```

3. Linj Syntax

Linj compiles source code that is very similar to Common Lisp to target code that is very similar to human-written Java. Syntactically speaking, Linj programs are so similar to Common Lisp that the Linj compiler inputs them using the exact same read function that is provided by Common Lisp. This was an important design decision because it allowed us to effortlessly reuse all the Common Lisp machinery for read-macros.[3] Just like in Common Lisp, reading a Linj program produces an S-expression.

S-expressions are the basis for Common Lisp macro capabilities but they are not enough for the full set of Linj macro capabilities. Common Lisp has an extremely simple syntax where forms can be categorized as symbols, conses or self-evaluating objects. On the other hand, Java has a very complex syntax with a large number of syntactical categories, including expressions, statements, blocks, compilation units, etc. Moreover, Common Lisp allows forms to be combined in arbitrary ways while Java has much more strict rules regarding the grammatical combination of expressions, statements, blocks, class declarations, etc. For example, contrary to Common Lisp, expressions in Java cannot contain variable declarations.

In order to reduce the gap between the syntaxes of Common Lisp and Java, Linj imposes additional syntactical constraints upon the S-expression forms, allowing the same S-expression to be classified according to the place where it occurs. This should be evident in the previous `max-x-y-z` example, where the first if form is classified as an expression and translated into Java's

[3] In spite of being frequently seen together, read-macros have little in common with (normal) macros, as they just implement specialized parsing behavior triggered when the reader encounters the corresponding macro character.

conditional expression and the second one is classified as a statement and translated into Java's if statement.

Although it might seem that increasing the complexity of S-expressions is a step back relative to the uniform syntax of Lisp, the fact is that, in Linj, macros can also take advantage of the rich syntax and semantics of the Java language to provide a more expressive macro system.

In general, the Linj syntax accepts the large majority of Common Lisp programs but there are some cases where a correct Common Lisp program is rejected. To understand the issue, we present a fragment of the Linj grammar:

<*statement*> ::= <*if statement*> | <*let statement*> | …
<*if statement*> ::= (if <*expression*> <*statement*> <*statement*>)
<*expression*> ::= <*literal*> | <*reference*> | <*if expression*> | …
<*if expression*> ::= (if <*expression*> <*expression*> <*expression*>)

Note, in the above grammar, that the same `if` form can be classified both as a statement or as an expression. Other forms, such as the `let`, can only be parsed as statements. Now, let's consider the following Common Lisp forms:

```
(if (let ((w (max y z)))
      (> x (* w w)))
  (+ x y)
  (+ x z))
```

```
(let ((w (max y z)))
  (if (> x (* w w))
    (+ x y)
    (+ x z)))
```

Although both forms are valid Common Lisp forms, only the form on the right is a valid Linj form. The form on the left isn't valid in Linj because an `if` (be it expression or statement) expects an <*expression*> as its first argument and the `let` is a <*statement*>.

Although the Linj grammar seems like a severe restriction to a Common Lisp programmer, using a more restricted syntax is an important advantage for a translation process that wants to generate *readable* code and, in fact, the Linj grammar was carefully designed to describe Common Lisp sources that can be effectively translated into *readable* Java code.

To use the Linj grammar we implemented a second parsing process (using a recursive descent parser with backup) that takes the S-expression produced by the Common Lisp parser and constructs abstract syntax trees (ASTs) where the nodes are instances of CLOS [18] classes related to the different syntactical categories. Each parse rule describes an S-expression-based pattern that will match an S-expression-based program: the parser operates over the S-expressions constructed by the Common Lisp reader and not over the textual representation of programs.

Both the S-expression based program representation and the AST based program representation are essential for Linj macro capabilities and, to this end, each AST node contains a reference to the S-expression that it represents. In the next section we discuss Linj macro capabilities.

4. Linj Macros

A Common Lisp macro is a Common Lisp program that accepts S-expressions as arguments and that generates Common Lisp code fragments as results.

Linj would not be a Common Lisp-like language if it didn't implement Common Lisp-like macros. However, besides providing the traditional Common Lisp macros, Linj also allows two other types of macros that take advantage of the richer syntactic and semantic information that is available in the Linj ASTs.

We will now explain these three different types of macros.

4.1 Traditional Macros

The first form of macro in Linj is indistinguishable from a Common Lisp macro. In fact, the body of the macro is written in Common Lisp. The macro expansion, however, entails a subtle difference: the same Linj macro call might be expanded more than once.

In Common Lisp, macro expansion occurs when a macro-call is encountered while processing forms that are meant to be evaluated. In Linj, macro expansion occurs during the parsing phase and has to be repeated whenever the parser backtracks past the macro-call.

To understand this issue, consider an hypothetical implementation and use of the anaphoric `if` macro[10] (we will call it `aif`). The idea is to introduce the locally scoped pronoun it that is lexically visible in the `if` branches and that stands for the result of the `if` test expression. Here is one example of its use:

```
(aif (long-computation)
  (princ it)
  (princ "Failed!"))
```

and here is the macro definition:

```
(defmacro aif (test-form then-form else-form)
  `(let ((it ,test-form))
     (if it
       ,then-form
       ,else-form)))
```

Using this macro, Linj is capable of expanding the previous macro call into

```
(let ((it (long-computation)))
  (if if
    (princ it)
    (princ "Failed!")))
```

and then it will translate the macro-expanded code into

```
Object it = longComputation();
if (it != null) {
  System.out.print(it);
} else {
  System.out.print("Failed!");
}
```

Now, let's imagine that the Linj parser is trying to parse the `aif` macro call as an expression. As we just saw, after expanding the macro call, the result is a let form that introduces a new variable declaration.

Unfortunately, a Java expression cannot contain variable declarations and, by extension, neither can a Linj expression, so the parser backtracks as far as necessary, undoing all macro-expansions, and attempts other parsing rules, including the one that treats the entire `aif` form as a statement, causing re-expansion of the macro call whose result can now be correctly parsed.

For this reason, the Linj (macro-)programmer should be aware that macros should be side-effect free so that their macro calls can be expanded more than once. This is not a problem in practice because, due to the different phases of macro expansion and evaluation, it is extremely rare, even in Common Lisp, to find a macro that causes side effects.

As another example, consider the Linj version of the `with-open-file` Common Lisp macro that was presented in the introduction. Given the fact that Java deals with files using a decorator design pattern, Linj provides a more sophisticated macro definition that allows the specification of the stream class to use for reading from and writing to files and also allows the specification of an arbitrary composition of stream decorators. Here, for illustrative purposes, we will present a simpler implementation that only provides the instantiation of two different classes, namely, `file-input-stream` and `file-output-stream`:

```
(defmacro with-open-file ((var filename
                           &key (direction :input))
                           &body body)
  `(let ((,var
          ,(ecase direction
             (:input `(new file-input-stream ,filename))
             (:output `(new file-output-stream ,filename)))))
     (unwind-protect
         (progn ,@body)
       (unless (eq ,var null)
         (close ,var)))))
```

The next Linj fragment shows a typical use of the previous macro where we open a file for input, another one for output, we read an S-expression from the first and we write it in the second:

```
(with-open-file (in "f1" :direction :input)
  (with-open-file (out "f2" :direction :output)
    (write out (read in))))
```

Its translation into Java is the following:

```
FileInputStream in=new FileInputStream("f1");
try {
  FileOutputStream out=new FileOutputStream("f2");
  try {
    out.write(in.read());
  } finally {
    if (out != null) {
      out.close();
    }
  }
} finally {
  if (in != null) {
    in.close();
  }
}
```

Note how the simple three-line Linj example is translated into a combination of `try-finally` statements that ensure that streams are properly closed even in the event of abnormal exits.

4.2 Context Sensitive Macros

Sometimes, Linj programmers want to write macros that produce different expansions depending on the syntactic context where the macro call occurs. Consider, for example, a (very) simplified implementation in Linj of a `sprintf` function that accepts any number of arguments whose textual representation will be interspersed in the middle of a string at the positions indicated by a `%` marker.

It is easy to define a first version that depends on Java's `StringBuffer` class:

```
(defmacro sprintf (str &rest args)
  (let ((parts (split-sequence #\% str)))
    `(let ((buff (new 'string-buffer ,(first parts))))
       ,@(loop for part in (rest parts)
               collect `(append buff ,(pop args))
               collect `(append buff ,part))
       (to-string buff))))
```

This macro is used, for example, in:

```
(defun foo (x y)
  (sprintf "I have % apples and % oranges" x y))
```

After macro expansion and translation into Java, we get:

```
public static String foo(Object x, Object y) {
  StringBuffer buff = new StringBuffer("I have ");
  buff.append(x);
  buff.append(" apples and ");
  buff.append(y);
  buff.append(" oranges");
  return buff.toString();
}
```

The macro seems useful but, unfortunately, it does not work in the following example:

```
(defun bar (x y)
  (length (sprintf "I have % apples and % oranges" x y)))
```

The reason for not working is that the macro expansion includes a `let` form that, as we said before, can only be parsed as a statement. However, in the above example, the macro call occurs in a position (call argument) where an expression is expected.

To solve this problem, we can provide a different macro that guarantees that the macro expansion can be parsed as an expression. Here is one possibility:

```
(defmacro sprintf (str &rest args)
  (let ((parts (split-sequence #\% str)))
    `(concat ,(first parts)
       ,@(loop for part in (rest parts)
               collect (pop args)
               collect part))))
```

Using this version, the `bar` function is translated into:

```
public static int bar(Object x, Object y) {
  return ("I have " + x + " apples and " +
          y + " oranges").length();
}
```

Given the fact that both macro definitions have advantages we don't want to be forced to prefer one over the other. Fortunately, the Linj macro system allow us to have *both* `sprintf` macro definitions available at the same time as long as we *tag* them with the syntactical category that is expected for the macro expansion, allowing the parser to choose the macro definition that is most appropriate for the parsing situation at hand. This is a very important feature to allow the best human-readable Java code generation for every conceivable situation. The tag is a symbol that names the intended syntactical category and is placed between the macro name and its list of parameters.

This feature is also very important for error reporting during the compilation process. To understand this issue, let's suppose that one programmer defines a macro that expands into a form that must be parsed as a statement but another programmer uses that macro in a context where an expression was expected. In this case, a parsing error is generated but the error is related to the expanded code and not to the original macro call, thus making it more difficult to the programmer to understand the error message. However, if the macro developer had tagged his macro with the appropriate syntactical category, then the macro expansion would not be attempted and the error message would simply report that it was not possible to parse the macro call in a context where an expression was expected, thus making it much more easier for the macro user to understand the problem.

4.3 Semantic Macros

Semantic Macros allow Linj to go one step further: they have the same syntax and same conceptual model as syntactical macros but they operate not on the S-expression representation as traditional and syntactical macros do but on the Linj AST instead. This gives them an additional power to analyze the AST and generate cleverer expansions.

In this section we will focus on semantic macros that only explore type information but, in practice, these macros can explore all the information that is available in the AST. To this end, the Linj compiler provides an API that includes functions for walking the AST, for inspecting its nodes, for obtaining the type of the expressions, etc.

As a first example, consider the Common Lisp function `logbitp`: it accepts an index and an integer and it tests the value of the indexed bit in the two-complement binary representation of the integer. Java's `BigInteger` method `testBit(i)` is similar: it accepts an integer parameter `i` and returns `true` if the *i*-th bit of the receiver is `1` and `false` otherwise.

Unfortunately, there is no similar method for Java primitive integer types `int` and `long`, not even in some utility class such as `java.lang.Math`, thus forcing the programmer to use a combination of shifting and masking to achieve the same effect. This is bad because it makes the conversion of code between primitive integer types and reference integer types more difficult than it needs to be.

To solve this problem, the Linj programmer can define a macro but this macro must be very different from the previous ones because, this time, the macro expansion does not depend on the syntactical form of the arguments neither does it depend on the syntactical category of the macro call. Now, the macro-expansion depends on the *type* of one of the arguments. To explore this type information, the Linj API exposes the `get-type` function. This function accepts an AST node that represents an expression and returns the (static) type of the expression, using type inference to derive that information.

Using this API, we can define a `logbitp` macro that accepts two expressions, the first one evaluating to an index and the second one to an integer. The macro computes the type of the second expression, distinguishes between the primitive types and the reference type `BigInteger` (using the predicates `primitive-type-reference-p` and `big-integer-type-p`, respectively) and returns an appropriate macro-expansion. Here is one possible definition:

```
(def-linj-macro expression
    (logbitp ?e1/expression ?e2/expression)
  (let ((type2 (get-type ?e2)))
    (cond ((primitive-type-reference-p type2)
           `(not (zerop (logand ,?e2 (ash 1L ,?e1)))))
          ((big-integer-type-p type2)
           `(test-bit ,?e2 ,?e1))
          (t
           (fail)))))
```

Note that if the type does not pass the macro tests, meaning that the second argument has a type that is neither primitive nor a `BigInteger`, then the macro *fails*, i.e., it declines to expand.

Now, let's consider the following example that uses the macro in two different places:

```
(defun baz (x/long y/big-integer)
  (eq (logbitp 5 x)
      (logbitp 5 y)))
```

In the previous example, it is obvious that the macro cannot syntactically distinguish between the two calls. However, due to the introspective capabilities explored by the macro, Linj is capable of translating the previous function into the following equivalent Java code:

```
public static boolean baz(long x, BigInteger y) {
  return ((x & (1L << 5)) != 0) == y.testBit(5);
}
```

It is worth mentioning that Linj's semantic macros go beyond what Common Lisp macros can usually do. This isn't a Common Lisp shortcoming but a consequence of the dynamic nature of the language: Common Lisp is a dynamically typed language, meaning that there is very little type information available at macro-expansion time. Linj, on the other hand, is as statically typed as Java, allowing semantic macros to explore much more semantic information.

As a final example, consider the use of a `for-each` macro that iterates different kinds of objects, as exemplified below:

```
(defun iterate (x/iterator)                  ;;an iterator
  (let ((y (new 'string-tokenizer "1 2 3")) ;;an enumeration
        (z #(1 2 3)))                        ;;a vector
    (for-each (e x)
      (princ e))
    (for-each (e y)
      (princ e))
    (for-each (e z)
      (princ e))))
```

The difficulty with the `logbitp` and `for-each` macros is that their expansion depends not on the syntactical form of the arguments (in the examples, they are indistinguishable) but on their semantic properties instead. In this case, it depends on the *type* of the iterated expression, that is, the types of `x`, `y` and `z` but this information can only be made available after parsing the S-expression and annotating the resulting AST with the types computed for the expression nodes. This means that the `for-each` macro expansion must be delayed until the AST is ready to provide the information it might need.

To achieve this effect, macro calls for these *semantic macros* are not expanded at parse time, thus becoming AST nodes themselves. Later on, a tree visitor responsible for expanding these macro calls is activated and all macro calls are expanded in a top-down fashion.

During the macro-expansion, what the macro receives as arguments is either S-expressions or completely parsed AST sub-trees that can be analyzed, manipulated, and reused at will. The macro specifies what it expects to receive by *tagging* each parameter with the syntactical category it wants (or none if it just wants the S-expression). If, during one expansion, information regarding other (semantic) macro calls is needed, its expansion is also computed, thus triggering a kind of chain reaction.

The final phase of the expansion is the generation of an S-expression that can also include fragments of the already parsed AST and that is subsequently parsed to compute a new AST subtree that replaces the semantic macro call. Obviously, to compute this expansion, one might use the same quasiquotation that is used in traditional and syntactical macros.

We will now complete the previous example, showing a possible implementation of the for-each macro:

```
(def-linj-macro statement
    (for-each (?var ?form/expression) . ?body)
  (let ((form-type (get-type ?form)))
    (cond ((array-type-reference-p form-type)
           `(dovector (,?var ,?form) . ,?body))
          ((super-type-p (iterator-type) form-type)
           `(let ((iter ,?form))
              (while (has-next iter)
                (let ((,?var (next iter))) . ,?body))))
          ((super-type-p (enumeration-type) form-type)
           `(let ((enum ,?form))
              (while (has-more-elements enum)
                (let ((,?var (next-element enum))) . ,?body))))
          (t
           (error "Unknown type for iteration ~A" form-type)))))
```

Note that the macro definition requires the form parameter to be parsed as an expression. Using this expression, the macro computes its *type* using the function `get-type`: this is an entry-point for the type inferencer that, given an expression, returns the *static* type of the value of the expression; it is guaranteed that its dynamic type will be a subtype of this static type. Depending on this type, the macro then expands into different forms that will be used in place of the macro call.

For the example given above, the three syntactically identical uses of the macro produces three completely different expansions, each dealing with a different *type* of iterated object:

```
public static void iterate(Iterator x) {
  StringTokenizer y = new StringTokenizer("1 2 3");
  int[] z = new int[] { 1, 2, 3 };
  Iterator iter = x;
  while (iter.hasNext()) {
    Object e = iter.next();
    System.out.print(e);
  }
  StringTokenizer enum = y;
  while (enum.hasMoreElements()) {
    Object e = enum.nextElement();
    System.out.print(e);
  }
  int limit = z.length;
  for (int i = 0; i < limit; ++i) {
    int e = z[i];
    System.out.print(e);
  }
}
```

The previous example also demonstrates the usefulness of user-defined syntax extensions: Linj programmers have been using the `for-each` macro since Java 1.1, while Java programmers had to wait until Java 5 for a similar syntax extension provided by the Java language itself. Another advantage is

that just by changing the macro definition, the same Linj programs can now be simply recompiled to generate Java sources that take advantage of the Java 5 `for-each` statement.

Given the fact that Linj macros can depend on the available type information and that the type inference mechanism depends on the expansion of macros, it is possible to create circularities. These circularities will be detected by the Linj compiler and will be represented using a *cyclic type*. As a result, The Linj (macro) programmer should be prepared to deal with this type, either by aborting the macro expansion or by replacing the cyclic type with some other type or by using some other strategy.

A final important point about semantic macros in Linj is that they can decline to expand, meaning that they might not want to generate any expansion. In this case, Linj shadows the macro (so that it cannot be recursively applied to the same AST node) and re-parses the original S-expression form of the node. This is useful to provide partial evaluators that can generate more efficient code but only if they have sufficient static information available.

5. Related Work

There is a large number of proposals for including macro capabilities in syntactically rich languages such as C or Java. In most cases, there is a serious attempt to reuse in the meta-language the same (or a similar) language that is used in the object-language. In [24], a Lisp-inspired template approach for C is used but where macros are programmed in an extended C that is interpreted at macro-expansion time.

Semantic macros were also proposed in [15] in terms that are very similar to ours. The authors present a new experimental language—XL—that borrows its semantics from Scheme but whose syntax, although S-expression based, has many more syntactical categories. Again, this is very similar to our own approach (except that we borrowed our semantics from Common Lisp and Java). The biggest difference, however, is that the extra syntax is visible in the source code, making macro definitions harder to write and understand.

OpenJava [22] is a macro system for Java where programmers customize the definition of class meta-objects for describing macro expansions. OpenJava was designed to address the needs of semantic macros and, to this end, it provides an object-oriented representation of programs that include logical and contextual information.

The biggest difference between Linj and OpenJava is that OpenJava does not use any template-based approach, instead preferring to construct the macro expansion by hand. Linj allows both, although quasiquotation is more used because it is clearer.

Jak [4] is an extensible superset of Java with support for meta-programming. Jak is part of JTS (Jakarta Tool Suite), a set of tools aimed at the construction of domain-specific languages. JTS represents source code

using SSTs (*surface syntax trees*) and ASTs (that are semantically-checked and annotated SSTs). Jak also uses a template based approach but where the code fragments must be surrounded by keywords (named *tree constructors*) that express the intended syntactical categories. There are several such tree constructors expressing, among others, expressions, statements, method definitions, classes, etc.

In contrast, Linj does not need tree constructors because the (meta-)programmer can indicate, in the macro definition, any syntactical category that he wants and this category is used to restrict the triggering of the macro expansion and for parsing its result. In Jak, the meta-language is Java complemented with an API for processing the SSTs and ASTs.

In general, all the approaches that attempt to add meta-programming capabilities to mainstream programming languages suffer from the fact that they target non-homoiconic languages. This makes it difficult to use template-based quasiquotation and, in many cases, it forces the macro writers to manually construct syntactically valid program fragments, a task that is difficult, tedious and error prone. Moreover, given the fact that macro expansion is done at (or before) compile-time, the macros must be coded using a meta-language that cannot be the same as the object-language, thus making the process more complex.

What we think is the major difference between our approach and others is the fact that our *real* object-language is hidden from the Linj programmer. In fact, while using Linj, the programmer never sees Java. What he sees is Linj, that is, a Common Lisp with a slightly restricted syntax. Our meta-language is (unrestricted) Common Lisp but since there's no relevant differences between Common Lisp and Linj and, moreover, macros generally use quasiquotation that further hides the differences, few Linj (meta-)programmers are aware of the fact that they are using two different languages. As a result, Linj *seems* to be an homoiconic language while, in practice, it is not.

6. Conclusions

S-expressions are one of John McCarthy marvelous inventions that still are, almost 50 years later, one of Lisp most distinguishing features. Being an uniform representation for both code and data, S-expressions allow code to be treated as data and data to be treated as code. Lisp macros are the best tool to explore, in a disciplined way, this uniform treatment. In a macro call, the code in the macro arguments is treated as data and the data produced by the macro is treated as code. When used in combination with quasiquotation, macros become an extremely simple but powerful tool for meta-programming.

In this paper, we presented the Linj approach to meta-programming for Java. Linj is a Common Lisp-like language that superimposes a Java-like grammar on top of S-expressions in order to allow Linj programs to be translated into human-readable Java programs. Linj programs are parsed into S-expressions that, according to the Linj grammar, are then parsed into ASTs.

Meta-programming in Linj is obtained through the use of Linj macros that accept either S-expressions or ASTs and that generate Linj S-expressions that are further parsed into ASTs. In the end of the translation process, these ASTs are finally transformed into Java source code.

The implementation of macros in Linj is more complex than in Common Lisp but we believe we managed to preserve the look and feel of Common Lisp macros while allowing much more sophisticated macro operations, including exploring syntactic and semantic aspects of the macro call arguments or of the expected results. It is also possible for these macros to operate arbitrary transformations of the entire AST. This sophistication is needed in Linj to provide the best possible translation from Linj S-expressions to human-readable Java source code.

7. References

1. J. Amsterdam. The iterate manual. Technical Report AIM-1236, MIT Artificial Intelligence Laboratory, Oct. 6 1990.
2. ANSI and ITIC. American National Standard for Information Technology: programming language — Common LISP. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996. Approved December 8, 1994.
3. H. G. Baker. Critique of DIN Kernel Lisp definition version 1.2. Lisp and Symbolic Computation, 4(4):371–398, Mar. 1992.
4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In Proceedings Fifth International Conference on Software Reuse, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.
5. A. Bawden. Quasiquotation in Lisp. In Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999., pages 4–12, Jan. 1999.
6. A. Bawden and J. Rees. Syntactic closures. In Proceedings of the 1988 ACM Symposium on LISP and Functional Programming, Salt Lake City, Utah., July 1988.
7. A. M. L. . J. Cachopo. Translating Lisp into Java. In International Lisp Conference, San Francisco, USA, October 2002.
8. A. M. L. . J. Cachopo. Pre-ProCLessing: Embedding Lisp within Java. In International Lisp Conference, New York, USA, October 2003.
9. W. Clinger and J. Rees. Macros that work. In Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, pages 155–162, Orlando, Florida, January 21–23, 1991. ACM SIGACT-SIGPLAN, ACM Press.
10. P. Graham. On Lisp: advanced techniques for Common Lisp. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.

11. T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, Oct. 1963.
12. A. H. Lee and J. L. Zachary. Reflections on metaprogramming. j-IEEE-TRANS-SOFTW-ENG, 21(11):883–893, Nov. 1995.
13. A. M. Leitao. Migration of Common Lisp programs to the Java platform - the Linj approach. csmr, 0:243–251, 2007.
14. A. M. Leitao. The next 700 programming libraries. In International Lisp Conference 2007. Association of Lisp Users, april 2007.
15. W. Maddox. Semantically-sensitive macroprocessing. Technical Report CSD-89-545, University of California, Berkeley.
16. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine (Part I). Communications of the ACM, 3(4):184–195, 1960.
17. J. McCarthy. The common business communication language. In V. Lifschitz, editor, Formalizing Common Sense: Papers by John McCarthy, pages 175–186. Ablex Publishing Corporation, Norwood, New Jersey, 1990.
18. A. Paepcke. Object-Oriented Programming: The CLOS Perspective. The MIT Press, 1993.
19. K. M. Pitman. Special forms in LISP. In LISP Conference, pages 179–187, 1980.
20. R. L. Rivest. SEXP (S-expressions). Internet Engineering Task Force - Internet Draft, 1997.
21. J. M. Siskind and D. A. McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In AAAI, pages 133–138, 1993.
22. M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, Reflection and Software Engineering, LNCS 1826, pages 119–135. Springer-Verlag, July 2000.
23. R. C. Waters. Common Lisp: The Language, chapter Appendix A. Digital Press, 89.
24. D. Weise and R. F. Crew. Programmable syntax macros. In SIGPLAN Conference on Programming Language Design and Implementation, pages 156–165, 1993.

António Menezes Leitão was born in Portugal and received a B.Sc. in Mechanical Engineering, a M.Sc. in Electrotechnical Engineering and PhD in Computer Science from Instituto Superior Técnico/Universidade Técnica de Lisboa (IST). He is a professor at IST and a Researcher at Instituto de Engenharia de Sistemas e Computadores-Research & Development (INESC-ID) His research interests include software engineering, software maintenance, and programming language design.

Received: July 16, 2008; Accepted: November 17, 2008.