DIY Meta Languages with Common Lisp

Alexander Lier Kai Selgrad Marc Stamminger

Computer Graphics Group, Friedrich-Alexander University Erlangen-Nuremberg, Germany

{alexander.lier, kai.selgrad, marc.stamminger}@fau.de

ABSTRACT

In earlier work we described C-Mera, an S-Expression to C-style code transformator, and how it can be used to provide high-level abstractions to the C-family of programming languages. In this paper we provide an in-depth description of its internals that would have been out of the scope of the earlier presentations. These implementation details are presented as a toolkit of general techniques for implementing similar meta languages on top of Common Lisp and illustrated on the example of C-Mera, with the goal of making our experience in implementing them more broadly available.

CCS CONCEPTS

•**Software and its engineering →Source code generation; Preprocessors;** *Translator writing systems and compiler generators;*

KEYWORDS

Code Generation, Common Lisp, Macros, Meta Programming

ACM Reference format:
Alexander Lier, Kai Selgrad, and Marc Stamminger. 2017. DIY Meta Languages with Common Lisp. In *Proceedings of European Lisp Symposium, Brussels, Belgium, April 2017 (ELS'17),* 8 pages.
DOI: 10.475/123_4

1 INTRODUCTION

In this paper we describe techniques that we employed to implement C-Mera [19], a meta language for C (and C-like languages), embedded in Common Lisp. C-Mera provides a Lisp-like syntax for C, that is, it is not a compiler from Lisp to C, but from C written in Lisp-form to regular C, i.e. there is no inherent cross-language compilation. An exemplary C-Mera (C++) program that simply prints all of its command-line arguments looks as follows:

```
1  (include <iostream>)
2
3  (defmacro println (&rest args)
4    `(<< #:std:cout ,@args #:std:endl))
5
6  (function main ((int argc) (char *argv[])) -> int
7    (for ((int i = 1) (< i argc) ++i)
8      (println " - " argv[i]))
9    (return 0))
```

The mapping to C++ is straightforward for the most part, and readers familiar with Lisp will recognize that lines 3-4 show a very simple macro that is then used in the main function. For a more thorough description and many more examples see our earlier work [12, 17–19], note, however, that even the above code shows features not present in many projects similar to C-Mera, e.g. inline type annotations (i.e. pointers), idiomatic C shorthands such as the post-increment and seamless integration with standard Common Lisp macros.

On the side of the language's user the most important features of C-Mera are its flexibility and extensibility, especially via Common Lisp macros. Using those, custom abstractions can be built easily, and with zero cost at run-time, which is very important when working in high-performance application domains. As these abstractions are quickly and easily attained such meta languages can be a valuable tool for prototyping, research, and when working on tight deadlines. Examples from this point of view can be found in previous work on C-Mera and its application [12, 17–19].

In this paper we provide a more in-depth description of C-Mera from the language implementor's side. One of the key features of C-Mera from this vantage point is the simplicity of its architecture. Due to its embedding in Common Lisp (and adoption and thus exploitation of its syntax) the problem of defining a system suitable for highly involved meta programming in C and C-like languages is reduced to constructing C-programs from S-Expressions, pretty-printing the internal representation in form of C-code and configuring the Common Lisp environment such that any inconsistencies with our target languages are resolved appropriately. The implementation of these details is described on a much more technical level than the scope of previous work allowed.

We believe that summarizing these details and documenting the design decisions behind them can be valuable to projects with similar goals, even when applied to different target languages or application domains. Especially since most of the implementation details described are not tied to C at all, they can be applied to help construct other meta languages on top of Common Lisp. Section 2 also lists a few Lisp-based projects that follow a similar path as C-Mera and those could naturally find inspiration from this detailed description.

In the remainder of this paper we first provide context for our work, starting with C-Mera and similar Lisp-based approaches over works that employ similar concepts in other languages to more general compiler technology and how it is used towards the same ends (Section 2). Following that, we detail the design goals we set up for C-Mera and the evaluation process of a C-Mera program, from how the source is read over its internal representation to tree traversal during C-code generation (Section 3). We then describe our package setup in more detail, noting the intricacies of overloading C-Mera and Common Lisp symbols (Section 4). Finally, we provide some very technical details on how to find a balance between the idiosyncrasies of the Common Lisp and C-family syntaxes (Section 5) and conclude with a short summary (Section 6).

ELS'17, Brussels, Belgium

DOI: 10.475/123_4

2 RELATED WORK

Our description of techniques for implementing meta languages in Common Lisp is naturally founded on our experience of working on C-Mera. Following the initial description [19] that demonstrated meta programming for stencil computations, we showed how it can be used to provide higher-level programming paradigms to the realm of C-like languages [18]. We also presented two real-world use cases. Firstly, a domain-specific language for high-performance image-processing applications [17] and, secondly, how C-Mera can be used to explore a vast space of implementation variants of a given algorithm [12]. With this paper we go back towards describing our base system presented earlier [19], however, the focus of this work is not on a description of the concept and providing examples to illustrate its versatility, but on the actual, low-level implementation details and design choices involved in the process.

Convenient, fully fledged macros and therefore extensive and easy meta programming is most prominently featured in the Lisp family of languages such as Racket, Scheme and Common Lisp. For this reason, these languages are host to many similar projects: Parenscript [16] generates JavaScript, whereas C-Amplify [6], Cl-Cuda [22], Lisp/c [2], and C-Mera target C, C++ and similar C-syle languages (with varying degrees of language support and maturity). While reaping the benefits of straightforward embedding in a powerful host language, following this approach the language designer is not as free as when starting out from scratch.

Some rather new languages, such as Rust [13], are designed to also support Lisp-style macros. However, as long as such languages show a less uniform syntax larger-scale meta programming (in the example of Rust using procedural macros) comes at a higher cost of engineering.

Naturally, the more ambitious and free option to write a language from scratch is, in principle, always available. Specific tools, for example Yacc [8], Lex [11] or ANTLR [14], and libraries such as Spirit [7] can ease the process of constructing an appropriate parser. However, building a consistent language and implementing powerful meta-programming capabilities are still the responsibility of the language designer.

Extending an existing language for meta programming provides a more efficient solution. For example, MetaOcaml [4] provides facilities for multi-stage programming with OCaml. Another example is Terra [5], which provides meta-programming capabilities by utilizing Lua as the host language. Lua functions can be applied to adjust, extend, and write Terra code and the embedded code can reference variables and call functions defined in Lua. Terra's sytanx is based on Lua and processed with a just-in-time compiler and can optionally be modified further with Lua prior to eventual compilation.

Such approaches require considerable effort to be realized, especially when targeting syntactically hard languages (e.g. C++). Other approaches utilize available language resources that were originally not intended for meta programming on that scale. C++ Template Meta Programming [25] (TMP), for example, exploits the template mechanism for extensive abstractions. The demand for such abstractions is visible from the field started by this work [1, 24], especially in the face of it being generally considered very hard [6, 10, 20].

In contrast, utilizing Lisp as an code generator is generally a straightforward task, but unleashing the full potential of Lisp's built-in functions and macro system while allowing convenient and naturally written input code (from the perspective of the C, as well as the Lisp programmer) can become rather tricky. In the most naïve approach, every syntactical element of a meta language implemented in, e.g., Common Lisp, will be mapped to S-Expressions, leading to highly verbose code. This can go on well unto the level of specifying how symbols have to be rewritten [2]. Sections 4 and 5 will describe the compromise found during C-Mera's implementation to have more free-form code while not suffering a loss in generality.

3 EVALUATION SCHEME

In this section we first define the most important characteristics that we wanted our meta language to exhibit (Section 3.1). We then provide a short overview of our intended syntax and mode of evaluation, exemplified with C-Mera (Section 3.2). Following that we describe the evaluation scheme in more detail, starting with how the internal representation is constructed (Section 3.3), kept and finally written out again (Section 3.4).

3.1 Design Goals

The most fundamental requirement for our language was being able to seamlessly interact with Common Lisp's macro system. This way we ensured that it is meta programmable to the same degree and not limited, e.g., to some specific form of templating. Interaction with Common Lisp's macro system also entails that writing our own macro-expansion routines was never intended, that is, our problem statement is much simpler as it seems at first glance. We also wanted to provide a system as accessible to C-programmers as possible, given our primary objective. For us, this entails to have the language properly keep the case of symbols (while not making the Common Lisp code in macros more awkward than necessary), to provide as many idiomatic C shorthands as possible (e.g., increments, declaration decorators), to interact with Common Lisp code (honoring lexical scope), to avoid quotation whenever possible and being able to reference symbols from external C files.

The style of meta programming we wanted to support and explore is strictly macro-based. That is, we want the language to be able to specify new syntax and cast semantics into it and not explicitly post-process a syntax tree (as possible, e.g., by working on our AST, as described in Section 3.4, or, for example when using C++ only, via Clang [23]). Note that the latter approach is in fact more powerful, but comes with a much larger overhead in engineering and might consequentially not pay off for projects of small to medium size [17].

3.2 Look and Feel

In the following we will provide a higher-level overview of a few aspects of C-Mera's internal workings, mainly aiming to provide a general outline of the intended look-and-feel we wanted to achieve for our meta language.

Symbols. The following toy example demonstrates the mapping of two addition expressions enclosed in a lexical environment that introduces a local variable in the Common Lisp context.

```
(let ((x  4))
  (set foo (+ 1 2 (cl:+ 1 2) x 'x y)))
```

As can be seen in the following line generated from this example, not all expressions appear in the resulting code, since the Common Lisp expressions are not part of the target language:

```
foo = 1 + 2 + 3 + 4 + x + y;
```

This example is comprised of the following components. The *set* form indicates an assignment for the target language and creates a syntax element that is carried over to the resulting code. Both the plain *+* and *cl:+* (from the *cl* package) denote addition, but the one from the *cl* package is evaluated directly within the host language and yields the number *3*, whereas the unqualified counterpart is kept as an expression of the target language. Common Lisp's special operator *let* defines a lexical scope and introduces the variable *x* in the example above. This operation is solely executed in the scope of the host language and does not contribute additional output to the resulting code. In contrast to *x*, the symbol *y* is undefined, thus it is taken to directly refer to a variable in C. Since there is obviously no useful application of an unquoted, undefined variable in the host language, the assumption that a symbol is designated to be used in the target language (and thus is undefined in the host's context) is justified. Therefore, there is no need to quote undefined symbols inside target language syntax elements. However, it is still required to quote symbols defined in the host language's context, to process them as variables for the target language and avoid value substitution. The implementation of this feature is described in Section 5.3.

Functions and Macros. Functions are managed in a similar fashion. This also holds for special forms to construct the syntax tree of the target language that are, naturally, defined in the host language's context (see Section 3.3). If the first element of a list is not defined during evaluation, it is taken to denote a function call in the target language. In the following example one function and two macros are defined (one function is commented out) and *foo* is assigned the result of calling those, in turn.

```
(defun bar (a b) (cl:+ a b))
; (defun baz (a b) (cl:+ a b))
(defmacro qux (a b) `(+ ,a ,b))
(defmacro qox (a b) `(cl:+ ,a ,b))

(set foo (bar 1 2))
(set foo (baz 1 2))
(set foo (qux 1 2))
(set foo (qox 1 2))
```

Here, the function *bar* returns the number *3*, which is used directly in the resulting code. The function *baz* is not defined (indicated by the line being commented out). Based on the aforementioned processing of unbound symbols, the undefined list head *baz* is treated as a function call in the target language. As can be seen in the generated code below, only one function call (for which there was no valid host-context function available) is generated in the target code:

```
foo = 3;
foo = baz(1, 2);
foo = 1 + 2;
foo = 3;
```

Also note how the expansion of the *qox* macro is evaluated in the host context.

As with symbols used for variable names, there is an ambiguity if the symbol is defined both in the host and target language. In these cases we opted to prefer the host language's version (as with variables), while a function call in the target language can be generated using the *funcall* form. Continuing the example above the following expressions do not trigger host-language function calls or macro invocations:

```
(set foo (funcall 'bar 1 2))    →   foo = bar(1, 2);
(set foo (funcall 'qux 1 2))    →   foo = qux(1, 2);
```

The implementation of this feature is also described in Section 5.3.

3.3 Evaluation

The evaluation scheme we apply in order to build an Abstract Syntax Tree (AST) may be one of the most simple approaches. Nevertheless, for our needs it is more than sufficient and, more importantly, very easy to implement, utilize, and extend. In the following we show how a target-language infix operator (e.g. *+*) can be defined.

```
(defclass infix-node ()
  ((operator :initarg :op)
   (member1  :initarg :lhs)
   (member2  :initarg :rhs)))
```

This specifies an infix expression as being comprised of an operation with a left- and right-hand side. Note that the representation is simplified at this point, a more detailed description of the AST nodes can be found in Section 3.4.

With this example AST, nodes for such expressions can be generated by calls to *make-instance*, building up a tree of node objects. As this would clearly not be very concise code, we wrap macros around each AST-node constructor. For the case of an addition expression this would be `(+ ...)`. The following macro suffices to wrap around the call to *make-instance*:

```
(defmacro + (lhs rhs)
  `(make-instance 'infix-node
                  :op '+
                  :lhs ,lhs
                  :rhs ,rhs))
```

Application of this macro yields the appropriate AST node:

```
(+ 1 2)   →   #<INFIX-NODE #x...>
```

Using this scheme we are not limited to one single level of evaluation, nor constrained to entirely stay in the target language. Nesting multiple target functions and mixing them with host code is supported and intended. The following example shows an evaluation process starting with:

```
(* (/ 1 2) (+ (cl:+ 1 2) 3))
```

Expressions are, as usual, evaluated from the inside (starting with the two leaf nodes of the AST to-be). In this case, one of those expressions generates a node object and the other evaluates as a build-in Common Lisp expression:

```
(* #<INFIX-NODE #x1...> (+ 3 3))
```

The next step is the evaluation of the remaining nested objects:

```
(* #<INFIX-NODE #x1...> #<INFIX-NODE #x2...>)
```

In the final step, the entire expression collapses to one single object:

```
#<INFIX-NODE #x3...>
```

As can be seen, every evaluation results in a node object that is thereafter used in the next evaluation step by its parent.

As described above, the full evaluation process also includes macro expansion that generates the instantiation calls. Thus, the evaluation first expands to the following *make-instance* form:

```
(make-instance 'infix-node
  :op '*
  :lhs (make-instance 'infix-node
         :op '/
         :lhs 1
         :rhs 2)
  :rhs (make-instance 'infix-node
         :op '+
         :lhs (cl:+ 1 2)
         :rhs 3))
```

According to this, the AST is built by expanding all macros and collapsing the individual calls to *make-instance* by evaluation:

```
(make-instance 'infix-node
  :op '+
  :lhs #<INFIX-NODE #x1...>
  :rhs #<INFIX-NODE #x2...>)
```

This shows that the evaluation scheme results in an implicitly self-managed construction of the AST. That is, we rely entirely on the standard Common Lisp reading and evaluation process. Therefore, there is no need for an extra implementation of a parser inside the host language, since every aspect is already handled by the Common Lisp implementation itself.

Note that this evaluation scheme seamlessly integrates with macro processing in general and thus facilitates the incorporation of new, user-defined syntax, even up to the scope of defining custom DSLs [17, 19] without any changes to the underlying AST.

3.4 Abstract Syntax Tree

The AST is the intermediate representation of the fully macro-expanded and evaluated input code. Every node type used for the AST is derived from one common class (*node*). Independent from additional information stored inside individual derived objects, every class instance contains the slots *values* and *subnodes* (inherited from *node*). This very simple setup renders the traversal of the AST almost trivial. Based on Common Lisp's multi-methods, we require only two methods to build various traversers. One of these methods handles the class *node*:

```
(defclass node ()
  ((values   :initarg :values)
   (subnodes :initarg :subnodes)))
```

For derived classes, the slot *values* contains a list of slot names, which are not processed further by the traverser (i.e. leaf nodes). The *subnodes* slot stores a list of slot names that the traverser descends into (i.e. internal nodes). The structure can be used to capture nodes where the sub-nodes have different semantics (e.g. conditional expressions). Nodes storing multiple objects with the same semantics (e.g. body forms) utilize *nodelist*:

```
(defclass nodelist (node)
  ((nodes :initarg :nodes)))
```

The *nodes* slot is a plain list containing the sub-node objects. All nodes in our AST are either a *nodelist* or derived from *node*, and most of the traversal is implemented in terms of them.

AST Traversal. Traversal then works as follows: The traverser starts at the root node and when it encounters an object of type *node* it calls itself recursively for slots of the current object listed in *subnodes*:

```
(defmethod traverser ((trav t) (node node))
  (with-slots (subnodes) node
    (loop for slot-names in subnodes do
      (let ((subnode (slot-value node slot-name)))
        (when subnode
          (traverser trav subnode))))))
```

A similar procedure is executed for *nodelist* nodes.

With the classes defined above and these methods we have a simple traversal scheme that can easily be extended for further tasks. Additionally, more specific traversal methods can implement mechanisms for processing the data of individual node types. As a result, building functionalities that require AST traversal becomes straightforward. The following traverser simply lists all infix expressions in the tree (continuing the example from Section 3.2):

```
(defclass debug-infix ())

(defmethod traverser ((_ debug-infix) (node infix-node))
  (format t "~a~%" (slot-value node 'op)
  (call-next-method)))
```

Note that *call-next-method* continues with the general tree traversal.

Before and After. With additional support from Common Lisp's *before* and *after* methods, the generation of syntactically faithful target code becomes even more comfortable. As an example, utilizing these features enables catching the beginning and end of an expression, which, for example, can easily be exploited for emitting parentheses:

```
(defmethod traverser :before ((pp pretty-printer) (_ infix-node))
  (format (stream pp) "("))

(defmethod traverser :after ((pp pretty-printer) (_ infix-node))
  (format (stream pp) ")")
```

With this approach we can easily ensure proper execution order for arithmetic expressions:

```
(/ (+ 1 3) (+ 2 5))   →   ((1 + 3) / (2 + 5))
```

Proxy-Node Extension. Although we are able to trigger traversal events when entering and leaving a node as described above, we cannot trigger them when descending and returning from specific child nodes while processing their parent node. Visitation of these nodes is implemented by methods on their respective node type, but this loses the context of their parent node (which might be required to generate inter-node output). This situation can easily be solved with *proxy nodes*. Nodes of this type are merely sentinels, without additional content, and are solely applied to identify transitions between nodes. They are usually inserted and removed by a node that needs control over the output between its child nodes.

One example would be placing the +-signs in an arithmetic expression such as `(+ a b)` to yield `a + b`, using the following proxy:

```
(defclass plus-proxy (node)
  ((subnode :initarg :subnode)))
```

Using such an object for the right-hand-side operand of the infix-node would then trigger the proper method with correct placement of the plus sign.

```
(defmethod traverser :before ((pp pretty-printer) (_ plus-proxy))
  (format (stream pp) " + "))
```

Note that this scheme is just an approach to keep traversal mechanics and output logic distinct. When mixing both, proxies will not be required, but then each method on a given node type will have to repeat the traversal logic.

Overall be believe that our AST scheme is very simple and consequently easy to use, while still being easily extensible.

4 PACKAGES

As seen in Section 3.2, we do not want to explicitly annotate symbols with their packages. However, starting from the default package (*cl-user*) the following attempt to define a macro fails:

```
(defmacro + (a b)
  `(make-instance 'infix-node ...))
```

This is due to the inherent *package-lock* on the user package's interned Common Lisp functions and macros. Every symbol interned form *common-lisp* (or short, *cl*) is locked by default. This would prevent any of the redefinitions we have already used many times until now. A possible solution to allow modifications is unlocking packages, which generally is a poor approach, as it is not standardized and drops the overridden symbols' default implementation.

The lock, however, only affects the actual symbols in the *cl* package, not symbols of the same name from different packages. This fact is usually not obvious as virtually all Common Lisp code uses the *cl* package, which then results in name conflicts that cannot be overridden due to the lock. The key to solve this issue is very simple: not using the *cl* package, or, when only few symbols are to be overridden, to not include those when using the *cl* package.

```
(defpackage :meta-lang
  (:use :common-lisp)
  (:shadow :+))
```

This package definition interns all symbols but *+* from the *cl* package. As a result the symbol *+* is unbound and can be used, e.g., for macro definitions:

```
(in-package :meta-lang)

(defmacro + (a b)
  `(cl:+ ,a ,b))
```

The example above defines a simple macro that maps the *+*-sign to its implementation from the *cl* package. As can be seen, access to the original implementation is still possible if the symbol is used with its package prefix. Such a package setup enables us do redefine symbols according to our needs:

```
(defpackage :cm-c
  (:use :common-lisp)
  (:shadow :+))

(in-package :cm-c)

(defmacro + (lhs rhs)
  `(make-instance 'infix-node
                  :op '+
                  :lhs ,lhs
                  :rhs ,rhs))
```

Using this package, superfluous prefixes can be omitted and are only required when accessing (overridden) symbols from *cl*:

```
(+ (cl:+ 1 2) (cl:+ 2 3))     →   3 + 5;
```

Since we also want to reduce the amount of explicitly qualified names in the input code we can utilize a simple *macrolet* to adjust the effects of those symbols in its lexical scope:

```
(defmacro lisp (&body body)
  `(macrolet ((+ (lhs rhs) `(cl:+ ,lhs ,rhs)))
     ,@body))
```

At this point we might end up at an impasse; once inside the lexical scope of the macrolet, globally defined functions redefined in the local scope are not accessible:

```
(+      1 2)      →   1 + 2
(cm-c:+ 1 2)      →   1 + 2
(cl:+   1 2)      →   3

(lisp
  (+      1 2)    →   3
  (cm-c:+ 1 2)    →   3
  (cl:+   1 2))   →   3
```

The obvious solution for this problem is to introduce a third variable that retains the initial functionality, as opposed to the locally used, volatile symbols. To keep the symbols' names, an additional package is required to place those symbols in:

```
(defpackage :swap (:use) (:export :+))
```

Macros plainly wrapping the original symbol or function, unfortunately, fail to provide the required behaviour. Such macros emit symbols that are then still bound in the lexical scope of the surrounding macrolet.

```
(defmacro swap:+ (lhs rhs)
  `(cm-c:+ ,lhs ,rhs))
```

To escape the lexical scope, we can access a symbol's original macro implementation with *macroexpand-1*:

```
(defmacro swap:+ (lhs rhs)
  (macroexpand-1 `(cm-c:+ ,lhs ,rhs)))
```

As long as *macroexpand-1* is called without an environment argument it returns the version of the macro defined in the global environment. With such *swap symbols* we are able to address the global implementation of our syntax, independent from the current lexical scope.

5 BRIDGING THE SYNTACTIC GAP

In this section we describe how certain details of C-Mera are laid out to strike a balance between the worlds of our host and target languages. The first part discussed is control over case. Common Lisp converts symbols that it reads to upper case unless otherwise specified. With this default behaviour, users from C-family languages would be surprised to see how their code changed when printed out in the target language. Section 5.1 details why the built-in modes in Common Lisp do not suffice and describes the compromise employed in our language.

A difficulty of a different kind is Lisp's very uniform notation on the one hand and C's (and even more so its derived languages) extensive syntax on the other hand. Although every aspect of C-family languages can be modeled with S-Expressions, we doubt the benefit of having to formulate every little aspect of C's syntax this way. Arrays can be used as an example here: It is obvious that writing `(array (array (array foo 1) 2) 3)` is not as convenient, at least not as concise, as writing `foo[1][2][3]`. We aim for supporting as much as possible of C's handy syntax by exploiting the extensible Common Lisp reader to parse special syntax. Details on our implementation of such shorthands are presented in Section 5.2.

The last aspect of our presentation is concerned with the input code's aesthetics and appeal. We want to write code as conveniently as possible and prevent exhaustive (and to part of our target audience, confusing) usage of quotes. Instead of writing `(funcall 'f 'a 'b (funcall 'g 1 'c))`, we simply want to allow (and do support) the call to be `(f a b (g 1 c))`, even if *f, a, b, g*, and *c* are unbound. Program code, even if it is target code, should appear as natural code in Lisp notation, and not require quotes when the situation can be uniquely resolved. Our implementation of adaptive quotation that tackles this issue is described in Section 5.3.

5.1 Preserving Case by Inversion

Code in Common Lisp is, unless explicitly avoided, converted to upper case, therefore it often appears that the case of the input code is not considered at all. It can be controlled on a per-symbol level via `(intern "foo")` and explicit literals such as `|foo|`. More general control is available via *readtable-case*, which can change the default behaviour. Even though there is the so called *modern style* for Common Lisp, which sets the *readtable-case* to *:preserve*, current implementations are usually compiled in *:upcase* (causing all *cl* symbols to be interned in upper case). Since our target language does not do any automatic case conversion, but keeps the input code's case as it is, we have been compelled to reproduce this behaviour as closely as possible in C-Mera.

The naïve approach is setting the *readtable-case* to *:preserve* when processing a source files. This is an inadequate solution, however, as it would require us to use upper-case representations of all the standard Common Lisp symbols. As a result, input code would have to be written in the following form:

```
(setf (readtable-case *readtable*) :preserve)

(DEFUN foo (a b) (+ b c))
(DEFUN bar (a b) (CL:+ a b))

(foo 1 (foo X y) (bar 1 2))
```

With *:preserve* we are free to use upper- and lower-case symbols for variables (*X* and *y* in this example), but also forced to write all existing Common Lisp symbols (such as *DEFUN*) in upper case.

Since *:downcase* would not work at all (does not keep case), the only option left to investigate is *:invert*. In fact, with *:invert*, input symbols in lower case are mapped to upper-case symbols (and vice versa). Therefore, input code in lower case can be mapped to existing functions and symbols. One problem remains: newly introduced functions and variables are also inverted. Luckily, Common Lisp processes symbols an additional time during printing, which is a natural part of source-to-source compilers such as we are targeting. Therefore, the desired functionality is available out of the box:

```
(format t "~a" 'foo)   →   FOO
(format t "~a" 'FOO)   →   FOO
(format t "~a" 'Foo)   →   FOO

(setf (readtable-case *readtable*) :invert)
(format t "~a" 'foo)   →   foo
(format t "~a" 'FOO)   →   FOO
(format t "~a" 'Foo)   →   Foo
```

This seems to be a reliable solution, but one detail should be kept in mind: The *intern* function now shows counter-intuitive behavior with inverted reading, since it does not use the reader and therefore is not affected by the *readtable*:

```
(setf (readtable-case *readtable*) :invert)
(format t "~a" (intern "foo"))   →   FOO
(format t "~a" (intern "FOO"))   →   foo
(format t "~a" (intern "Foo"))   →   Foo
```

Interning does not read symbols, but strings, and therefore it misses the initial inversion step. In the given situation, we have implemented and used our own *intern* function that inverts the read string in the same way as the reader does.

5.2 Universal Reader

As exemplified at the outset of of Section 5, forcing the use of S-Expressions for every minor syntactic detail to be generated can become a nuisance. Luckily, Common Lisp's flexible reader can be used to strike a balance between having a macro-processable S-Expression language and supporting idiomatic C-isms.

With `(set-macro-character #\& #\'&-processor)`, for example, the reader can be extended to process symbols starting with an ampersand by applying the function *&-processor* to such symbols. This particular reader function sets up a specific mechanism that covers one single macro character. Therefore, it is usually required to set up functions for individual syntax elements that differ from S-Expressions, but this scheme is limited to elements that can be captured by such a simple prefix.

According to that, it is easy to implement a function that handles C syntax for the address-of operator. In that case the reader simply consumes, e.g., `(+ &a &b)` and emits `(+ (addr-of a) (addr-of b))`. Unfortunately, it is not easily possible to identify C++ references, for example in `(decl ((int& a)))`, since they can occur at the end of the corresponding symbol. In addition to that, we are limited to one single character. Therefore, we are unable to use the reader in that fashion for neither prefix increments nor decrements: `(+ ++a --b)`.

Surprisingly, there is a very simple, general solution for processing almost every type of symbol: hooking the reader macro to whitespace.

```
(set-macro-character #\Space   #'pre-process)
(set-macro-character #\Tab     #'pre-process)
(set-macro-character #\Newline #'pre-process)
```

This setup configures the reader to utilize the function *pre-process* to handle every symbol with a leading whitespace character. The task of *pre-process* is parsing individual symbols, identifying non-Lisp syntax and emitting proper S-Expressions. By doing this, we can support very convenient, but tricky C syntax, as shown in the following examples:

```
(* ++a[4] --b[x++])
→ (* (aref (prefix++ a) 4) (aref (prefix-- b) (postfix++ x)))

(+ foo[baz[1]][2][3] &qox)
→ (+ (aref (aref (aref foo (aref baz 1)) 2) 3) (addr-of qox))

(set foo->bar->baz 5)
→ (set (pref (pref foo bar) baz) 5)
```

So far we are able to process symbols as long as they have leading whitespace. However, list heads usually do not have leading whitespace, but begin directly after the opening parenthesis. These situations should also be managed, for example in calls of type

`(obj->func args...)`. Therefore, we need to register an additional macro character:

```
(set-macro-character #\( #'pre-process-heads)
```

Contrary to the previous symbol processing, where each symbol is handled separately, the macro-character setup above requires us to imitate Common Lisp's standard mode of reading lists, which is easily achieved using `(read-delimited-list #\))` in order to get all list elements. Thereafter, *pre-process-heads* emits a slightly adjusted list comprised of the altered list head and the (untouched) remaining list elements. Eventually, the list's head has been adapted to our needs by *pre-process-heads* and the remaining list elements will be modified later-on by *pre-process* (as described above), if necessary.

Additionally, we might not want to process all list heads in general, but only those that are neither bound variables, nor functions or macros. This is due to the fact that valid Common Lisp macros, for example, can be named in a way that these reader macros would pick up on. In general, we opted to take the meaning defined in the host language for any ambiguous cases. Our approach to identify bound symbols is detailed in the next section.

One conflict that cannot be solved with the aforementioned reader still remains. Packages in Common Lisp are denoted similarly to namespaces in C++, but using the reader for this issue would break Common Lisp's package annotations. As an alternative, fully qualified symbols could be exploited for C++ namespaces:

```
(defpackage :N1)

(set N1::foo 4)
  →   N1::foo = 4;
```

This, however, does not support nesting of namespaces, since *nested packages* are not available in Common Lisp [21]. Naturally, the explicit form can always be utilized, but is very verbose:

```
(set (from-namespace N1 N2 foo) 4)
  →   N1::N2::foo = 4;
```

Therefore we apply *set-dispatch-macro-character* to introduce a specific annotation for C++ namespaces:

```
(set-dispatch-macro-character #\# #\: #'colon-reader)

(set #:N1::N2::var 4)
  →   (set (from-namespace N1 N2 var) 4)
  →   N1::N2::var = 4;
```

As a further convenience for Common Lisp users, our reader macro also supports the single-colon notation: `#:N1:N2:var`.

5.3 Adaptive Quotation

A crucial part of being able to write code as we claim in Section 3.2 is identifying which symbols are bound to host-language interpretations. One example is when it comes to using function calls for the target language in traditional Lisp notation: Instead of being forced to write `(funcall 'foo 1 2 3)` we want to support `(foo 1 2 3)`, even if the symbol *foo* is unbound and have it emit `foo(1, 2, 3)`.

The first attempt to realize the aforementioned notation has been the application of *boundp* and *fboundp*. Both functions work well for globally defined variables, functions, and macros:

```
(defvar foo 1)
(boundp 'foo) ;; -> T
```

However, they cannot be applied to symbols from lexical environments:

```
(let ((bar 1))
  (boundp 'bar)) ;; -> NIL
```

Since the naïve approach cannot handle such symbols, we had to look for an alternative. Every implementation of Common Lisp that supports lexical scoping has to keep track of bound symbols and their meaning. This information is stored in the *environment*, but not every implementation has a convenient method for accessing this data. In case of SBCL [15] and Clozure CL [3] we can exploit *function-information* to check whether a function is defined in the lexical or global scope. Similar to functions, we can use *variable-information* for symbols. For implementations that do not supply these functions, such as, for example, ECL [9], we have to implement a look-up in the environment object itself. The following example shows how one could utilize the listed functions and implement the missing look-up for ECL in order to retrieve information whether a symbols is bound or not.

```
(defun fboundp! (function &optional env)
  #+sbcl (sb-cltl2::function-information function env)
  #+clozure   (ccl::function-information function env)
  #+ecl   (or (fboundp function)
              (find function (rest env)
                :test #'(lambda (x y) (eql x (car y)))))
  #-(or sbcl clozure ecl) (error "..."))

(defun vboundp! (variable &optional env)
  #+sbcl (sb-cltl2::variable-information variable env)
  #+clozure   (ccl::variable-information variable env)
  #+ecl   (or (boundp variable)
              (find variable (first env)
                :test #'(lambda (x y) (eql x (car y)))))
  #-(or sbcl clozure ecl) (error "..."))
```

Due to the fact that these functions require access to the environment object, they can only be applied usefully inside macros. The following macro is a minimal example for a possible use of these functions:

```
(defmacro xboundp (item &environment env)
  (if (or (fboundp! item env)
          (vboundp! item env))
      t     ; item bound
      nil)) ; item unbound
```

With such functions at hand, we are now free to build a more flexible quotation scheme, specifically tailored to our meta language:

```
(defmacro quoty (item &environment env)
  (cond ((listp item)
         (if (fboundp! (first item) env)
             item
             `(function-call ...)))
        ((symbolp item)
         (if (vboundp! item env)
             item
             `',item))
        (t item)))
```

We can now add the *quoty* macro to the tree construction process (see also Section 3.3):

```
(defmacro + (lhs rhs)
  `(make-instance 'infix-node
                  :op '+
                  :lhs (quoty ,lhs)
                  :rhs (quoty ,rhs)))
```

This allows us to freely and seamlessly mix and match globally and lexically bound symbols and functions with unbound symbols taken to denote target-language functions and variables:

```
(labels ((foo (a b) (cl:+ a b)))
  (+ (foo 1 2) (bar 1 2)))

(labels ((foo (a b) (cl:+ a b)))
  (let ((x 5))
    (set A (+ (+ x y) (+ 'x (+ (foo 1 2) (bar 1 2)))))))
```

The arithmetic expression in the last line results in the following code for the target language:

```
A = 5 + y + x + 3 + bar(1, 2);
```

Quoty is most useful in special forms, where we do not want to quote every individual symbol, but still want to be flexible enough to call functions or use symbol values. Another example from C-Mera is that we utilize *quoty* in the variable-declaration macro, *decl*:

```
(decl ((const super_fancy_type x = 4)) ...)
→ const super_fancy_type x = 4; ...

(defmacro with-pointer (pointer &body body)
  `(decl (((postfix* ,pointer) x = (foo)))
     ,@body))

(with-pointer int ...)
→ int* x = foo(); ...
```

The flexible quotation allows us to use types (*super_fancy_type*) and functions (*foo*) that are not defined in the host language's context. Additionally we are now able to evaluate functions inside these *quasi-special* forms (*postfix** and *foo* in the example).

We have opted for this scheme to provide a simple syntax, even in the face of effects similar to *unwanted capture* (by definition of host functions).

6 CONCLUSION

In this paper we have presented many details on how we have constructed our meta language, ranging from Common Lisp implementation techniques to reader-macro hackery. Our pragmatic approach shows with how little effort Common Lisp can be bent toward our ends, resulting in an efficient meta-programming system for C-like languages.

We showed how our simple, Lisp-like notation can be evaluated to provide seamless integration with Common Lisp code during compilation, most notably with support for macros that are our primary vehicle for meta programming (this is also illustrated in our previous work on C-Mera). We also detailed the intricacies of our scheme, namely how to properly override built-in symbols while retaining their original interpretation in an accessible way, how to configure the Common Lisp system to keep our target language's case while not sacrificing a modern notation of the Common Lisp meta part of the language. We furthermore showed how we manage to provide many C-isms that programmers from that area would find awkward working without (and even seasoned Lisp users might miss for their conciseness), and how unnecessary quoting of unbound symbols can be avoided while keeping the Common Lisp interaction fully working.

Overall, none of these aspects are new findings. Our primary goal with this summary paper is to have all of this information collected in a single, clearly marked place. We hope this will help projects with similar demands to get up to speed more easily than when solutions to all of those issues have to found independently and without proper context.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

REFERENCES

[1] Andrei Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley.

[2] Jonathan Carlos Baca. 2016. Lisp/c. https://github.com/eratosthenesia/lispc. (2016). GitHub Repository, Accessed Jan 2017, Active May 2016.

[3] Gary Byers. 2017. Clozure Common Lisp. http://ccl.clozure.com/. (2017). Accessed Jan 2017.

[4] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*. Springer-Verlag New York, Inc., New York, NY, USA, 57–76.

[5] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-stage Language for High-performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 105–116.

[6] Andreas Fredriksson. 2010. Amplifying C. http://voodoo-slide.blogspot.de/-2010/01/amplifying-c.html and https://github.com/deplinenoise/c-amplify. (2010). Personal Blog & Github Report, Accessed Jan 2017, Repository active Feb 2010 – Mar 2010.

[7] de Guzman Joel, Kaiser Hartmut, and Nuffer Dan. 2016. Boost Spirit. http://www.boost.org/doc/libs/1_63_0/libs/spirit/doc/html/index.html. (2016). Accessed Jan 2017.

[8] S. C. Johnson. 1975. *YACC—yet another compiler-compiler.* Technical Report CS-32. AT&T Bell Laboratories, Murray Hill, N.J.

[9] Daniel Kochmański. 2017. Embeddable Common Lisp. https://common-lisp.net/project/ecl/main.html. (2017). Accessed Jan 2017.

[10] Jan Cornelis Willem Kroeze. 2010. *Tracing rays the past, present and future of ray tracing performance.* Ph.D. Dissertation. North-West University.

[11] M. E. Lesk and E. Schmidt. *Lex — A Lexical Analyzer Generator.* Technical Report. AT&T Bell Laboratories, Murray Hill, New Jersey 07974. PS1:16–1 – PS1:16–12 pages. http://kjellggu.myocard.net/misc/tutorials/lex.pdf

[12] Alexander Lier, Franke Linus, Marc Stamminger, and Kai Selgrad. 2016. A Case Study in Implementation-Space Exploration. In *Proceedings of ELS 2016 9th European Lisp Symposium.* 83–90.

[13] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 103–104.

[14] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[15] Christophe Rhodes. 2008. Self-Sustaining Systems. Springer-Verlag, Berlin, Heidelberg, Chapter SBCL: A Sanely-Bootstrappable Common Lisp, 74–86. DOI: http://dx.doi.org/10.1007/978-3-540-89275-5_5

[16] Vladimir Sedach. 2016. Parenscript. http://common-lisp.net/project/parenscript/. (2016). Accessed Jan 2017.

[17] Kai Selgrad, Alexander Lier, Jan Dörntlein, Oliver Reiche, and Marc Stamminger. 2016. A High-Performance Image Processing DSL for Heterogeneous Architectures. In *Proceedings of ELS 2016 9th European Lisp Symposium.* 39–46.

[18] Kai Selgrad, Alexander Lier, Franz Köferl, Marc Stamminger, and Daniel Lohmann. 2015. Lightweight, Generative Variant Exploration for High-Performance Graphics Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. ACM, New York, NY, USA, 141–150. DOI: http://dx.doi.org/10.1145/2814204.2814220

[19] Kai Selgrad, Alexander Lier, Markus Wittmann, Daniel Lohmann, and Marc Stamminger. 2014. Defmacro for C: Lightweight, Ad Hoc Code Generation. In *Proceedings of ELS 2014 7th European Lisp Symposium.* 80–87.

[20] Philipp Slusallek. 2015. Approaches for Real-Time Ray Tracing and Lighting Simulation. http://www.dreamspaceproject.eu/dyn/1429609964713/DREAMSPACE_-D4.1.1_Approaches_v1.3.pdf. (Jan. 2015).

[21] Alessio Stalla. 2017. Symbols as Namespaces. *ELS 2016 9th European Lisp Symposium*, Lightning Talks Session 1, https://www.european-lisp-symposium.org/editions/2016/lightning-talks-1.pdf. (May 2017).

[22] Masayuki Takagi. 2017. Cl-Cuda. https://github.com/takagi/cl-cuda. (2017). GitHub Repository, Accessed Jan 2017, Active Apr 2012 – Jan 2017.

[23] The Clang Developers. 2014. Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org. (2014).

[24] David Vandevoorde and Nicolai M. Josuttis. 2002. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[25] Todd Veldhuizen. 1995. Template Metaprograms. *C++ Report* (May 1995).