A modern implementation of the LOOP macro

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux1.fr

ABSTRACT

Most Common Lisp implementations seem to use a derivative of MIT `loop`. This implementation predates the Common Lisp standard, which means that it does not use some of the features of Common Lisp that were not part of the language before 1994. As a consequence, the `loop` implementation in all major Common Lisp implementation is *monolithic* and therefore hard to maintain and extend.

Furthermore, MIT `loop` is not a conforming `loop` implementation, in that it produces the wrong result for certain inputs. In addition, MIT `loop` accepts sequences of `loop` clauses with undefined behavior according to the standard, though whether such extended behavior is a problem is debatable.

We describe a modern implementation of the Common Lisp `loop` macro. This implementation is part of the SICL project. To make this implementation of the macro modular, maintainable, and extensible, we use *combinator parsing* to recognize `loop` clauses, and we use CLOS generic functions for code generation.

CCS Concepts

•Software and its engineering → Control structures;

Keywords

CLOS, Common Lisp, Iteration, Combinator parsing

1. INTRODUCTION

The `loop` macro is part of the Common Lisp standard, so every conforming Common Lisp implementation contains an implementation of this macro.

This macro is frequently criticized as un-Lispy since it does not use S-expressions for its clauses, and for being impossible to extend, at least by using only features available in the Common Lisp standard. In addition, advocates of purely-functional programming also criticize it, along with all other iteration constructs that can not be explained in terms of recursion.

Despite all this criticism, the `loop` macro is an essential and widely used part of any non-trivial Common Lisp program. It is able to satisfy the vast majority of iteration needs. In addition, it is far easier to understand than equivalent loops using other iteration constructs such as `dotimes`, `dolist`, and `do`.

Most current implementations of Common Lisp seem to use an implementation of the `loop` macro that was largely written before the Common Lisp standard was adopted. Consequently, some of the interesting features of the standardized Common Lisp language are not used in the implementation of the `loop` macro in these implementations. In particular, the use of *generic functions* is typically minimal. As a result, the implementation of this macro is quite *monolithic*, making it hard to maintain, whether in order to remove defects or to extend it.

We present a modern implementation of the `loop` macro. This implementation was written as part of the SICL[1] project, of which one of the explicit goals is to use improved coding techniques.

We are able to obtain a more modular `loop` implementation by using two key techniques. The first one is to parse the clauses using a parsing technique that allows for individual clause parsers to be *textually separated* according to clause type. The second modularity technique is to use *generic functions* for semantic analysis and code generation. By defining clause types as standard classes, we are able to textually separate processing according to clause type through the use of methods specialized to these clause classes.

2. PREVIOUS WORK

2.1 MIT `loop` with variations

One of the first implementations of the Common Lisp `loop` macro is the one that is often referred to as "MIT `loop`" [1]. A popular variation of this implementation includes modifications by Symbolics Inc.

This implementation of the `loop` macro is sometimes more permissive than the Common Lisp standard. For example, the standard requires all *variable clauses* to precede all *main clauses*. Code such as the one in this example:

```
(loop until (> i 20)
      for i from 0
      do (print i))
```

[1]See https://github.com/robert-strandh/SICL

is thus not conforming according to the standard, since `until` is a *main clause* whereas `for` is a *variable clause*.

However, MIT `loop` and its variation accepts the code in the example.

Another example of non-conforming behavior is illustrated by the following code:

```
(loop for i from 0 below 10
      sum i
      finally (print i))
```

The Common Lisp standard clearly states that the loop variable does not take on the value of the upper limit, here 10, so the value printed in the `finally` clause should be 9. However, `loop` implementations derived from MIT `loop` print 10 instead.

Notice that the two examples above are non-conforming in two different ways, as explained in section 1.5 of the Common Lisp standard.

In the first case, we have an example of *a non-conforming program* as explained in section 1.5.2 for the simple reason that the standard does not specify what an implementation must do when the clause order is violated. By default, then, the behavior is said to be *undefined*, meaning that the implementation is free to reject the non-conforming program or to accept it and interpret it in some (perhaps unexpected) way. The MIT `loop` implementation is therefore conforming in this respect.

In the second case, we have an example of *a non-conforming implementation* as explained in section 1.5.1. The reason is that the standard clearly stipulates that every implementation must print 9, whereas MIT `loop` prints 10.

The MIT `loop` implementation is *monolithic* and that holds true for its variations too. The code is contained in a single file with around 2000 lines of code in it.

Code generation uses a significant number of special variables holding various pieces of information that are ultimately assembled into the final expansion of the macro.

2.2 ECL and Clasp

ECL[2] includes two implementations of the `loop` macro, namely the initial MIT `loop` with only minor modifications, and the variation by Symbolics Inc also with minor modifications.

Clasp[3] is a recent implementation of Common Lisp. It is derived from ECL in that the C code of ECL has been translated to C++ whereas most of the Common Lisp code has been included with no modification, including the code for the `loop` macro.

ECL `loop` being derived from MIT `loop`, the non-conforming example shown in Section 2.1 is also accepted by ECL and Clasp.

2.3 SBCL

SBCL[4] includes an implementation of the `loop` macro that was originally derived from MIT `loop`, but that also includes code from the `loop` macro of the Genera operating system. Furthermore, the SBCL implementation of the `loop` macro has been modified and improved to allow for user-definable extensions, such as the extension defined in [3] for iterating over user-definable sequences.

SBCL `loop` being derived from MIT `loop`, the non-conforming example shown in Section 2.1 is also accepted by SBCL.

2.4 CLISP

CLISP has its own implementation of the `loop` macro. The bulk of the implementation can be found in a function named `expand-loop`. This function consists of more than 900 lines of code.

2.5 CCL

Like many other implementations, CCL[5] includes the variation of MIT `loop` containing modifications by Symbolics Inc.

2.6 LispWorks

Evaluating the two examples in Section 2.1 on LispWorks[6] gives the same result as the implementations using MIT `loop`, suggesting that LispWorks also uses a derivative of that `loop` implementation.

3. OUR TECHNIQUE

3.1 Parsing clauses

In order to parse `loop` clauses, we use a simplified version[7] of a parsing technique known as *combinator parsing* [4].

With this parsing technique, client code defines *elementary parsers* that are then combined using combinators such as *alternative* and *sequence*. The resulting parser code is *modular* in that individual parsers do not have to be listed in one single place. For the `loop` clauses, this modularity means that each type of clause can be defined in a different module.

In our parsing framework, an individual parser is an ordinary Common Lisp function that takes a list of Common Lisp expressions and that returns three values:

1. A generalized Boolean indicating whether the parse succeeded.
2. The result of the parse. If the parse does not succeed, then this value is unspecified.
3. A list of the tokens that remain after the parse. If the parse does not succeed, then this list contains the original list of tokens passed as an argument.

Consider the following example:

```
(define-parser arithmetic-up-1-parser
  (consecutive
   (lambda (var type-spec from to by)
     (make-instance 'for-as-arithmetic-up
       :order '(from to by)
       :var-spec var
       :type-spec type-spec
       :start-form from
       :end-form (cdr to)
```

[2]ECL stands for "Embedded Common Lisp.
See: //https://gitlab.com/embeddable-common-lisp/ecl
[3]See: https://github.com/drmeister/clasp
[4]SBCL stands for Steel-Bank Common Lisp.
See: http://www.sbcl.org/
[5]CCL stands for Clozure Common Lisp.
See: http://ccl.clozure.com/
[6]See: http://www.lispworks.com/
[7]It is simplified because we do not need the full backtracking power of combinator parsing.

```
    :by-form by
    :termination-test (car to)))
'simple-var-parser
'optional-type-spec-parser
(alternative 'from-parser
             'upfrom-parser)
(alternative 'to-parser
             'upto-parser
             'below-parser)
'by-parser))
```

The macro `define-parser` defines a named parser. This parser consists of four consecutive parsers:

1. A parser that recognizes a simple variable. The result of this parser is the variable.
2. A parser that recognizes an optional type specifier. The result of this parser is the type specifier or `t` if the type specifier is absent.
3. A parser that recognizes one of the `loop` keywords `from` or `upfrom` followed by a form. The result of the parser is the form.
4. A parser that recognizes one of the `loop` keywords `to`, `upto`, or `below` followed by a form. The result of this parser is a `cons`, where the `car` is either the symbol < or the symbol <= depending on which keyword was recognized, and the `cdr` is the form.

The function defined by the `lambda` expression combines the results of those four parsers into a single result for the newly defined parser. In this example, the result of the new parser is an instance of the class `for-as-arithmetic-up`.

Initially, the `loop` body is parsed as a sequence of individual `loop` clauses, without any consideration for the order between those clauses. A failure to parse during this phase will manifest itself as an error relating to a particular clause, whether it is in a valid position or not. Furthermore, ignoring restrictions on clause ordering allows us to check the syntax of each clause. If order had been taken into account, we would either have to abandon the parsing phase when a syntactically correct clause were found in the wrong position and thereby being unable to verify subsequent clauses, or else we would have to implement some sophisticated error recovery, allowing the parsing process to continue after a failure.

3.2 Representing parsed clauses

The result of the initial parsing process is a list of clauses, where each clause has been turned into a standard instance.[8]

The classes representing different clause are organized into a graph that that mostly mirrors the names and descriptions of different clause types defined by the Common Lisp standard.

So for example, the class named `main-clause` is the root class of all clauses of that type mentioned in the standard. The same is true for `variable-clause`, `name-clause`, etc.

Classes representing clauses that admit the `loop` keyword `and` also have a list of sub-clauses.

[8]We avoid using language such as "an instance of a CLOS class" since all classes are CLOS classes and therefore all Common Lisp objects are instances of CLOS classes. A "standard instance" representing a `loop` clause is created by calling `make-instance` on a class defined by `defclass`.

This organization allows us to capture commonalities between different clause types by defining methods on generic functions that are specialized to the appropriate class in this graph.

In addition to representing each clause as a standard instance, we also represent the `loop` body itself as an instance of the class named `loop-body`. This instance contains a list of all the clauses, but also other information, in particular about default accumulation for this call to the `loop` macro.

3.3 Semantic analysis

We use generic functions to analyze the contents of the parsed clauses, and to generate code from them. The reason for using generic functions is again one of modularity. A method specialized to a particular clause type, represented by a particular standard class, can be textually close to other code related that clause type.

Checking the validity of the order between clauses is done in the first step of the *semantic analysis*, allowing us to signal pertinent error conditions if the restrictions concerning the order of clauses are not respected.

Next, we verify that the variables introduced by a clause are unique when it would not make sense to have multiple occurrences of the same variable. We also verify that there is at most one *default accumulation category*, i.e, one of the categories *list*, *min/max*, and *count/sum*.

3.4 Code generation

The main control structure for code generation consists of two steps:

- First, the `loop` prologue, the `loop` body, and the `loop` epilogue are constructed in the form of a `tagbody` special form.
- To the resulting `tagbody` form is then applied a set of nested *wrappers*, one for each clause. A wrapper for a clause typically contains `let` *bindings* required for the clause, but also iterator forms where such iterators are required by the clause type, for example `with-package-iterator`.

The `loop` body consists of three consecutive parts:

1. The *main* body, containing code for the `do` clause and the accumulation clauses.
2. The *termination-test* part, containing code that checks whether iteration should terminate.
3. The *stepping* part, containing code that updates iteration variables in preparation for the next iteration.

The essence of code generation is handled by a number of generic functions, each extracting different information from a clause:

- `accumulation-variables` extracts the accumulation variables of a clause, indicating also whether the `loop` keyword `into` is present.
- `declarations` extracts any declarations that result from the clause.
- `prologue-form` returns a form that should go in the `loop` prologue, or `nil` if no prologue form is required for the clause.

- `epilogue-form` returns a form that should go in the `loop` epilogue, or `nil` if no epilogue form is required for the clause.

- `termination-form` returns a form that should become a termination test, or `nil` if the clause does not result in a termination test.

- `step-form` returns a form that should be included in the stepping part of the `loop` body, for those clause types that define stepping. This generic function returns `nil` if the clause does not have any step forms associated with it.

- `body-form` returns a form that should be present in the main body of the expansion, or `nil` if the clause does not result in any form for the body.

The generic function `prologue-form` takes a clause argument and returns a form that should go in the `loop` prologue. The `initially` clause is an obvious candidate for such code. But the stepping clauses also have code that goes in the prologue, namely an initial termination test to determine whether any iterations at all should be executed.

Of the clause types defined by the Common Lisp standard, only the `finally` clause has a method that returns a value other than `nil` on the generic function `epilogue-form`.

The generic function `termination-form` takes a clause argument and returns a form for that clause that should go in the termination-test part of the body of the expanded code. Some of the `for/as` clauses and also the `repeat` clause have specialized methods on this generic function.

The generic function `step-form` returns takes a clause argument and returns a form for that clause that should go in the stepping part of the body of the expanded code. The `for/as` clauses and also the `repeat` clause have specialized methods on this generic function.

The generic function `body-form` returns takes a clause argument and returns a form for that clause that should go in the main body of the expanded code. The `do` and the accumulation clauses have specialized methods on this generic function.

3.5 Tests

Our code has been thoroughly tested. The code for testing contains almost 5000 lines. This code has been taken from the Paul Dietz' ANSI test suite[9] and adapted to our needs. In particular, we had to remove some tests that did not conform to the standard, and we added tests where the test suite omitted to test potentially non-conforming behavior.

4. BENEFITS OF OUR METHOD

As already mentioned in Section 3.1, the main advantage of our technique is that it allows for a *modular* structure of the `loop` implementation.

The most immediate consequence of this improved modularity is that the code is easier to maintain than a monolithic code for the same purpose. There is less code that a maintainer needs to understand for a given modification, and a modification in one module is less likely to break other modules.

[9]See: https://gitlab.common-lisp.net/groups/ansi-test

This modularity also makes it very simple for additional clause types to be added by the Common Lisp implementation, such as the extension for iterating over the user-extensible sequences described by Rhodes in his paper on user-extensible sequences [3]. This extension defines the new `loop` keywords `element` and `elements` for this purpose.

5. CONCLUSIONS AND FUTURE WORK

We have described a modern implementation of the Common Lisp `loop` macro. The main benefit of our method is better *modularity* compared to existing implementations, which makes maintenance easier, and also allows for more modular integration of client-defined extensions.

Our implementation contains significantly more code than, for instance, MIT `loop`; more than 5000 lines compared to 2000. There are several explanations for this discrepancy:

- Our code has more lines of comments; nearly 1500 compared to less than 200 for MIT `loop`.

- Our implementation is divided into nearly 50 files, or *modules*, and each new file represents some overhead in terms of code size,

- Our implementation contains more semantic verification as shown by the fact that it rejects the examples of non-conforming code shown in Section 2.1.

- Commonalities between clause types are captured as explicit class definitions which require additional code.

- We most likely have not identified all the instances where refactoring the code would be beneficial.

5.1 Use external parser framework

When we started the work on this library, we were unaware of any existing libraries for combinator parsing written in Common Lisp. Since then, we have been made aware of several libraries with such functionality, in particular:

- "cl-parser-combinators"[10] which is a library for combinator parsing inspired by Parsec [2]. Parsec was originally written in Haskell, and later re-implemented in other languages as well.

- "SMUG"[11] which seems to be more self contained than cl-parser-combinators, especially when it comes to the documentation.

We plan to evaluate cl-parser-combinators and SMUG to determine whether they provide the functionality required for parsing `loop` clauses, and if not, whether any of them can extended to obtain this functionality.

A significant advantage of using one of these libraries over the existing technique is that they both have full support for the most general backtracking capabilities of combinator parsing. Using one of them rather than our current technique would make it unnecessary to consider careful ordering of clause parsers the way we currently need to do.

A possible disadvantage might be that full backtracking is potentially costly in terms of performance. However, we do not expect performance of clause parsers to be a determining factor for the overall performance of a Common Lisp compiler.

[10]https://github.com/Ramarren/cl-parser-combinators
[11]https://github.com/drewc/smug

5.2 Second clause parser

As mentioned in Section 3, we are able to signal appropriate conditions in some cases when the initial attempt is made to parse the body of the `loop` form as individual clauses. However, when a syntax error is detected in some clause, all further analysis is abandoned. It would clearly be better if the analysis could continue with the remaining clauses, and if an appropriate error condition could be signaled for the faulty clause.

A simple way of improving error reporting would be to add more parsers for each clause type. This additional parsers would recognize incorrect clause syntax and ultimately result in an error being signaled, but more importantly, they would succeed so that parsing could continue with subsequent clauses.

Unfortunately, however, while the parsing technique we use has many advantages as described in refSecsec-benefits, it also has the main disadvantage that parsing gets slower as more parsers need to be tried, in particular if no care is taken to order the parsers with respect to probability of success.

We plan to avoid this conundrum by implementing a *second parser* for parsing individual clauses. This second parser would be invoked only when the first one fails. In that situation, we estimate that performance is of secondary importance and that emphasis should be on appropriate error signaling.

5.3 Code refactoring

As suggested in the beginning of this section, there are very likely several remaining opportunities for code refactoring. Part of the plan for future work is to identify such opportunities and restructure the code accordingly, while respecting the existing modular structure of the code.

6. ACKNOWLEDGMENTS

We would like to thank David Murray for providing valuable feedback on early versions of this paper.

7. REFERENCES

[1] G. Burke and D. Moon. Loop iteration macro. Technical Report MIT/LCS/TM-169, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 1980.

[2] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, University of Utrecht, 2001.

[3] C. Rhodes. User-extensible sequences in common lisp. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM.

[4] P. Wadler. How to replace failure by a list of successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.