Pattern-Based S-Expression Rewriting in Emacs

Ryan Culpepper
Czech Technical University in Prague
ryanc@racket-lang.org

ABSTRACT

`sexp-rewrite` is an Emacs library for doing pattern-based rewriting of S-expression-structured code—ie, code in Lisp, Scheme, and Racket. The library provides a simple but powerful pattern language that enables users to define rewriting rules (called "tactics") and auxiliary nonterminals.

CCS CONCEPTS

• **Software and its engineering** → *Domain specific languages*; *Development frameworks and environments*;

ACM Reference Format:
Ryan Culpepper. 2019. Pattern-Based S-Expression Rewriting in Emacs. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.5281/zenodo.2642093

1 INTRODUCTION

`sexp-rewrite` is an Emacs library that allows users to apply rewriting rules for S-expression structured program text. It also allows users to define their own rewriting rules using a simple but expressive pattern language based on the `syntax-parse` notation for macros (Culpepper 2012).

This section introduces `sexp-rewrite` by showing examples of rewriting tactics included with `sexp-rewrite` for the Racket programming language.

One example is the conversion of nested trees of `if` expressions to `cond` expressions. Here is one transformation:

```
(define-sexprw-tactic if-to-cond
  (if $test $then $else)
  (cond [$test $then] !NL
        [else $else]))
```

Another tactic rewrites `let` followed by an immediate `if` test:

```
(define-sexprw-tactic let-if-to-cond
  (let ([$name:id $rhs])
    (if $name:id $then $else))
  (cond [$rhs !SL => (lambda ($name) !SL $then)] !NL
        [else !SL $else]))
```

These tactics, along with a few others not shown here, make it possible to transform a tree of `if`s and `let`s into a `cond` expression with a single command (Figure 1).

The Quack (Van Dyke 2002) major mode for Scheme and Racket has a function for toggling the function definition at the cursor between implicit and explicit `lambda` notation. For example:

```
(define (add1 n)          (define add1
  (+ 1 n))          ⇔       (lambda (n) (+ 1 n)))
```

The main elisp function implementing this transformation, `quack-toggle-lambda`, is 74 lines, not including helper functions and regular expressions defined elsewhere in `quack.el`.

Using `sexp-rewrite`, the editor transformations can be expressed with two rewriting "tactics" of three lines each:[1]

```
(define-sexprw-tactic define-absorb-lambda
  (define $name:id (lambda ($arg ...) $body:rest))
  (define ($name $arg ...) !NL $body))
(define-sexprw-tactic define-split-lambda
  (define ($name:id $arg ...) $body:rest)
  (define $name !NL (lambda ($arg ...) !SL $body)))
```

Another example comes from SXML (Kiselyov 2004), which uses a particular idiom for optional arguments: the formals include a rest argument, then the procedure body starts with a `let` expression that binds the optional argument to the first element of the rest argument, if there is one, or a default value otherwise. In Racket it is more idomatic (and efficient) to use the language's built-in support for optional arguments. For example:

```
(define (ddo:ancestor test-pred? . nums)
  (let ((num (if (null? nums) 0 (car nums))))
    (do-stuff-with test-pred? num)))
⇒
(define (ddo:ancestor test-pred? [num 0])
  (do-stuff-with test-pred? num))
```

A tactic for translating the former to the latter is shown in Figure 2. The tactic's guard handles a case where a default value of `null` is written as the rest argument (known to be `null` on that branch).

2 TACTICS AND NONTERMINALS

A tactic definition consists of a pattern, template, and optional `:with` or `:guard` clauses:

```
(define-sexprw-tactic Name
  Pattern WithOrGuard ... Template)

WithOrGuard = :with Pattern Template
            | :guard GuardFunction
```

A `:with` clause performs an additional pattern match on the text produced by an intermediate template. A guard procedure takes an environment (an association list mapping attribute names to matches) and returns either a singleton list with an environment (possibly extended or modified) to accept or `nil` to reject.

[1] With this ~1500 line supporting library.

```
(if (not k)                                    (cond [(not k) (error)]
    (error)                          ⇔               [(assq k env) => (lambda (x) (cdr x))]
    (let ([x (assq k env)])                          [else (error)])
      (if x (cdr x) (error))))
```

Fig. 1. Conversion of `if` and `let` to `cond`

```
(define-sexprw-tactic define-rest-to-optional
  (define ($name:id $arg:id ... . $rest:id)
    (let (($optional-arg:id (if (null? $rest:id) $default (car $rest:id))))
      $body:rest))
  :guard (lambda (env)
           ; If $default = $rest, rewrite to null; unsafe if refs to $rest remain.
           (if (sexprw-entry-equal (sexprw-env-ref env '$default) (sexprw-env-ref env '$rest))
               (list (cons (cons '$default (sexprw-template 'null env)) env))
               (list env)))
  (define ($name $arg ... [$optional-arg $default]) !NL $body))
```

Fig. 2. A tactic for optional arguments

A nonterminal definition consists of one or more patterns:

```
(define-sexprw-nt Name
  MaybeAttrs
  (pattern Pattern WithOrGuard ...) ...)

MaybeAttrs = ε
           | :attributes (ATTR-NAME ...)
```

Pattern variables defined inside of a nonterminals patterns are available as attributes of instances of the nonterminal.

A tactic name can also be used as a nonterminal name. In addition to the tactic's pattern variables, it also exports an attribute named `$out` with the result of the template. This makes it simple to compose tactics.

The following built-in nonterminals are provided:

- `id` matches any atom (currently includes numbers, etc, too).
- `pure-sexp` matches a single sexp.
- `sexp` matches a single sexp, which may have comments preceding it.
- `rest` matches the rest of the enclosing sexp, including comments and terms; useful for function bodies, for example.

3 PATTERNS AND TEMPLATES

Rewriting tactics use *patterns* to match regions of text to which the rewriting applies. Patterns bind *pattern variables* to subregions of text and *templates* use pattern variables, literal text, and formatting instructions to form the replacement text.

A pattern is one of the following:

- `symbol` matches that literal symbol. The symbol must not start with a `$` character.
- `$name` matches any S-expression and binds it to the pattern variable `$name`.
- `$name:nt` matches an occurrence of the nonterminal `nt` and binds it to the pattern variable `$name`.
- `(pattern1 ··· patternN)` (that is, a list of patterns—the ··· are not literal) matches a parenthesized sequence of N terms where each term matches the corresponding pattern.
- `(!SPLICE pattern1 ··· patternN)` matches a sequence of N terms (non-parenthesized) where each term matches the corresponding pattern.
- `pattern ...` (that is, a pattern followed by literal ellipses) matches zero or more occurrences of `pattern`. The variables within `pattern` are bound to a sequence of matches, and they must be used under ellipses in the corresponding template.
- `(!OR pattern1 ··· patternN)` matches if any of the given patterns match.
- `(!AND pattern1 ··· patternN)` matches if all of the given patterns match.

The same forms are allowed for templates, except `!OR` and `!AND` are not allowed, and pattern variables are written as `$name` instead of `$name:nt`. The following additional template forms are supported:

- `$name.$attr` produces the text bound to the attribute `$attr` of the pattern variable `$name`, which must be bound to a nonterminal the defines `$attr`.
- `(!SQ template1 ··· templateN)` encloses the results of the N templates within square brackets.
- `!NL` prefers a new line before the next non-empty template.
- `!SL` prefers a new line before the next template only if its contents span multiple lines.

The square-bracket and spacing instructions are helpful for producing idiomatically formatted output.

4 AVAILABILITY

`sexp-rewrite` is available at
https://github.com/rmculpepper/sexp-rewrite/

BIBLIOGRAPHY

Ryan Culpepper. Fortifying macros. *Journal of Functional Programming* 4-5(22), pp. 224–243, 2012.

Oleg Kiselyov. SXML. http://okmij.org/ftp/Scheme/SXML.html, 2004.

Neil Van Dyke. Quack. https://www.neilvandyke.org/quack/, 2002.