The CL-Yacc Manual

Juliusz Chroboczek

CL-Yacc is a LALR(1) parser generator for Common Lisp, somewhat like Yacc, GNU Bison, Zebu, lalr.cl or lalr.scm.

Table of Contents

1 A complete example

CL-Yacc exports its symbols from the package `yacc`:

```
(use-package '#:yacc)
```

A parser consumes the output of a lexer, that produces a stream of terminals. CL-Yacc expects the lexer to be a function of no arguments (a *thunk*) that returns two values: the next terminal symbol, and the value of the symbol, which will be passed to the action associated with a production. At the end of the input, the lexer should return `nil`.

A very simple lexer that grabs tokens from a list:

```
(defun list-lexer (list)
  #'(lambda ()
      (let ((value (pop list)))
        (if (null value)
            (values nil nil)
            (let ((terminal
                   (cond ((member value '(+ - * / |(| |)|)) value)
                         ((integerp value) 'int)
                         ((symbolp value) 'id)
                         (t (error "Unexpected value ~S" value)))))
              (values terminal value))))))
```

We will implement the following grammar:

```
expression ::= expression + expression
expression ::= expression - expression
expression ::= expression * expression
expression ::= expression / expression
expression ::= term

term ::= id
term ::= int
term ::= - term
term ::= ( expression )
```

As this grammar is ambiguous, we need to specify the precedence and associativity of the operators. The operators `*` and `/` will have the highest precedence, `+` and `-` will have a lower one. All operators will be left-associative.

If no semantic action is specified, CL-Yacc provides default actions which are either `#'list` or `#'identity`, depending on how a production is written. For building a Lisp-like parse tree with this grammar, we will need two additional actions:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defun i2p (a b c)
    "Infix to prefix"
    (list b a c))

  (defun k-2-3 (a b c)
    "Second out of three"
    (declare (ignore a c))
    b)
)
```

The parser definition itself:

```
(define-parser *expression-parser*
```

```
(:start-symbol expression)
(:terminals (int id + - * / |(| |)|))
(:precedence ((:left * /) (:left + -)))

(expression
 (expression + expression #'i2p)
 (expression - expression #'i2p)
 (expression * expression #'i2p)
 (expression / expression #'i2p)
 term)

(term
 id
 int
 (- term)
 (|(| expression |)| #'k-2-3)))
```

After loading this code, the parser is the value of the special variable `*expression-parser*`, which can be passed to `parse-with-lexer`:

```
(parse-with-lexer (list-lexer '(x * - - 2 + 3 * y)) *expression-parser*)
    ⇒ (+ (* X (- (- 2))) (* 3 Y))
```

2 Reference

2.1 Running the parser

The main entry point to the parser is `parse-with-lexer`.

`parse-with-lexer` *lexer parser* [Function]

Parse the input provided by the lexer *lexer* using the parser *parser*.

The value of *lexer* should be a function of no arguments that returns two values: the terminal symbol corresponding to the next token (a non-null symbol), and its value (anything that the associated actions can take as argument). It should return `(values nil nil)` when the end of the input is reached.

The value of *parser* should be a `parser` structure, as computed by `make-parser` and `define-parser`.

2.2 Macro interface

`define-grammar` *name option... production...* [Macro]

option ::= `(` *keyword value* `)`
production ::= `(` *symbol rhs...* `)`
rhs ::= *symbol*
rhs ::= `(` *symbol...* [*action*] `)`

Generates a grammar and binds it to the special variable *name*. This has the side effect of globally proclaiming *name* special.

Every production is a list of a non-terminal symbol and one or more right hand sides. Every right hand side is either a symbol, or a list of symbols optionally followed with an action.

The action should be a non-atomic form that evaluates to a function in a null lexical environment. If omitted, it defaults to `#'identity` in the first form of *rhs*, and to `#'list` in the second form.

The legal options are:

`:start-symbol`
: Defines the starting symbol of the grammar. This is required.

`:terminals`
: Defines the list of terminals of the grammar. This is required.

`:precedence`
: The value of this option should be a list of items of the form (*associativity* `.` *terminals*), where *associativity* is one of `:left`, `:right` or `:nonassoc`, and *terminals* is a list of terminal symbols. *Associativity* specifies the associativity of the terminals, and earlier items will give their elements a precedence higher than that of later ones.

`define-parser` *name option... production...* [Macro]

Generates a parser and binds it to the special variable *name*. This has the side effect of globally proclaiming *name* special.

The syntax is the same as that of `define-grammar`, except that the following additional options are allowed:

`:muffle-conflicts`
: If `nil` (the default), a warning is signalled for every conflict. If the symbol `:some`, then only a summary of the number of conflicts is signalled. If `T`, then no

warnings at all are signalled for conflicts. Otherwise, its value should be a list of two integers (*sr rr*), in which case a summary warning will be signalled unless exactly *sr* shift-reduce and *rr* reduce-reduce conflicts were found.

`:print-derives-epsilon`
: If true, print the list of nonterminal symbols that derive the empty string.

`:print-first-terminals`
: If true, print, for every nonterminal symbol, the list of terminals that it may start with.

`:print-states`
: If true, print the computed kernels of LR(0) items.

`:print-goto-graph`
: If true, print the computed goto graph.

`:print-lookaheads`
: If true, print the computed kernels of LR(0) items together with their lookaheads.

2.3 Functional interface

The macros `define-parser` and `define-grammar` expand into calls to `defparameter`, `make-parser`, `make-grammar` and `make-production` with suitable `make-load-form` magic to ensure that the time consuming parser generation happens at compile time rather than at load time. The underlying functions are exported in case you want to design a different syntax for grammars, or generate grammars automatically.

`make-production` *symbol derives* `&key` *action action-form* [Function]
Returns a production for non-terminal *symbol* with right-hand-side *derives* (a list of symbols). *Action* is the associated action, and should be a function; it defaults to `#'list`. *Action-form* should be a form that evaluates to *action* in a null lexical environment; if null (the default), the production (and hence any grammar or parser that uses it) will not be fasdumpable.

`make-grammar &key` *name start-symbol terminals precedence productions* [Function]
Returns a grammar. *Name* is the name of the grammar (gratuitious documentation). *Start-symbol*, *terminals* and *precedence* are as in `define-grammar`. *Productions* is a list of productions.

`make-parser` *grammar* `&key` *discard-memos muffle-conflicts print-derives-epsilon print-first-terminals print-states print-goto-graph print-lookaheads* [Function]
Computes and returns a parser for grammar *grammar*. *discard-memos* specifies whether temporary data associated with the grammar should be discarded. *Muffle-conflicts*, *print-derives-epsilon*, *print-first-terminals*, *print-states*, *print-goto-graph* and *print-lookaheads* are as in `define-parser`.

2.4 Conditions

CL-Yacc may signal warnings at compile time when it finds conflicts. It may also signal an error at parse time when it finds that the input is incorrect.

2.4.1 Compile-time conditions

If the grammar given to CL-Yacc is ambiguous, a warning of type `conflict-warning` will be signalled for every conflict as it is found, and a warning of type `conflict-summary-warning` will be signalled at the end of parser generation.

`conflict-warning` *kind state terminal* [Condition]
Signalled whenever a conflict is found. *Kind* is one of `:shift-reduce` or `:reduce-reduce`. *State* (an integer) and *terminal* (a symbol) are the state and terminal for which the conflict arises.

`conflict-summary-warning` *shift-reduce reduce-reduce* [Condition]
Signalled at the end of parser generation if there were any conflicts. *Shift-reduce* and *reduce-reduce* are integers that indicate how many conflicts were found.

`yacc-compile-warning` [Condition]
A superclass of `conflict-warning` and `conflict-summary-warning`, and a convenient place to hook your own condition types.

2.4.2 Runtime conditions

If the output cannot be parsed, the parser will signal a condition of type `yacc-parse-error`. It should be possible to invoke a restart from a handler for `yacc-parse-error` in order to trigger error recovery, but this hasn't been implemented yet.

`yacc-parse-error` *terminal value expected-terminals* [Condition]
Signalled whenever the input cannot be parsed. The symbol *terminal* is the terminal that couldn't be accepted; *value* is its value. *Expected-terminals* is the list of terminals that could have been accepted in that state.

`yacc-runtime-error` [Condition]
A superclass of `yacc-parse-error`, and a convenient place to hook your own condition types.

Acknowledgements

I am grateful to Antonio Bucciarelli, Guy Cousineau and Marc Zeitoun for their help with implementing CL-Yacc.

Copying

Index