An LP Parser for Lisp

Gene Michael Stover

created Monday, 2005 February 28
updated Tuesday, 2005 March 15

Contents

Chapter 1

Introduction

Under construction. Come back in a few weeks.

A parser-generator. Its input is a grammar expressed as Lisp data. Its output is the table(s), as Lisp data, which are required for the LR(1) parsing algorithm. The LR(1) parsing algorithm is implemented as a Lisp function. So you can feed a grammar to the table-generator function, & it gives you tables which parameterize the LR parser, which is a Lisp function.

Chapter 2

Goals

Here are some simple high-level examples of what I wish the accomplish.

2.1 Obtain parse tree for a file

This is probably the main use I have for a parser generator.

Given a grammar as a Lisp data structure (probably a list of lists), & an input file containing data that is described by that grammar, I'd like to obtain the parse tree for that file by doing this or something like it:

```
lisp> (defvar *grammar* '((s -> statements)
                          (statements -> stmt ; stmt)
                          ...))
*GRAMMAR*
lisp> (defun lexer (strm) ...)
LEXER
lisp> (with-open-file (strm "input.txt")
        (setq tree (parse *grammar* #'(lambda ()
                                        (lexer strm)))))
(S (STATEMENTS (STMT ...) (STATEMENTS (STMT ...) ...)))
```

The important parts of this example include:

- the grammar is a Lisp data structure,
- the parser function uses that grammar as an argument,
- the parser uses the lexical analyzer as an argument,
- the parser returns the parse tree on success.

If my program were a compiler, I would have a function whose argument was the parse tree & which returned the resulting compiled code (or wrote it to a file or whatever).

The PARSE function has the grammar as its argument. That PARSE function is not generated for the grammar from a compiler compiler. I often wonder why compiler compilers for other languages so often insist on writing new functions. After all, there is only one LR(1) parsing algorithm, & it is parameterized with two tables generated from the grammar; there is no need to create a new parser function for each grammar.

Creating the tables from the grammar might require a non-trivial amount of work, so the PARSE function will probably be a simple wrapper, like this:

```
(defun parse (grammar lex)
  (parse-lr1 (make-canonical-lr1-tables grammar) lex))
```

If a program needed to parse many files using the same grammar, it could generate the parse tables once & call PARSE-LR1 many times instead of calling the PARSE function.

2.2 Evaluate the input as expressions are reduced

I most often prefer to obtain the entire parse tree, then process it, but sometimes a program needs to process expressions as the LR parser reduces them. My parser function should offer a callback function to do that. Here's an example:

```
lisp> (defvar *grammar* '((S -> list) (list -> list E)
                          (list -> E)
                          (E -> ...)
                          ...))
*GRAMMAR*
lisp> (defun callback (rule &rest args)
        (cond ((equal rule '(E -> ...))
               ;; evaluate the expression
               )
              ((equal rule '(list -> list E))
               ;; handle list -> list E
               )
              (...)
              (t error))
        'callback)
CALLBACK
lisp> (with-open-stream (strm "input.txt")
        (parse *grammar* #'(lambda () (lex strm))
               :dispatch #'callback))
CALLBACK
```

The important things here are:

- The parse function allows the caller to specify a callback function that is called each time the LR parser reduces an expression.

- On success, the parser returns the last value returned by the callback function.

The case in which PARSE returns the parse tree should be a special case in which the callback function is `#'LIST`, or something like that.

2.2.1 Callbacks in the grammar

I chose not to insert the callback actions in the grammar because in my experience, a grammar should not be welded to the callbacks. You very often need to generate a tree from a grammar for debugging or so you can write the callbacks for it.

If someone really wanted their actions embedded in the grammar, they might define their grammars like this:

```
lisp> (defvar *grammar*
        (list
          (list '(S -> A B C) #'do-s)
          (list '(A -> A D)   #'do-a0)
          (list '(A -> D)     #'do-a1)
          ...))
```

Then they might have a function which takes an action-embedded grammar like that & returns a stripped-down grammar as I've used in the previous examples. They could also write a function which takes the action-embedded grammar & returns a single callback function that maps rules to the actions. Then they could use the action-embedded grammar like this:

```
lisp> (parse (strip-grammar *grammar*)
             (make-callback-from-action-grammar *grammar*))
```

2.3 No ORs

Grammars are often expressed with conditionals, like this small BNF grammar:

```
S := E + E | E - E ;
E := T | ( S ) ;
T := E * E | E / E ;
```

The grammars for my PARSE function don't do that. If you would write the grammar with an *or*, then in my grammars, you write multiple rules for the same term. Like this:

```
;; Lisp data
((S E + E))
 (S E - E)
 (E T)
 (E |(| S |)|)   ; symbols named ( and )
 (T E * E)
 (T E / E))
```

If someone really needed a grammar with *or* in it, she might write the grammar like this:

```
;; Lisp data
((S (or (E + E) (E - E)))
 (E (or T (|(| S |)|)))
 (T (or (E * E) (E / E))))
```

Given a grammar with *or* in it like the one above, it wouldn't be too difficult to write a function which expanded the *or*s into explicit rules. You could use that function to preprocess the grammar with *or* in it before giving the result to my PARSE function.

Chapter 3

Specification

3.1 Grammar

A grammar is a list of lists. I could have used structures, but I figure lists are general & are already defined.

Figure 3.1 shows the rules in English for a grammar.

Figure 3.2 shows an example grammar list which corresponds to the rules from Figure 3.1.

You might notice that this format of grammar lists does not include a `->` symbol in the SECOND of a production. That symbol might make the grammar easier to read, but it doesn't do anything else useful, so I chose to drop it.

3.2 Parser Table

The PARSE-LR function doesn't consume grammar lists directly. Instead, it uses a table that describes a state machine which can recognize expressions in the grammar. The table are complicated & probably generated from the grammar by another function, not by hand.

Figure 3.3 shows the rules for a table for the PARSE-LR function.

1. A *grammar list* is a list of *productions*.
2. A *production* is a list whose FIRST is the production's name & whose REST are the names of other productions or of terminals.
3. The names may be symbols (recommended), strings, or numbers. They will be compared with EQUAL.

Figure 3.1: The rules which describe grammar lists

```
((E E + T)
 (E T)
 (T T * F)
 (T F)
 (F |(| E |)|)
 (F id))
```

Figure 3.2: A template for grammar lists

1. A table for PARSE-LR is a hash table.
2. The keys of the table are lists of two elements.
3. The FIRST of a key is a state. It could be a symbol, number, or string. I recommend an integer.
4. The SECOND is the name of a production or the type of a terminal. It may be a symbol, number, or string. I recommend a symbol.
5. The values of the table are lists of two elements.
6. A value's FIRST is the *action part*.
7. An *action part* is a list. It's FIRST is one of the symbols ACCEPT, REDUCE, or SHIFT.
8. If an *action part*'s FIRST is ACCEPT, the REST of that action part may be NIL.
9. If an *action part*'s FIRST is SHIFT, it's SECOND is a state.
10. If an *action part*'s FIRST is REDUCE, it's REST is a production.
11. A value's SECOND is the *goto part*.
12. A *goto part* is a state. I recommend integers, but it could also be symbols or strings.

Figure 3.3: The rules which describe a table for PARS-LR

(state token)	(action goto)
(0 id)	((shift 5))
(0 |(|)	((shift 4))
(0 E)	(nil 1)
(0 T)	(nil 2)
(0 F)	(nil 3)
(1 +)	((shift 6))

Table 3.1: A table for PARSE-LR, derived from Figure 3.2

Table 3.1 shows a table derived from the grammar in Figure 3.2. Table 3.1 is the Lisp translation of Figure 4.31 from [aRSaJDU86].

Chapter 4

Implementation

4.1 Function PARSE-LR

Function PARSE-LR is a translation from the "LR parsing program" of Figure 4.30 in [aRSaJDU86].

To avoid a monolithic DO loop, I restructured the algorithm into *next* functions for each temporary variable. I suspect there is a more compact, more understandable form still, but I have not yet found it. I hope that form is not purely recursive; I would like to avoid a recursive structure for this function.

The book's algorithm uses a stack whose entries can be either a state or a production. That could work in Lisp, too, but it is easier to extract items from the stack if each stack item is a pair whose CAR is a state & whose CDR is a production.

The final difference in my translation of the algorithm is that instead of having an *action* table & a *goto* table, I have one table whose values are pairs. The CAR of a value pair is the *action* part. The CDR is the *goto* part. The *action* part is a list whose CAR is SHIFT, REDUCE, or ACCEPT & whose CDR is a production (which is itself a list). The *goto* part is a state. (At this time, I don't know how states are represented. They will probably be integers.)

Here's the basic part of the PARSE-LR function:

4.2 Function PARSE

Appendix A

Other File Formats

- This document is available in multi-file HTML format at http://lisp-p.org/ctlr/.
- This document is available in Pointless Document Format (PDF) at http://lisp-p.org/dhn/ctlr.pdf.

Bibliography

[aRSaJDU86] Alfred V. Aho Ravi Sethi Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, March 1986. ISBN 0-201-10088-6; often called "the dragon book".