Not Yet Another Compiler-Compiler!

A LALR(1) Parser Generator Implemented in Guile

Matt Wette

1 Demonstration

WARNING: This manual is work in progress.

A LALR(1) parser is a pushdown automata for parsing computer languages. In this tool the automata, along with its auxiliary parameters (e.g., actions), is called a *machine*. The grammar is called the *specification*. The program that processes, driven by the machine, input token to generate a final output, or error, is the *parser*.

1.1 A Simple Batch Calculator

A simplest way to introduce working with NYACC@texinfoc is to work through an example. Consider the following contents of the file `calc1.scm` in the distributed directory `examples/nyacc/lang/calc/`:

```
(add-to-load-path "../../../../module")

(use-modules (nyacc lalr))
(use-modules (nyacc lex))
(use-modules (nyacc parse))

(define spec
  (lalr-spec
   (prec< (left "+" "-") (left "*" "/"))
   (start expr)
   (grammar
    (expr
     (expr "+" expr ($$ (+ $1 $3)))
     (expr "-" expr ($$ (- $1 $3)))
     (expr "*" expr ($$ (* $1 $3)))
     (expr "/" expr ($$ (/ $1 $3)))
     ($fixed ($$ (string->number $1)))
     ($float ($$ (string->number $1)))
     ("(" expr ")" ($$ $2))))))

(define mach (make-lalr-machine spec))
(define raw-parser (make-lalr-parser mach))
(define gen-lexer (make-lexer-generator (lalr-match-table mach)))

(define (calc1-eval str)
  (with-input-from-string str
    (lambda () (raw-parser (gen-lexer)))))

(define (calc1-demo string)
  (simple-format #t "~A => ~A\n" string (calc1-eval string)))

(demo "2 + 2")
```

Here is an explanation of the above code:

1. The relevent modules are imported using Guile's `use-modules` syntax.

2. The syntax form `lalr-spec` is used to generate a (canonical) specification from the grammar and options provided in the form.
3. The `prec<` directive indicates that the tokens appearing in the sequence of associativity directives should be interpreted in increasing order of precedence. The associativity statements `left` indicate that the tokens have left associativity. So, in this grammar `+`, `-`, `*`, and `/` are left associative, `*` and `/` have equal precedence, `+` and `-` have equal precedence, but `*` and `/` have higher precedence than `+` and `-`.
4. The `start` directive indicates which left-hand symbol in the grammar is the starting symbol for the grammar.
5. The `grammar` directive is used to specify the production rules. In the example above one left-hand side is associated with multiple right hand sides. But this is not required.
   - Multiple right-hand sides can be written for a single left-hand side.
   - Non-terminals are indicated using symbols (e.g., `expr`).
   - Terminals are indicated using constants or reserved symbols. Constants take the form of string literals (e.g.,`"+"`), character literals (e.g., `#\+`), or quoted symbols (e.g., `'+`). In this example, reserved symbols used as terminals are `$fixed` and `$float`. Note that tokens or terminals to not need to be declared as in bison or the Guile lalr module.
   - The reserved symbols `$fixed` and `$float` indicate an unsigned integer and floating point number, respectively. The NYACC@texinfoc procedures for generating lexical analyzers will emit this token when the corresponding numbers are detected in the input. In general, symbols starting with `$` are reserved by NYACC@texinfoc .
   - Within the right-hand side specification a `$$` form is used to specify an action associated with the rule. Ordinarily, the action appears as the last element of a right-hand side, but mid-rule actions are possible. Inside the `$$` form, the variables `$1`, `$2`, etc. refer to the symantic value of the corresponding item in the rule.
   - The expression returned by `lalr-spec` is an association list; you can peek at the internals with Guile.
6. The data structures comprising the automaton (aka machine) is generated using the procedure `make-lalr-machine`. This routine does the bulk of the processing to produce an LALR(1) automata. The value returned by this procedure is also an associative list.
7. Generating a usable parser procedure requires a few steps. The first is to create a raw parser:

   ```
   (define raw-parser (make-lalr-parser mach)))
   ```

   Note that `raw-parser` is a thunk: a procedure of no arguments. This code generates a parser (procedure) from the machine and the match table. The match table is the handshake between the lexical analyzer and the parser for encoding tokens. In this example the match table is symbol based, but there is an option to hash these symbols into integers. See [Hashing and Compacting], page 8,
8. The next task is to create a generator for lexical analyzers. This is performed as follows:

   ```
   (define gen-lexer (make-lexer-generator (lalr-match-table mach)))
   ```

   We create a generator here because a lexical analyzer may require internal state (e.g., line number, mode). The generator is constructed from the *match table* provided by

the machine. The procedure `make-lexer-generator` is imported from the module `(nyacc lex)`. Optional arguments to `make-lexer-generator` allow the user to specify custom readers for identifiers, comments, numbers, etc. See [lex], page 7,

9. The actual parser/evaluator we use calls the raw parser with a lexical analyser created from the generator.

```
(define (calc1-eval str)
  (with-input-from-string str
    (lambda () (raw-parser (gen-lexer)))))
```

   The lexical analyzer reads code from `(current-input-port)` so we set up the environment using `with-input-from-string`. See Section "Input and Output" in `guile`

10. And now we can run it:

```
(calc1-eval "1 + 1") => 2
```

If we execute the example file above we should get the following:

```
$ guile calc1.scm
2 + 2 => 4
$
```

1.2 An Interactive Calculator

If one uses the above code to make an interactive parser it will get stuck, requiring multiple return keystrokes to get output. If you replace `make-lalr-parser` with `make-lalr-ia-parser` then you get an interactive parser. See the example `calc2.scm` in the same directory. (Note: If you look at the NYACC@texinfoc parse module you will see that the baser parser is quite a bit cleaner than the ia-parser, hence the motivation to provide both, at least for now.) Here is an interactive calculator. Note that it uses mid-rule actions and other features not discussed above. Details are in the sequel.

```
(use-modules (nyacc lalr))
(use-modules (nyacc lex))
(use-modules (nyacc parse))

(define (next) (newline) (display "> ") (force-output))

(define calc2-spec
  (lalr-spec
   (prec< (left "+" "-") (left "*" "/"))
   (start stmt-list)
   (grammar
    (stmt-list
     (stmt)
     (stmt-list stmt))
    (stmt
     (expr ($$ (display $1) (next)) "\n"))
    (expr
     ($empty)
     (expr "+" expr ($$ (+ $1 $3)))
```

```
      (expr "-" expr ($$ (- $1 $3)))
      (expr "*" expr ($$ (* $1 $3)))
      (expr "/" expr ($$ (/ $1 $3)))
      ($fixed ($$ (string->number $1)))
      ($float ($$ (string->number $1)))
      ("(" expr ")" ($$ $2))))))

(define calc2-mach (make-lalr-machine calc2-spec))
(define match-table (assq-ref calc2-mach 'mtab))
(define parse (make-lalr-ia-parser calc2-mach))
(define gen-lexer
  (make-lexer-generator match-table #:space-chars " \t"))

(next)
(parse (gen-lexer))
```

1.3 Generating a Language to run in Guile

One cool feature of Guile is that it provides a backend infrastructure for evaluation of multiple languages. The files `parser.scm`, `compiler.scm` in the `nyacc/lang/calc` directory and `spec.scm` in the `language/calc` directory implement our calculator in Guile. To execute the calculator as follows:

```
$ cd examples
$ guile -L ../module -L .
...
scheme@(guile-user)> ,L calc
...
Happy hacking with calc!  To switch back, type ‘,L scheme’.
calc@(guile-user)> 1 + 1
2
calc@(guile-user)>
```

Our evaluator uses SXML as the intermediate representation between the parser and compiler (to tree-il). See also the example in the `examples/nyacc/lang/javascript` directory.

1.4 Running with the Debugger

You can run with debugging on. In `calc1.scm` one can see a modified version of `calc1-eval` which will print out debugging info:

```
(define (calc1-eval str)
  (with-input-from-string str
    (lambda () (raw-parser (gen-lexer) #:debug #t))))
```

To make use of this info you probably want to generate an output file as describe in Section [Human Readable Output], page 9.

2 Parsing

Most of the syntax and procecures for generating skelton parsers exported from the module `(nyacc lalr)`. Other modules include

(lalr lex) This is a module providing procedures for generating lexical analyzers.

(lalr util) This is a module providing utilities used by the other modules.

2.1 The Grammar Specification

The syntax for generating specifications is `lalr-spec`. As mentioned in the previous chapter, the syntax generates an association list, or *a-list*.

`lalr-spec` *grammar* `=>` *a-list* [Syntax]

This routine reads a grammar in a scheme-like syntax and returns an a-list. The returned a-list is normally used as an input for `make-lalr-machine`. The syntax is of the form

```
(lalr-spec (directive ...) ...)
```

The order of the directives does not matter, but typically the `grammar` directive occurs last.

The directives are

`notice` bla

`reserve` This is a list of tokens which do not appear in the grammar but should be added to the match table.

`prec< prec>`
these psecific pre

`expect` This is the expected number of shift-reduce conflicts to occur.

`start` This specifies the top-level starting non-terminal.

`grammar` the grammar see below

Notably the grammar will have rhs arguments decorated with type (e.g., `(terminal . #\,)`).

Each production rule in the grammar will be of the form `(lhs rhs1 rhs2 ...)` where each element of the RHS is one of

- `('terminal . atom)`
- `('non-terminal . symbol)`
- `('action . (ref narg guts)`
- `('proxy . production-rule)`

Currently, the number of arguments for items is computed in the routine `process-grammar`.

We have (experimental) convenience macros:

```
($? foo bar baz) => ``foo bar baz'' occurs never or once
```

```
($* foo bar baz) => ‘‘foo bar baz’’ occurs zero or more times
($+ foo bar baz) => ‘‘foo bar baz’’ occurs one or more times
```

However, these have hardcoded actions and are considered to be, in current form, unattractive for practical use.

Todo: discuss

- reserved symbols (e.g., `$fixed`, `$ident`, `$empty`)
- Strings of length one are equivalent to the corresponding character.
- `(pp-lalr-grammar calc-spec)`
- `(pp-lalr-machine calc-mach)`
- `(define calc-mach (compact-mach calc-mach))`
- `(define calc-mach (hashify-Machine calc-mach))`
- The specification for `expr` could have been expressed using

```
(expr (expr "+" expr ($$ (+ $1 $3))))
(expr (expr "-" expr ($$ (- $1 $3))))
(expr (expr "*" expr ($$ (* $1 $3))))
(expr (expr #\/ expr ($$ (/ $1 $3))))
(expr ('$fx ($$ (string->number $1))))
```

- rule-base precedence
- multiple precedence statements so that some items can be unordered

```
(prec< "then" "else")
(prec< "t1" "t2" "t3" "t4" "t5")
=> ((t1 . t2) (t2 . t3) (t3 . t4) (t4 . t5) (then . else))
```

2.2 Other Directives

```
(expect 1)
(notice "Copyright (C) 2017 John Doe")
(reserve "abc" "def")
```

2.3 Recovery from Syntax Errors

The grammar specification allows the user to handle some syntax errors. This allows parsing to continue. The behavior is similar to parser generators like *yacc* or *bison*. The following production rule-list allows the user to trap an error.

```
(line
  ("\n")
  (exp "\n")
  ($error "\n"))
```

If the current input token does not match the grammar, then the parser will skip input tokens until a `"\n"` is read. The default behavior is to generate an error message: "*syntax error*". To provide a user-defined handler just add an action for the rule:

```
(line
  ("\n")
  (exp "\n")
```

```
($error "\n" ($$ (format #t "line error\n"))))
```

Note that if the action is not at the end of the rule then the default recovery action ("*syntax error*") will be executed.

2.4 The Lex Module

The NYACC@texinfoc `lex` module provide routines for constructing lexical analyzers. The intension is to provide routines to make construction easy, not necessarily the most efficient.

2.5 The Match Table

In some parser generators one declares terminals in the grammar file and the generator will provide an include file providing the list of terminals along with the associated "hash codes". In NYACC@texinfoc the terminals are detected in the grammar as non-identifiers: strings (e.g., `"for"`), symbols (e.g., `'$ident`) or characters (e.g., `#\+`). The machine generation phase of the parser generates a match table which is an a-list of these objects along with the token code. These codes are what the lexical analyzer should return. BLA Bla bla. So in the end we have

- The user specifies the grammar with terminals in natural form (e.g., `"for"`).
- The parser generator internalizes these to symbols or integers, and generates an a-list, the match table, of (natural form, internal form).
- The programmer provides the match table to the procedure that builds a lexical analyzer generator (e.g., `make-lexer-generator`).
- The lexical analyzer uses this table to associate strings in the input with entries in the match table. In the case of keywords the keys will appear as strings (e.g., `for`), whereas in the case of special items, processed in the lexical analyzer by readers (e.g., `read-num`), the keys will be symbols (e.g., `'$fl`).
- The lexical analyzer returns pairs in the form (internal form, natural form) to the parser. Note the reflexive behavior of the lexical analyzer. It was built with pairs of the form (natural form, internal form) and returns pairs of the form (internal form, natural form).

Now one item need to be dealt with and that is the token value for the default. It should be `-1` or `'$default`. WORK ON THIS.

2.6 Parsing a Sublanguage of a Specification

Say you have a NYACC@texinfoc specification `cspec` for the C language and you want to generate a machine for parsing C expressions. You can go this via

```
(define cxspec (restart-spec cspec 'expression))
(define cxmach (make-lalr-machine cxspec))
```

2.7 The Parser-Lex'er Interface

To be documented.

```
typedef int foo_t;
foo_t x;
```

2.8 Parser Tables

Note that generating a parser requires a machine argument. It is possible to export the machine to a pair of files and later regenerate enough info to create a parser from the tables saved in the machine.

For example, Tables can be generated

```
(write-lalr-actions calc1-mach "calc1-act.scm")
(write-lalr-tables calc1-mach "calc1-tab.scm")
```

Then, without reference to the original specification or need to run `make-lalr-machine`, you can ...

```
(include "calc1-tab.scm")
... code for parser ...
(include "calc1-act.scm")
```

Check some of the examples in the NYACC@texinfoc distribution.

2.9 Hashing and Compacting

The procedure `compact-machine` will compact the parse tables. That is, if multiple tokens generate the same transition, then these will be combined into a single *default* transition. Ordinarily NYACC@texinfoc will expect symbols to be emitted from the lexical analyzer. To use integers instead, use the procedure `hashify-machine`. One can, of course, use both procedures:

```
(define calc-mach
  (compact-machine
   (hashify-machine
     (make-lalr-machine calc-spec))))
```

2.10 Exporting Parsers

NYACC@texinfoc provides routines for exporting NYACC@texinfoc grammar specifications to other LALR parser generators.

The Bison exporter uses the following rules:

- Terminals expressed as strings which look like C identifiers are converted to symbols of all capitals. For example `"for"` is converted to `FOR`.
- Strings which are not like C identifiers and are of length 1 are converted to characters. For example, `"+"` is converted to `'+'`.
- Characters are converted to C characters. For example, `#\!` is converted to `'!'`.
- Multi-character strings that do not look like identifiers are converted to symbols of the form `ChSeq_`*i*`_`*j*`_`*k* where *i*, *j* and *k* are decimal representations of the character code. For example `"+="` is converted to `ChSeq_43_61`.
- Terminals expressed as symbols are converted as-is but `$` and `-` are replaced with `_`.

TODO: Export to Bison xml format.

The Guile exporter uses the following rules: TBD.

2.11 Debugging

2.11.1 Human Readable Output

You can generate text files which provide human-readable forms of the grammar specification and resulting automaton, akin to what you might get with bison using the '-r' flag.

```
(with-output-to-file "calc1.out"
  (lambda ()
    (pp-lalr-grammar calc1-mach)
    (pp-lalr-machine calc1-mach)))
```

The above code will generate something that looks like

```
0 $start => stmt-list
1 stmt-list =>
2 stmt-list => stmt-list $P1 stmt
3 $P1 =>
4 stmt => "\n"
g5 stmt => expr "\n"
6 expr => expr "+" expr
7 expr => expr "-" expr
8 expr => expr "*" expr
9 expr => expr "/" expr
10 expr => "*" '$error
11 expr => '$fixed
12 expr => '$float
13 expr => "(" expr ")"

0:  $start => . stmt-list
stmt-list => .
stmt-list => . stmt-list $P1 stmt
stmt-list => shift 1
'$end => reduce 1
"(" => reduce 1
'$float => reduce 1
'$fixed => reduce 1
"*" => reduce 1
"\n" => reduce 1

1:  stmt-list => stmt-list . $P1 stmt
$P1 => .
$start => stmt-list .
$P1 => shift 2
"(" => reduce 3
'$float => reduce 3
'$fixed => reduce 3
"*" => reduce 3
"\n" => reduce 3
'$end => accept 0
```

```
...

21:  expr => expr . "/" expr
expr => expr . "*" expr
expr => expr . "-" expr
expr => expr . "+" expr
expr => expr "+" expr .
"+" => reduce 6
"-" => reduce 6
"*" => shift 13
"/" => shift 14
"\n" => reduce 6
")" => reduce 6
["+" => shift 11] REMOVED by associativity
["-" => shift 12] REMOVED by associativity
["*" => reduce 6] REMOVED by precedence
["/" => reduce 6] REMOVED by precedence
```

3 Translation

Under ‘`examples/nyacc`’ are utilities for translating languages along with some samples. The approach that is used here is to parse languages into a SXML based parse tree and use the SXML modules in Guile to translate. We have built a javascript to tree-il translater which means that one can execute javascript at the Guile command line:

```
scheme@(guile-user)> ,L javascript
need to complete
```

3.1 Tagged-Lists

In actions in NYACC@texinfoc can use our tagged-lists to build the trees. For example, building a statement list for a program might go like this:

```
(program
 (stmt-list ($$ `(program ,(tl->list $1))))
 (...))
(stmt-list
 (stmt ($$ (make-tl 'stmt-list $1)))
 (stmt-list stmt ($$ (tl-append $1 $2))))
```

3.2 Working with SXML Based Parse Trees

To work with the trees described in the last section use

```
(sx-ref tree 1)
(sx-attr tree)
(sx-attr-ref tree 'item)
(sx-tail tree 2)
```

3.3 Example: Converting Javascript to Tree-IL

This illustrates translation with `foldts*-values` and `sxml-match`.

4 Administrative Notes

4.1 Installation

Installation instructions are included in the top-level file `README` of the source distribution.

4.2 Reporting Bugs

Bug reporting will be dealt with once the package is place on a publically accessible source repository.

4.3 The Free Documentation License

The Free Documentation License is included in the Guile Reference Manual. It is included with the NYACC@texinfoc source as the file COPYING.DOC.

5 TODOs, Notes, Ideas

Todo/Notes/Ideas:

16 add error handling (lalr-spec will now return #f for fatal error)

3 support other target languages: (write-lalr-parser pgen "foo.py" #:lang 'python)

6 export functions to allow user to control the flow i.e., something like: (parse-1 state) => state

9 macros - gotta be scheme macros but how to deal with other stuff (macro ($? val ...) () (val ...)) (macro ($* val ...) () (_ val ...)) (macro ($+ val ...) (val ...) (_ val ...)) idea: use $0 for LHS

10 support semantic forms: (1) attribute grammars, (2) translational semantics, (3) operational semantics, (4) denotational semantics

13 add ($abort) and ($accept)

18 keep resolved shift/reduce conflicts for pp-lalr-machine (now have rat-v – removed action table – in mach, need to add to pp)

19 add a location stack to the parser/lexer

22 write parser file generator (working prototype)

25 think

26 Fix lexical analyzer to return tval, sval pairs using `cons-source` instead of `cons`. This will then allow support of location info.

6 References

[DB] Aho, A.V., Sethi, R., and Ullman, J. D., "Compilers: Principles, Techniques and Tools," Addison-Wesley, 1985 (aka the Dragon Book)

[DP] DeRemer, F., and Pennello, T., "Efficient Computation of LALR(1) Look-Ahead Sets." ACM Trans. Prog. Lang. and Systems, Vol. 4, No. 4., Oct. 1982, pp. 615-649.

[RPC] R. P. Corbett, "Static Semantics and Compiler Error Recovery," Ph.D. Thesis, UC Berkeley, 1985.