Not Yet Another Compiler-Compiler!

A LALR(1) Parser Generator Implemented in Guile

Matt Wette

1 Introduction

WARNING: This manual is currently in a very immature state.

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 Example

A simplest way to introduce working with `nyacc` is to work through an example. Consider the following contents of the file `calc.scm`.

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

(define calc-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 $3)))
     ('$fx ($$ (string->number $1)))))))

(define calc-mach (make-lalr-machine calc-spec))

(define parse-expr
  (let ((gen-lexer (make-lexer-generator (assq-ref calc-mach 'mtab)))
(calc-parser (make-lalr-parser calc-mach)))
    (lambda () (calc-parser (gen-lexer)))))

(define res (with-input-from-string "1 + 4 / 2 * 3 - 5" parse-expr))
(simple-format #t "expect 2; get ~S\n" res) ;; expect: 2
```

Here is an explanation of the code:

1. The relevent modules are imported using guile's `use-modules` syntax.
2. The `lalr-spec` syntax is used to generate a (canonical) specification from the grammar and options. The syntax is imported from the module `(nyacc lalr)`.
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 `-`. (Note: this syntax may change in the future.)

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 as normal identifiers.
   - Terminals are indicated as non-identifiers using double-quotes (e.g., `"+"`), scheme character syntax (e.g., `#\+`), or quoted identifiers (e.g., `'+`). There is no syntax to declare tokens.
   - The reserved symbol `'$fx` indicates an unsigned integer. The lexical analyzer tools will emit this token when an integer is detected in the input.
   - A quoted identifier cannot match a normal identifier. For example, one could not use `function` to indicate a non-terminal and `"function"` to indicate a terminal. The reader will signal an error when this condition is detected.
   - 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 (see Section TBD).
   - The output of `lalr-spec` is an associative array so you can peek at the internals using standard Scheme procedures.
6. The machine is generated using the procedure `make-lalr-machine`. This routine does the bulk of the processing to produce an LALR(1) automata.
7. Generating a parser function requires a few steps. The first step we use is to create a lexical analyzer (generator).

   ```
   (gen-lexer (make-lexer-generator (assq-ref calc-mach 'mtab)))
   ```

   We build a generator because a lexical analyzer may require 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 how identifiers, comments, numbers, etc are read in.
8. The next item in the program is

   ```
   (calc-parser (make-lalr-parser calc-mach)))
   ```

   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 Section TBD.
9. The actual parser that we use calls the generated parser with a lexical analyser created from the generator.

   ```
   (lambda () (calc-parser (gen-lexer)))))
   ```

   Note that `parse-expr` is a thunk: a procedure of no arguments.
10. Now we run the parser on an input string. The lexical analyzer reads code from `(current-input-port)` so we set up the environment using `with-input-from-string`. See the Input/Ouput section of the Guile Reference Manual for more information.

    ```
    (define res (with-input-from-string "1 + 4 / 2 * 3 - 5" parse-expr))
    ```

11. Lastly, we print the result out along with the expected result.

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

```
$ guile calc.scm
expect 2; get 2
$
```

1.2 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 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 Modules

nyacc provides several modules:

lalr — This is a module providing macros for generating specifications, machines and parsers.

lex — This is a module providing procedures for generating lexical analyzers.

util — This is a module providing utilities used by the other modules.

2.1 The `lalr` Module

WARNING: This section is quite crufty.

The `lalr1` module provides syntax and procedures for building LALR parsers. The following syntax and procedures are exported:

- `lalr-spec` syntax
- `make-lalr-machine` procedure

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., `'$fx`, `'$ident`)
- 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))))
```

2.2 The `lex` Module

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

2.3 The `export` Module

NYACC provides routines for exporting NYACC 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.

3 Implementation

The implementation is based on algorithms laid out in the Dragon Book. See Chapter 6 [References], page 12. In NYACC one writes out a (context-free) grammer using Backus-Naur form. See the example in Chapter TBD. In addition to the grammar the start symbol must be provided. Optional inputs include specifiers for precedence and associativity.

3.1 Preliminaries

Consider a set of symbols *T*. A *string* is a sequence of symbols from *T*. A *language* is a set of strings. Now we can introduce the following. A *context free grammar* is defined by the aggregate [CHECK]

terminals A set of symbols *T* which are used to compose a language.

non-terminals
A set of symbols *N*, disjoint from *T*, used in production rules.

production rules
A set of production rules, to be defined below.

start symbol
This is a symbol which represents an entire string from the language. (CHECK)

A *production* consists of a left-hand side (LHS) symbol from *N* and a right-hand side (RHS) which is a sequence of symbols from the union of *T* and *N*.

In the following we use capital letters for non-terminals, the lower case letters a-h for terminals and the lower case letters p-z (sometimes in italics) for strings, where a string is defined as a sequence of non-terminals and terminals. We occasionally use italic capital T (i.e., *T*) to represent a set of terminals.

An *item* is a position within a production rule. It is represented by the notation

```
A => p.q
```

where the dot `.` represents the position in the production rule. In NYACC we use a pair (a cons cell) of integers to represent an item: the car is the index of the p-rule in the grammar and the cdr is the index into the RHS of the symbol to the right of the dot. If the position is at the end (past the last symbol in the RHS), then the cdr contains -1.

Given a production rule, a *lookahead* is a terminal that can appear after the production in the grammar. We will associate an item with the set of possible lookaheads in the context of parsing an input string from left to right. A *la-item* (sometimes called a LR(1) item) is the explicit association of the item with the lookahead set. We denote this with the following notation

```
A => p.q, {a, b, c}
```

An important function used in NYACC is `first`. It has the following signature:

```
first string t-set => la-t-set
```

where *string* is a sequence of grammar symbols and the argument *t-set* and result *la-t-set* are sets of terminals. The routine `first` computes the set of terminals that apear the front of *string* followed by any terminals in the argument *t-set*. If *string* is empty, then the result

will be just the argument *t-set*. If *string* starts with a terminal, then the result will be a singleton consisting of that terminal.

In NYACC the canonical grammer will always include the internally generated accept production

```
$start => S
```

where 'S' is the start symbol of the grammer. The symbol `$start` has only this single production, and this production always has index zero. (Note that Bison includes the endmarker `$end` in this production, which results in Bison parsers having one additional state with respect to NYACC.)

Now consider starting to parser an input, left to right. The initial al-item derived from the cananical 0-index production, with the singleton lookahead set consisting of `{$end}`, will be in effect:

```
$start => .S, {$end}
```

Note that `$end` is the only lookahead for the 0-index production. Now `S` may have have several associated production rules. Assume they are the following:

```
S => Ap
S => Bq
```

Then the following la-items are in effect

```
S => .Ap, {$end,...}
S => .Bp, {$end,...}
```

At this point we don't know the entire set of possible lookaheads because `A` and `B` may appear as right-hand sides in other rules.

One can imagine that during process of parsing an input left-to-right the parser may be at a state where several productions may be candidates for matching the input. Each (partial) production is represented by an item and the set of these partial productions is called an *itemset*. If the items are associated with a set of lookaheads (i.e., the items are actually la-items) then we may call this a *la-itemset*. These itemsets correspond to the states of the automaton which is the parser.

At the start of parsing the 0-th rule will have all items with dots at position zero. For the remainder of parsing (after we have acted on the first input token, or terminal) at least one effective item will have the dot at position greater than zero. For those itemsets the subset of items with positive dot positions will be called the *kernel itemsets*. For the initial state the kernel itemset is the 0-dot position for the 0-th production rule.

Now consider, in state i, an item as follows

```
i: A => B.q
```

We can associate this with a phony lookahead `$@`, which we call the *anchor*, to make an la-item:

```
i: A => B.q, {@}
```

Let T = `first(q,{@})` and notice then if `@` is in T then any lookaheads for this production must also be lookaheads for XXX

Here is the algorithm from `lalr.scm`

working toward explaining closure: Compute the fixed point of I, aka `la-item-l`, with procedure

```
for each item [A => x.By, a] in I
  each production B => z in G
  and each terminal b in FIRST(ya)
  such that [B => .z, b] is not in I do
    add [B => .z, b] to I
```

It turns out that each state in the parsing automaton is an itemset. We can associate an integer with each state of the automaton. Now consdier, some state i with la-item as follows:

`i: A => p.Bq, {c}`

There will be a transition from state i to state j on symbol B after a reduction of a production for B. Then that the tokens in

`first(q,{c})`

are lookaheads the associated item in state j:

`j: B => r.`

and thus when we build this automaton the set of terminals given by `first(q,{c})` should be added to the lookaheads for j. If there is a production

`i: B => .Cs`

then the tokens in

`first(sq,{c})`

are lookaheads for the associated item in some state k:

`k: B => C.s`

Now let us assume the set of lookaheads for the first item above is the singleton {@} consisting of the dummy, or anchor, token `$@`. We compute the set

`J = first(Bq,$@)`

If `$@` is in J then we say the lookaheads for XXX propagate

```
for-each item I in some itemset
  for-each la-item J in closure(I,#)
    for-each token T in lookaheads(J)
      if LA is #, then add to J propagate-to list
      otherwise add T to spontaneously-generated list
```

Now condider the la-item

```
$start => .S, { $end }
```

where `$end` represents the end of input. Now `$end` will also be the lookahead for any productions of `S`. Say we have read token `x` and our state includes an item of the form

```
S => x.By, { $end }
```

The lookahead `$end` is there because it was propagated from the accept production.

```
B => .z, first(zy, {$end})
```

This says if `z` and `y` have epsilon productions then `$end` will be included in the lookaheads for this la-item in it's associated state.

We define terms

- handle: If S => aAw => abw, then A => b is a handle of abw where w only contains terminals.

4 Administrative Notes

4.1 Installation

Installation instructions are included in the top-level file `README.nyacc` 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 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

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.