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
   (precedence< (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 `precedence<` 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
$
```

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

The `lalr1` module provides syntax and procedures for building LALR parsers. Exports:

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

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

3 Random Implementation Notes

Notes on mid-rule actions: To support mid-rule actions we track:

1. length of rule (len or l)
2. size of stack reduction (red or x)
3. number of args to action (narg or n)

```
X: A B C ()              (l=3, x=3, n=3)
```

Mid-rule actions are expanded via proxy:

```
Z: A ($$ foo) B C ($$ bar) D ($$ baz)
=>
Z: A $P1 B C $P2 D ($$ baz)  (l=2, x=6, n=6)   n[k] = l[k] + n[k=1]
$P1: ($$ foo)        (l=1, x=0, n=1)
$P2: ($$ bar)         (l=3, x=0, n=4)   n[k] = l[k] + n[k=1]
```

If proxy x = 0, else x = len of original list of narg Macros:

```
($? optional stuff)
($* xxx) => zero or more
($+ xxx) => one or more
($.* x yyy) => x plus zero or more yyy
```

All symbols starting with `$` are reserved. The following reserved symbols are in use:

`$start` used in specification to indicate the real start

`$end` long emitted by the lexical analysis to indicate end of input

`$ident` emitted by the lexical analyzer to indicate an identifier

`$fx` emitted by the lexical analyzer to indicate an unsigned integer

`$fl` emitted by the lexical analyzer to indicate an unsigned floating point number

`$P1, ...` used as symbols for proxy productions

`$?, $*, $+`
: macros used for grammar specification

`$$, $$-ref, $$/ref`
: definition of an action, an action reference or an action with reference

`$action, $action-ref, $action/ref`
: long name for `$$`, `$$-ref` and `$$/ref`

`$1, $2, ...`
: arguments to user-supplied actions

`$default` used in the generated parser to indicate a default action

`$dummy` used in the machine generation stage to track epsilon productions

`$epsilon` used in the machine generation stage to indicate an empty production

`$cpp-ident (C parser)`
: used in the (not yet distributed) C parser to indicate a C preprocessor symbol

4 Implementation

The implementation is based on algorithms laid out in the Dragon Book. See Chapter 7 [References], page 10. 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.

The macro `lalr-spec`, with the aid of the procedure `process-spec` reads the user-specified grammar and generates an a-list of the specification in canonical form. The fields of the a-list include

`non-terms`
: The list of non-terminals.

`start`
: The start symbol representing the starting rule for the grammar.

`lhs-v`
: The vector of left-hand side symbols for each production rule.

`rhs-v`
: The vector of vectors of right-hand side symbols for each production rule.

`act-v`
: The vector of actions corresponding to each associated production rule.

`ref-v`
: The vector of action references corresponding to each associated production rule. This will be explained later.

`nrg-v`
: The number of arguments to each action. This is necessary for handling mid-rule-actions.

`expect`
: The expected number of undirected shift-reduce conflicts.

`err-l`
: Not used yet.

The canonical form has no mid-rule-actions: they are reposed using proxy symbols (e.g., `$P1`).

The machine adds the following items:

`pat-v`
: This is the parse action table, that provide per-state transition map.

`kit-v`
: xxx

`kip-v`
: xxx

In the following we use capital letters for non-termianls, lower case a-o for terminals and lower case p-z for strings (defined as a sequence of non-terminals and terminals.

An *item* is position within a production rule. It is represented in the Dragon Book for SLR grammars by the notation

```
A => B . C D
```

We use a box of integers to represent this: the car is the index of the p-rule in the grammar and the cdr is the index of the location in the p-rule.

In the DB an item is used to refer to the position in a production and the position with associated lookaheads that give the possible set of terminals that can generate a reduction when the item is a candidate for reduction (i.e., the dot appears at the end of the p-rule). In this report we use the terms *item* for the position and *la-item* for the position and associated lookaheads. An example of an la-item is as follows:

```
A => B . C D, e/f/g
```

where `e/f/g` is a tuple of terminals which can appear at

```
A => B C D ., e/f/g
```

We denote an item in the code using a box with the index of the p-rule in the car and the index of the right-hand side symbol after the dot in the cdr. The end of p-rule will be denoted with index `-1`. So if the rule '`A=>BCD`' appears as index 7, then the above would be item `(7 . 1)` or la-item `((7 . 1) e f g)`.

Note that the grammer will always include the canonical accept production

```
$start => S
```

where '`S`' is the start symbol of the grammer. The symbol `$start` has only this single production. 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.

5 Administrative Items

5.1 Installation

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

5.2 Reporting Bugs

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

5.3 The Free Documentation License

The Free Documentation License is included in the Guile Reference Manual. It is included with the NYACC source but not expanded here.

6 Todos, Notes, Ideas

Todo/Notes/Ideas:

16 add error handling

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

23 add strings as substitute for symbol (i.e., "+=" instead of 'add-eq) working this – should also add default p-rules

25 next

7 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.