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 for Simple or Development Parser

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 Example for Production Parser

1.2.1 Generating the Tables

1.2.2 Running the Compiler

```
(use-modules (nyacc parser)
(use-modules (nyacc lex)
(use-modules (nyacc lang util)
```

1.3 The Grammar Specification

Explain it all

1.3.1 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.

1.4 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 for Constructing Parsers and Lexical Analyzers

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., `$fixed`, `$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 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 Language 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 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.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

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.