Not Yet Another Compiler-Compiler

A LALR(1) Parser Generator Implemented in Guile
DRAFT User's manual for NYACC version 0.79.0

Matt Wette

Table of Contents

1 Demonstration

This document is a ROUGH DRAFT.

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 is to work through an example. Consider the following contents of the file `calc1.scm` in the distributed directory `examples/nyacc/lang/calc/`:

```
(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)))

(calc1-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 string literals (e.g.,`"+"`), character literals (e.g., `#\+`), quoted symbols (e.g., `'+`) or NYACC reserved symbols, which always begin with `$`. Reserved symbols used in this example are `$fixed` and `$float`. Note that tokens or terminals do not need to be declared as in Bison or the Guile `(system base lalr)` module.
   - The reserved symbols `$fixed` and `$float` indicate an unsigned integer and floating point number, respectively. The NYACC procedures for generating lexical analyzers will emit this token when the corresponding numbers are detected in the input.
   - Within the right-hand side of a production rule 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 right hand side.
   - The expression returned by `lalr-spec` is an association list (a-list); you can peek at the internals using typical Scheme procedures for a-lists.
6. The expression comprising the automaton (aka machine) is generated using the procedure `make-lalr-machine`. This routine does the bulk of the processing to produce what is needed to generate a LALR(1) parser. The result is an association 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)))
   ```

   The procedure `make-lalr-parser` generates a parser (procedure) from the machine. The generated procedure `raw-parser` takes one argument, a lexical analyzer procecdure, and optional keyword arguments.
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 13, 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 17,

9. We bring the above items together to provide a usable procedure for evaluating strings:

```
(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` The raw parser is provided a lex'er.

10. And now we can run it:

```
(calc1-eval "2 + 2") => 4
```

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 sets up the above code to take input from the terminal it will not work as expected, requiring keystrokes beyond a RETURN that completes an expression. If you replace `make-lalr-parser` with `make-lalr-ia-parser` and motify the code a bit, you can get an interactive parser, as shown below. This example appears as `calc2.scm` in the same directory as the example above. (Note: If you look at the NYACC parse module you will see that the base parser is quite a bit cleaner than the ia-parser, hence the motivation to provide both, at least for now.) Note that this example uses mid-rule actions and other features not discussed above.

```
(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 of the many cool features of Guile is that it provides a backend infrastructure for evaluation of multiple frontend languages. The files `parser.scm`, `compiler.scm` in the `examples/nyacc/lang/calc` directory and `spec.scm` in the `examples/language/calc` directory implement our calculator within this Guile infrastructure. To demonstrate the calculator try the following from the `examples` directory.

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

The evaluator uses SXML as the intermediate representation between the parser and compiler, which generates to tree-il. See also the example in the directory `examples/language/javascript` and `examples/nyacc/lang/javascript` directories.

1.4 Debugging Output

The parser can provide debugging output with the appropriate keyword argument. In `calc1.scm` there is 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 18, which provides context for the debugging output. The output looks like

```
state 0, token "2"      => (shift . 3)
state 3, token "+"      => (reduce . 5)
state 0, token expr     => (shift . 4)
state 4, token "+"      => (shift . 5)
state 5, token "2"      => (shift . 3)
state 3, token #<eof>   => (reduce . 5)
state 5, token expr     => (shift . 14)
state 14, token #<eof>  => (reduce . 1)
state 0, token expr     => (shift . 4)
state 4, token #<eof>   => (accept . 0)
2 + 2 => 4
```

2 Parsing

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

`(lalr lex)`
: This module provides procedures for generating lexical analyzers.

`(lalr util)`
: This module provides utilities used by the other modules.

2.1 The 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 (specifier ...) ...)
```

: The order of the specifiers does not matter, but typically the `grammar` specifier occurs last.

The specifiers are

`notice` This is used to push a comment string (e.g., copyright) into the resulting parser tables.

`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 specifiers are used to specify precedence and associativity symbols.

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

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

`alt-start`
: This specifies alternative start symbols used in `restart-spec`. Its use prevents warning messages.

`grammar` the grammar see below

The Notice

The `notice` specifier allows one to provide a comment that will be carried into generated output files (e.g., parse tables generated by `write-lalr-tables`. For example, if the spec' looks like

```
(define spec
  (lalr-spec
   (notice "last edit: Mar 25, 2015")
```

```
...))
```

and one generates parse tables from the machine with `write-lalr-tables` then the resulting file will look like

```
;; calctab.scm

;; last edit: Mar 25, 2015

(define calc-len-v
 #(1 1 ...
 ...
```

The notice is available using the expression

`pp-lalr-notice` *spec* [*port*] [Procedure]
Print the notice to the port, if specified, or `(current-output-port)`.

Reserving Tokens

The `notice` specifier allows one to provide a comment that will be carried into generated output files (e.g., parse tables generated by `write-lalr-tables`. In the javascript parser we have added reserved keywords:

```
(reserve "abstract" "boolean" "byte" "char"
         "class" "const" ...)
```

and this results in a generated match table (for a hashed machine) that looks like:

```
... ("abstract" . 86) ("boolean" . 87) ("byte" . 88)
("char" . 89) ("class" . 90) ("const" . 91) ...
```

Precedence and Associativity

Recall the following specifier from the `calc1` example:

```
(prec< (left "+" "-") (left "*" "/"))
```

This declaration indicates presedence among the math operators. The `<` in `prec<` indicates that the precedence is in increasing order. An equivalent specification would be as follows:

```
(prec> (left "*" "/") (left "+" "-"))
```

The precedence specification can be used along with `$prec` in the grammer to resolve shift-reduce or reduce-reduce conflicts in a grammar. The classical case is the if-then-else construct in C, where a conflict occurs on the input

```
if (expr1) if (expr2) bar(); else baz();
```

The above could be interpreted as

```
if (expr1) { if (expr2) bar(); } else baz();
```

or as

```
if (expr1) { if (expr2) bar(); else baz(); }
```

hence a conflict. The language specification indicates the latter, so the parser should shift. This rule can be specificed in a C parser as follows:

```
(lalr-spec
 ...
```

```
(prec< 'then "else")        ; "then/else" SR-conflict resolution
...
(grammar
 ...
 (selection-statement
  ("if" "(" expression ")" statement ($prec 'then)
   ($$ `(if ,$3 ,$5)))
  ("if" "(" expression ")" statement "else" statement
   ($$ `(if ,$3 ,$5 ,$7)))
   ...
```

It is important to note here that we use a quoted symbol `'then` rather than a string `"then"` as a dummy token. If we would have used `"then"` as the dummy then the lex'er would return the associated token when `"then"` appears in the input and the C declaration

```
int then(int);
```

would produce a syntax error.

Expected Conflicts

There are default rules for handling shift-reduce conflicts. If you can live with these it is possible to inhibit the error messages generated by using the `expect` specifier, which takes as argument the expected number of shift-reduce conflicts:

```
(lalr-spec
 (expect 3)
 ...
```

Grammar

The grammer is a list of production rules. Each production rule take the form

```
(lhs (rhs1 ...) (rhs2 ...) ...)
```

where *lhs* is the left hand side is a non-termianl represented as a Scheme identifier. Each right hand side is a list non-terminals, terminals, actions or proxies, represented by Scheme indentifiers, Scheme constants, `$$`-expressions or proxy expressions, respectively. The terminals can be Scheme strings, character constants or quoted symbols, but not numbers. For example, the following is the production rule for a C99 additive expression:

```
(add-expr
 (mul-expr)
 (add-expr "+" mul-expr ($$ `(add ,$1 ,$3)))
 (add-expr "-" mul-expr ($$ `(sub ,$1 ,$3))))
```

Here `add-expr` and `mul-expr` are non-terminals and `"+"`, `"-"` are terminals and `($$ `(add ,$1 ,$3))` and `($$ `(sub ,$1 ,$3))` are actions. In the actions `$1` refers to the semantic value of the term `add-expr`.

Symbols starting with `$` are reserved. The following symbols have special meaning: All symbols starting with `$` are reserved. Unused reserved symbols will likely not signal an error. The following reserved symbols are in use:

`$prec` TO BE DOCUMENTED

`$error` TO BE DOCUMENTED

`$empty` TO BE DOCUMENTED

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

`$fixed` This is emitted by the lexical analyzer to indicate an unsigned integer.

`$float` This is emitted by the lexical analyzer to indicate an unsigned floating point number.

`$string` This is emitted by the lexical analyzer to indicate a string.

`$code-comm`
This is emitted by the lexical analyzer to indicate a comment starting after code appearing on a line.

`$lone-comm`
This is emitted by the lexical analyzer to indicate a comment starting on a line without preceeding code.

`$$, $$-ref, $$/ref`
These define an action in the right-hand side of a production. They have the forms

```
($$ body)
($$-ref 'rule12)
($$/ref 'rule12 body)
```

The `ref` forms are used to provide references for future use to support other (non-Scheme) languages, where the parser will be equipped to execute reduce-actions by reference (e.g. an associative array).

`$1, $2, ...`
These appear as arguments to user-supplied actions and will appear in the *body* shown above. The variables reference the symantic values of right-hand-side symbols of a production rule. Note that mid-rule actions count here so

```
(lhs (l-expr ($$ (gen-op)) r-expr ($$ (list $2 $1 $3))))
```

generates a list from the return of `(gen-op)` followed by the semantic value associated with `l-expr` and then `r-expr`.

`$?, $*, $+`
These are (experimental) macros used for grammar specification.

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

In addition, the following reserved symbols may appear in output files:

`$chlit` ??? This is emitted by the lexical analyzer to indicate a character constant.

`$start` This is used in the machine specification to indicate the production rule for starting the grammar.

`$end` This is emitted by the lexical analysis to indicate end of input and appears in the machine to catch the end of input.

`$P1, $P2, ...`
Symbols of the form `$P1`, `$P2`,... are as symbols for proxy productions (e.g., for mid-rule actions). For example, the production rule

```
(lhs (ex1 ($$ (gen-x)) ex2 ex3) ($$ (list $1 $2 $3 $4)))
```

will result in the internal p-rules

```
(lhs (ex1 $P1 ex2 ex3) ($$ (list $1 $2 $3 $4)))
($P1 ($empty ($$ (gen-x))))
```

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

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.

2.2 Parsing a Sublanguage of a Specification

Say you have a NYACC specification `cspec` for the C language and you want to generate a machine for parsing C expressions. You can do this using `restart-spec`:

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

2.3 Generating the Machine

`make-lalr-machine` *spec* => *a-list* [Procedure]
Given a specification generated by `lalr-spec` this procecure generates an a-list which contains the data required to implement a parser generated with, for example, `make-lalr-parser`.

The generated a-list includes the following keys:

`pat-v` a vector of parse action procecures

`ref-v` a vector of parse action references (for supporting other languages)

`len-v` a vector of p-rule lengths

`rto-v` a vector of lhs symbols ("reduce to" symbols)

`lhs-v` a vector of left hand side symbols

`rhs-v` a vector of vectors of right hand side symbols

`kis-v` a vector of itemsets

Using Hashed Tables

The lexical analyzer will generate tokens. The parser generates state transitions based on these tokens. When we build a lexical analyzer (via `make-lexer`) we provide a list of strings to detect along with associated tokens to return to the parser. By default the tokens returned are symbols or characters. But these could as well be integers. Also, the parser uses symbols to represent non-terminals, which are also used to trigger state transitions. We could use integers instead of symbols and characters by mapping via a hash table. We will bla bla bla. There are also standard tokens we need to worry about. These are

1. the `$end` marker
2. identifiers (using the symbolic token `$ident`
3. non-negative integers (using the symbolic token `$fixed`)
4. non-negative floats (using the symbolic token `$float`)
5. `$default` => 0

And action

1. positive => shift
2. negative => reduce
3. zero => accept

However, if these are used they should appear in the spec's terminal list. For the hash table we use positive integers for terminals and negative integers for non-terminals. To apply such a hash table we need to:

1. from the spec's list of terminals (aka tokens), generate a list of terminal to integer pairs (and vice versa)
2. from the spec's list of non-terminals generate a list of symbols to integers and vice versa.
3. Go through the parser-action table and convert symbols and characters to integers
4. Go through the XXX list passed to the lexical analyizer and replace symbols and characters with integers.

One issue we need to deal with is separating out the identifier-like terminals (aka keywords) from those that are not identifier-like. I guess this should be done as part of `make-lexer`, by filtering the token list through the ident-reader. NOTE: The parser is hardcoded to assume that the phony token for the default (reduce) action is `'$default` for unhashed machine or `-1` for a hashed machine.

`hashify-machine` *mach* `=>` *mach* [Procedure]
Convert machine to use integers instead of symbols. The match table will change from

```
("abc" . 'abc)
```

to

```
("abc" . 2)
```

and the pax will change from

```
("abc" . (reduce . 1))
```

to

```
("abc" . 2)
```

`machine-hashed?` *mach* `=>` *#t*|*#f* [Procedure]
Indicate if the machine has been hashed.

Compacting Machine Tables

`compact-machine` *mach* [*#:keep 3*] [*#:keepers '()*] `=>` *mach* [Procedure]
A "filter" to compact the parse table. For each state this will replace the most populus set of reductions of the same production rule with a default production. However, reductions triggered by user-specified keepers and the default keepers – `'$error`, `'$end`, `'$lone-comm` and `'$lone-comm` are not counted. The parser will want to treat errors and comments separately so that they can be trapped (e.g., unaccounted comments are skipped).

2.4 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., `'$float`).
- 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.

2.5 Constructing Lexical Analyzers

The `lex` module provides a set of procedures to build lexical analyzers. The approach is to first build a set of *readers* for different types of tokens (e.g., numbers, identifiers, character sequences) and then process input characters (or code points) through the procedures. The signature of most readers is the following:

```
(reader ch) => #f | (type . value)
```

If the reader fails to read a token then `#f` is returned. If the reader reads more characters from input and fails, then it will push back characters. So, the basic structure of a lexical analyzer is

```
(lambda ()
  (let iter ((ch (get-char)))
    (cond
     ((eof-object? ch) '($end . ""))
     ((whitespace-reader ch) (iter (read-char)))
     ((comment-reader ch) (iter (read-char)))
     ((number-reader ch))
     ((keyword-reader ch))
     ((ident-reader ch))
     ...
     (else (error)))))
```

The types of readers used are

ident-reader
: reads an identifier

num-reader
: reads a number

string-reader
: reads a string literal

chlit-reader
: reads a character literal

comm-reader
: reads a comment

comm-skipper
: same as comm-reader

chseq-reader
: a reader for a sequence of characters (e.g., `+=`)

Note that some of our parsers (e.g., the C99 parser) is crafted to keep some comments in the output syntax tree. So comments may be passed to the parser or skipped, hence the "skipper".

The Lex Module does not provide lexical analyzers (lex'ers), but lexical analyzer generator generators. The rationale behind this is as follows. A lexical analyzer may have state (e.g., beginning of line state for languages where newline is not whitespace). In addition, our generator uses a default set of readers, but allows the caller to specify other readers. Or, if the user prefers, lex'ers can be rolled from provided readers. Now we introduce our lex'er generator generator:

`make-lexer-generator` *match-table* [*options*] `=>` *generator* [Procedure]

Returns a lex'er generator from the match table and options. The options are

`#:ident-reader reader`
: Use the provided reader for reading identifiers. The default is a C language ident reader, generated from

```
(make-ident-reader c:if c:ir)
```

`#:num-reader` *reader*
: Use the provided number reader.

`#:string-reader` *reader*
: Use the provided reader for string literals.

`#:chlit-reader` *reader*
: Use the provide charater literal reader. The default is for C. So, for example the letter 'a' is represented as `'a'`.

`#:comm-reader` *reader*
: Use the provided comment reader to pass comments to the parser.

`#:comm-skipper` *reader*
: Use the provided comment reader, but throw the token away. The default for this is `#f`.

`space-chars` *string*
: not a reader but a string containing the whitespace characters (fix this)

```
(define gen-lexer (make-lexer-generator #:ident-reader my-id-rdr))
(with-input-from-file "foo" (parse (gen-lexer)))
```

(Minor note: The *ident-reader* will be used to read ident-like keywords from the match table.)

`make-space-skipper` *chset* `=>` *proc* [Procedure]

This routine will generate a reader to skip whitespace.

`skip-c-space` *ch* `=>` *#f*|*#t* [Procedure]

If `ch` is C whitespace, skip all spaces, then return `#t`, else return `#f`.

`make-ident-reader` *cs-first cs-rest* `=>` *ch* `->` *#f*|*string* [Procedure]

For identifiers, given the char-set for first character and the char-set for following characters, return a return a reader for identifiers. The reader takes a character as input and returns `#f` or `string`. This will generate exception on `#<eof>`.

`read-c-ident` *ch* `=>` *#f*|*string* [Procedure]

If ident pointer at following char, else (if #f) ch still last-read.

`make-ident-like-p` *ident-reader* [Procedure]
Generate a predicate, from a reader, that determines if a string qualifies as an identifier.

`like-c-ident?` *ch* [Procedure]
Determine if a string qualifies as a C identifier.

`make-string-reader` *delim* [Procedure]
Generate a reader that uses `delim` as delimiter for strings. TODO: need to handle matlab-type strings. TODO: need to handle multiple delim's (like python)

`read-oct` *ch* => "*0123*"|*#f* [Procedure]
Read octal number.

`read-hex` *ch* => "*0x7f*"|*#f* [Procedure]
Read octal number.

`read-c-string` *ch* => (*$string* . "*foo*") [Procedure]
Read a C-code string. Output to code is `write` not `display`. Return #f if *ch* is not `"`. This reader does not yet read trigraphs.

`make-chlit-reader` [Procedure]
Generate a reader for character literals. NOT DONE. For C, this reads `'c'` or `'\n'`.

`read-c-chlit` *ch* [Procedure]

```
... 'c' ... => (read-c-chlit #\') => '($ch-lit . #\c)
```

`make-num-reader` => (*proc ch*) => *output* [Procedure]
Generates a procedure to read C numbers where *output* is of the form `#f`, `($fixed . "1")` or `($float . "1.0")` This routine will clean up floats by adding `"0"` before or after dot.

`cnumstr->scm` *C99-str* => *scm-str* [Procedure]
Convert C number-string (e.g, `0x123LL`) to Scheme numbers-string (e.g., `#x123`).

`read-c-num` *ch* => *#f*|*string* [Procedure]
Reader for unsigned numbers as used in C (or close to it).

`make-chseq-reader` *strtab* [Procedure]
Given alist of pairs (string, token) return a function that eats chars until (token . string) is returned or `#f` if no match is found.

`make-comm-reader` *comm-table* [*#:eat-newline #t*] => \ [Procedure]
ch bol -> ('$code-comm "..")|('$lone-comm "..")|#f comm-table is list of cons for (start . end) comment. e.g. ("–" . "\n") ("/*" . "*/") test with "/* hello **/" If `eat-newline` is specified as true then for read comments ending with a newline a newline swallowed with the comment. Note: assumes backslash is never part of the end

Rolling Your Own Lex'er

The following routines are provided for rolling your own lexical analyzer generator. An example is provided in the file `examples/nyacc/lang/matlab`.

`filter-mt` *p? al => al* [Procedure]
Filter match-table based on cars of al.

`remove-mt` *p? al => al* [Procedure]
Remove match-table based on cars of al.

`map-mt` *f al => al* [Procedure]
Map cars of al.

`eval-reader` *reader string => result* [Procedure]
For test and debug, this procedure will evaluate a reader on a string. A reader is a procedure that accepts a single character argument intended to match a specific character sequence. A reader will read more characters by evaluating `read-char` until it matches or fails. If it fails, it will pushback all characters read via `read-char` and return `#f`. If it succeeds the input pointer will be at the position following the last matched character.

2.6 The Parser-Lex'er Interface

Sometimes LALR(1) parsers must be equipped with methods to parse non-context free grammars. With respect to typenames, C is not context free. Consider the following example.

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

The lexical analyzer must identify the first occurance of `foo_t` as an identifier and the second occurance of `foo_t` as a typename. This can be accomplished by keeping a list of typenames in the parent environment to the parser and lexical analyzer. In the parser, when the first statement is parsed, an action could declare `foo_t` to now be a typename. In the lexical analyzer, as tokens that look like identifers are parsed they are checked against the list of typenames and if a match is found, `'typename` is returned, otherwise `$ident` is returned.

Another example of this handshaking is used in the JavaScript parser. The language allows newline as a statement terminator, but it must be prevented in certain places, for example between `++` and an expression in the post-increment operator. We handle this using a mid-rule action to tell the lexer to skip newline if that is the next token.

```
(LeftHandSideExpression ($$ (NSI)) "++" ($$ `(post-inc $1)))
```

The procedure `NSI` in the lex'er is as follows:

```
(define (NSI) ;; no semicolon insertion
  (fluid-set! *insert-semi* #f))
```

and the newline reader in the lex'er acts as follows:

```
...
((eqv? ch #\newline)
```

```
        (if (fluid-ref *insert-semi*)
            (cons semicolon ";")
            (iter (read-char))))
       ...
```

2.7 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")
```

This saves the variable `act-v` to the file `calc1-act.scm` and the following variables to the file `calc1-tab.scm`:

`len-v`

`pat-v`

`rto-v`

`mtab`

The variable `act-v` is a vector of procedures associated with each of the production rules, to be executed as the associated production ruled is reduced in parsing.

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 distribution.

2.8 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 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))))
```

`machine-compacted?` *mach* => *#t* | *#f* [Procedure]

Indicate if the machine has been compacted.

2.9 Exporting Parsers

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.

2.10 Debugging

The provided parsers are able to generate debugging information.

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 $P1 =>
4 stmt => "\n"
5 stmt => expr "\n"
6 expr => expr "+" expr
7 expr => expr "-" expr
8 expr => expr "*" expr
9 expr => expr "/" expr
10 expr => "*" '$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

In this chapter we present procedures for generating syntax trees in a uniform form. This format, based on SXML, may use more cons cells than other formats but upon use you will see that it is easy to produce code in the parser, and one can use all the processing tools that have been written for SXML (e.g., `(sxml match)` `(sxml fold)`.

The syntax of SXML trees is simple:

```
expr => (tag item ...) | (tag (@ attr ...) item ...)
item => string | expr
attr => (tag . string)
```

where tag names cannot contain the characters

```
( ) " ’ ‘ , ; ? > < [ ] ~ = ! # $ % & * + / \ @ ^ | { }
```

and cannot begin with `-`, `.` or a numeric digit.

For example our Javascript parser given the input

```
function foo(x, y) {
  return x + y;
}
```

will produce the following syntax tree:

```
(Program
  (SourceElements
    (FunctionDeclaration
      (Identifier "foo")
      (FormalParameterList
        (Identifier "x")
        (Identifier "y"))
      (SourceElements
        (EmptyStatement)
        (ReturnStatement
          (add (PrimaryExpression (Identifier "x"))
               (PrimaryExpression (Identifier "y"))))
        (EmptyStatement)))
    (EmptyStatement)))
```

And by the way, put through our tree-il compiler, which uses `foldts*-values` from the module `(sxml fold)` we get

```
(begin
  (define foo
    (lambda ((name . foo))
      (lambda-case
        ((() #f @args #f () (JS~5575))
         (prompt
           (const return)
           (begin
             (abort (const return)
                    ((apply (@@ (nyacc lang javascript jslib) JS:+)
```

```
                        (apply (toplevel list-ref)
                               (lexical @args JS~5575)
                               (const 0))
                        (apply (toplevel list-ref)
                               (lexical @args JS~5575)
                               (const 1))))
                (const ())))
          (lambda-case
            (((tag val) #f #f #f () (JS~5576 JS~5577))
             (lexical val JS~5577)))))))))
```

3.1 Tagged Lists

Paring actions in NYACC can use tagged-lists from the module `(nyacc lang util)` to help build SXML trees efficiently. Building a statement list for a program might go as follows:

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

The sequence of calls to the `tl-` routines goes as follows:

`(make-tl 'stmt-list)`
: Generate a tagged list with tag `'stmt-list`.

`(tl-append $1 $2)`
: Append item `$2` (not a list) to the tagged-list `$1`.

`(tl->list $1)`
: Convert the tagged-list `$1` to a list. It will be of the form

  ```
  '(stmt-list (stmt ...) (stmt ...) ...)
  ```

  The first element of the list will be the tag `'stmt-list`. If attributes were added, the list of attributes will be the second element of the list.

The following procedures are provided by the module `(nyacc lang util)`:

`make-tl` *tag* [*item item ...*] [Procedure]
: Create a tagged-list structure for tag *tag*. Any number of additional items can be added.

`tl->list` *tl* [Procedure]
: Convert a tagged list structure to a list. This collects added attributes and puts them right after the (leading) tag, resulting in something like

  ```
  (<tag> (@ <attr>) <item> ...)
  ```

`tl-insert` *tl item* [Procedure]
: Insert item at front of tagged list (but after tag).

`tl-append` *tl item ...* [Procedure]
: Append items at end of tagged list.

`tl-extend` *tl item-l* [Procedure]
Extend with a list of items.

`tl-extend!` *tl item-l* [Procedure]
Extend with a list of items. Uses `set-cdr!`.

`tl+attr` *tl key val*) [Procedure]
Add an attribute to a tagged list. Return the tl.

```
(tl+attr tl 'type "int")
```

`tl-merge` *tl tl1* [Procedure]
Merge guts of phony-tl `tl1` into `tl`.

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)
```

`sx-ref` *sx ix => item* [Procedure]
Reference the `ix`-th element of the list, not counting the optional attributes item. If the list is shorter than the index, return `#f`.

```
(sx-ref '(abc "def") 1) => "def"
(sx-ref '(abc (@ (foo "1")) "def") 1) => "def"
```

`sx-tag` *sx => tag* [Procedure]
Return the tag for a tree

`sx-cons*` *tag* (*attr*|*#f*)? ... *=> sx* [Procedure]
`sx-list` *tag* (*attr*|*#f*)? ... *=> sx* [Procedure]
Generate the tag and the attr list if it exists. Note that

`sx-tail` *sx* [*ix*] *=>* (*list*) [Procedure]
Return the ix-th tail starting after the tag and attribut list, where *ix* must be positive. For example,

```
(sx-tail '(tag (@ (abc . "123")) (foo) (bar)) 1) => ((foo) (bar))
```

Without second argument *ix* is 1.

`sx-has-attr?` *sx* [Procedure]
A predicate to determine if *sx* has attributes.

`sx-attr` *sx => '*(@ ...)|*#f* [Procedure]

```
(sx-attr '(abc (@ (foo "1")) def) 1) => '(@ (foo "1"))
```

should change this to

```
(sx-attr sx) => '((a . 1) (b . 2) ...)
```

`sx-attr-ref` *sx key => val* [Procedure]
Return an attribute value given the key, or `#f`.

`sx-set-attr!` *sx key val* [Procedure]
Set attribute for sx. If no attributes exist, if key does not exist, add it, if it does exist, replace it.

`sx-set-attr*` *sx key val* [*key val* [*key ...*]] [Procedure]
Generate sx with added or changed attributes.

`sx+attr*` *sx key val* [*key val* [*. . .*]] `=>` *sx* [Procedure]
Add key-val pairs. *key* must be a symbol and *val* must be a string. Return a new *sx*.

`sx-find` *tag sx* `=>` ((*tag ...*) (*tag ...*)) [Procedure]
Find the first matching element (in the first level).

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

4 Coding to the Compiler Tower

```
(define-module (language javascript spec)
  #:export (javascript)
  #:use-module (nyacc lang javascript separser)
  #:use-module (nyacc lang javascript compile-tree-il)
  #:use-module (nyacc lang javascript pprint)
  #:use-module (system base language))

(define-language javascript
  #:title       "javascript"
  #:reader      js-reader
  #:compilers   `((tree-il . ,compile-tree-il))
  #:printer     pretty-print-js)
(define-module (nyacc lang javascript compile-tree-il)
  #:export (compile-tree-il)
  #:use-module (nyacc lang javascript jslib)
  #:use-module ((sxml match) #:select (sxml-match))
  #:use-module ((sxml fold) #:select (foldts*-values))
  #:use-module ((srfi srfi-1) #:select (fold))
  #:use-module (language tree-il))

...

(define (compile-tree-il exp env opts)
  (let* ((xrep (js-sxml->tree-il-ext exp env opts))
         (code (parse-tree-il xrep)))
    (values code env env)))
```

4.1 Pretty Print

`make-pp-formatter` [*port*] [*#:per-line-prefix* `""`] => *fmtr* [Procedure]

```
(fmtr 'push) ;; push indent level
(fmtr 'pop)  ;; pop indent level
(fmtr "fmt" arg1 arg2 ...)
```

5 Administrative Notes

5.1 Installation

Installation instructions are included in the top-level file `INSTALL` of the source distribution. If you have an installed Guile then the basic steps are

```
$ ./configure
$ make install
```

Help with alternative usage is available with

```
$ ./configure --help
```

If Guile is not installed it is possible to install source only:

```
$ ./configure --site-scm-dir=/path/to/dest --site-scm-go-dir=/dummy
$ make install-srcs
```

5.2 Reporting Bugs

Please report bugs to the support site '`https://savannah.nongnu.org/support/?group=nyacc`' or to the Guile user's mailing list `guile-user@gnu.org`.

5.3 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`.

6 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 ...))
```

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

13 add ($abort) and ($accept)

19 add a location stack to the parser/lexer

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

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.