NYACC Hacker's Guide

rough and incomplete

Matt Wette

1 The Introduction

This is a manual for ...

2 The Algorithms

The implementation is based on algorithms laid out in the Dragon Book. See ⟨undefined⟩ [References], page ⟨undefined⟩. In the application of NYACC one writes out a (context-free) grammer using Backus-Naur form. See the example in the user's manual. In addition to the grammar the start symbol must be provided. Optional inputs include specifiers for precedence and associativity. See the user's manual for further details.

2.1 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 four-tuple

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 production rule along with a position within the RHS. It is represented by the notation

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

where the dot `.` represents the position in the RHS. One may see that the item can be viewed as a state of parsing the production rule left to right. In fact, each state in the parser can be represented by a set of items. 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, or -1 if the position is at the end (past the last symbol in the RHS).

Given a production rule, a *lookahead* is a terminal that can appear after the production in the language. 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}
```

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 of the parse, the start state, which includes the item `$start => .S`, will have all items with dots at position zero. For the remainder of parsing (after we have acted on the first input token, or terminal) at least one effective item will have the dot at position greater than zero. For those itemsets the subset of items with non-zero dot positions and at least one RHS symbol will be called the *kernel itemsets*. For the initial state of the parser the kernel itemset is `{$start => .S}`.

Before we continue we introduce the function `first`. It is an important function used by NYACC in generating a parser. 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.

2.2 Building the Automaton

In NYACC the canonical grammer will always include the following internally generated production for specified start symbol '`S`':

```
$start => S
```

This production represents the acceptance of the input. The symbol `$start` has only this single production, and this production always has index zero. (Note that Bison includes the endmarker, which we call `$end`, in this production. This results in Bison parsers having one additional state with respect to NYACC parsers.)

Now consider parsing an input left to right. The initial state, or itemset, will include the la-item consisting of the cananical 0-indexed production, at position zero, with the lookahead set consisting of `{$end}`:

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

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

```
S => ap
S => Bq
```

and assume the non-terminal `B` has the single production

```
B => cr
```

Then the initial la-itemset, or initial state of the automaton, is as follows:

```
0: $start => .S, {$end}
   S => .ap, {$end,...}
   S => .Bq, {$end,...}
   B => .cr, {$end,...}
```

where the `...` represents unknown other lookaheads. At this point we don't know the entire set of possible lookaheads because `S` may appear in right-hand sides of other production rules.

Now consider, in state *i*, an item as follows

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

Say we associate this with the "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.

3 Translating SXML

Many of the example parsers generate SXML. Here we describe the process of translating to something else.

3.1 Using `foldts*-values`

In the modules `(sxml fold)` is the routine `foldts*-values`. This has been used to convert the SXML output of the example javascript parser into Tree-IL. The signature to this is

```
foldts*-values fdown fup fhere tree seed1 ... => tree seed1 ...
```

where any number of seeds can be passed. We choose to use two: `seed` and `dict`. The arguments we pass to `foldts*-values` are

`fdown tree seed dict => tree seed dict`
: The down processor.

`fup tree seed dict kseed kdict => seed dict`
: The up processor.

`fhere tree seed dict => seed dict`
: The here processor.

4 Habits

This chapter explains some of my programming habits. I hope it helps to relieve the head-scratching.

4.1 Modifying Compound Data Types

In general I do not use `set-car!` or `set-cdr!`. I do use `vector-set!` and `hashq-set!`

4.2 A-Lists Versus Hash Tables

In general I prefer a-lists over hash tables in Scheme because a-lists are Scheme-like. To update an entry in an alist I will just paste the new entry on the front. For example,

```
(let ((al '((foo . 1) (bar . 2) (baz . 3))))
  ...
  (acons 'bar 99 al))
```

If the values of the a-list are lists and I want to add something to the list I just use cons, as in the following:

```
(let ((al '((foo 1) (bar 2) (baz 3))))
  ...
  (acons 'bar (cons 99 (assq-ref al 'bar)) al)
```

This modification costs just two cons cells.

4.3 Iteration

For iteration I usually use *named-let* and often in concert with *cond*. The order of variable declarations in my named-let are the result variable, followed by iteration variables in order of slowest to fastest modification. In the cond I usually evaluate in the order fastest to slowest modification. Consider the following C code fragment:

```
res = 0;
for (i = 0; i < ni, i++) {
  res += 100*i;
  for (j = 0; j < nj, j++) {
    res += 10*j;
    for (k = 0; k < nk, k++) {
      res += k;
    }
  }
}
```

In Scheme, I would express this as

```
(let iter ((res 0) (i 0) (j 0) (k 0))
  (cond
   ((< k nk) (iter (+ res k) i j (1+ k)))
   ((< j nj) (iter (+ res (* 10 j)) i (1+ j) 0))
   ((< i ni) (iter (+ res (* 100 i)) (1+ i) 0 0))
   (else res)))
```

4.4 The Free Documentation License

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