booze-tools

Ian Kjos

Aug 08, 2023

CONTENTS:

Booze-Tools aims to be the ultimate Swiss-Army Knife of language-processing tasks. It does quite a bit so far, and has some features not found in other such tools. It happens to be written in Python.

Project Maturity Level:

The deep algorithmic stuff works like a champ, but there are some rough edges. Some APIs may change, but the general concept to keep the grammar definition independent of the target-language is not going anywhere. (In concept, this should allow a compiler, a pretty-printer, and an IDE to all run from the same grammar definition.) It's used in a few different projects and generates feedback.

CHAPTER

ONE

INTRODUCTION

Booze-Tools aims to be the ultimate Swiss-Army Knife of language-processing tasks. It does quite a bit so far. and has some features not found in other such tools. It happens to be written in Python.

1.1 Documentation Migration in Progress

At this very moment, most of what documentation there is lives on the wiki page. In time, more should migrate over.

1.2 Why and Wherefore?

Yes, there are other language tools out there, but this is a journey and I plan to enjoy it.

The original pipe-dream was for this library to become the complete programming-language development workbench of choice, but written in a high-level language (not C or C++, like GCC or LLVM). Therefore, the Booze-Tools will grow whatever additional bits make that happen. It does not stop with lexical analysis and context-free parsing.

The next goal was to explore solutions to annoying features of other popular language-processing subsystems. That's why, for instance, `macroparse` supports macro-expansion of context-free production rules. (This turns out to be more expressive than EBNF.)

I wanted a single system that could *in principle* be used to generate scanners/parsers for other host-languages than just Python. That's why there's a JSON output format and a system of *messages* rather than embedded code: current scripting languages can read JSON directly, and independent tools can turn it into (for instance) C code. By implementing a small driver, you can re-use a language definition that was originally made for some other purpose. This feature may keep an ecosystem of tools in sync with each other.

A once-and-future goal is that the project source code should end up clear and instructive. Therefore, there are heavy comments in parts that deal in the classical algorithms and data structures peculiar to the field of compilers. If you see something unclear, please speak up. Opacity is a bug.

To that same end, I may eventually add on some alternative constructions, as (perhaps) for SLR and LR(0) as ends in themselves, with heavy commentary. At one point I was digging deep into the issues involved with ambiguous grammars, which is why the `parsing.general` submodule exists. The experience contributed greatly to the error-recovery mechanisms in the main parsing algorithm.

Finally, it seems that many data analysis tasks are greatly simplified if you can treat them as *parsing with a bit extra*. I want to make it absolutely convenient to develop and deploy such intermixed applications.

CHAPTER

TWO

INVOKING THE *BOOZE-TOOLS* METACOMPILER

Let's say you looked at the GitHub repository at the examples, and decided the concept was pretty cool. How shall we proceed?

All of the examples are coded to build their grammar into a parser at run-time. That's alright for a demonstration, but how would you deal with this in a real project?

You could choose to copy a line of code, but actually there are a number of other handy features available on the command-line.

2.1 Extremely Short Version

On the command line:

```
D:\GitHub\booze-tools>py -m boozetools example\pascal.md
Wrote automaton in JSON format to:
        example\pascal.automaton
```

Then later, in Python code, something like:

```python
import json
from boozetools.macroparse import runtime

tables = json.load('example/pascal.automaton')

class MyParser(runtime.TypicalApplication):
    def __init__(self):
        super().__init__(tables)

    def scan_this(self, yy):
        yy.token('this', yy.match())

    def parse_that(self, left, middle, right):
        return That(left, middle, right)

syntax_tree = MyParser().parse("some big long text")
```

2.2 A Bit More Detail

You have lots of options about how you invoke this:

```
D:\GitHub\booze-tools>py -m boozetools -h
usage: py -m boozetools [-h] [-f] [-o OUTPUT] [-i] [--pretty] [--csv] [--dev] [--dot] [-
↪m {LALR,CLR,LR1}] [-v]
                        source_path

Compile a macroparse grammar/scanner definition from a markdown document into a set of␣
↪parsing and scanning tables in
JSON format. The resulting tables are suitable for use with the included runtime modules.
↪ Pypi:
https://pypi.org/project/booze-tools/ GitHub: https://github.com/kjosib/booze-tools/wiki
ReadTheDocs: https://boozetools.readthedocs.io/en/latest/

positional arguments:
  source_path           path to input file

optional arguments:
  -h, --help            show this help message and exit
  -f, --force           allow to write over existing file
  -o OUTPUT, --output OUTPUT
                        path to output file
  -i, --indent          indent the JSON output for easier reading.
  --pretty              Display uncompressed tables in attractive grid format on STDOUT.
  --csv                 Generate CSV versions of uncompressed tables, suitable for␣
↪inspection.
  --dev                 Operate in "development mode" -- which changes from time to time.
  --dot                 Create a .dot file for visualizing the parser via the Graphviz␣
↪package.
  -m {LALR,CLR,LR1}, --method {LALR,CLR,LR1}
                        Which parser table construction method to use.
  -v, --verbose         Squawk, mainly about the table compression stats.
```

2.3 Error Handling?

There is support for that.

Error Rules:
You can write error production-rules using the metatoken `$error$`. The machinery surrounding this takes pains to do well. It may not be the fastest concept, but it's smarter than the average parser.

On-Error Call-Backs:
For everything else, there are error call-backs. If you look in `boozetools/macroparse/runtime.py` (here) you'll find `class AbstractTypical` which defines default behavior for situations in which (a) the parser's error-rule mechanism was unable to resolve, or (b) a stuck scanner.

CHAPTER

THREE

FORMAT OF TABLES

Table of Contents

The .automaton files could be of interest to someone who wants to:

- implement alternative rule-following strategies.
- write specialized post-processing tools.
- write a driver in another host language than Python.
- be curious for curiosity's own sake.

What's here describes the most current version of these files.

3.1 General Structure

An `.automaton` file is a JSON (JavaScript Object Notation) object with keys as follows:

- description: A string. Currently always `"MacroParse Automaton"`, but in principle could be taken from the grammar definition.
- parser: Another JSON object described below.
- scanner: Another JSON object described below.
- source: The base-name of the file from which the grammar definition came.
- version: An array of three small integers meant to obey the conventions of semantic versioning.

3.2 Scanner Table

3.2.1 Overview

Booze-Tools scanners are based on deterministic finite state machines. To scan a lexeme, it begins in the initial state for its current scan condition, then follows a delta function of (state, character) until no further progress is possible. At that point, the scanner follows the rule corresponding to the leftmost-longest-match. Rather, the states are annotated such that the leftmost-longest-match is selected in this process.

The source code for the algorithm is pretty short and illustrative. Start with the function `scan_one_raw_lexeme` in module `boozetools.scanning.recognition`.

3.2.2 Top-Level Fields

- action: a table of information telling the scan algorithm what to do once it's found a match.
- alphabet: encodes a function from character code-point to character class.
- dfa: encodes the deterministic finite state automaton which the scanner follows.

3.2.3 Fields of the DFA

- delta: encodes the transition function from <*state, a character class*> to <*subsequent state*>.
- final: a list of accepting states.
- rule: a corresponding list of rule numbers as accepted by those states.
- initial: a mapping from scan-condition string (such as `"INITIAL"`) to a pair initial states.

The exact form of the `delta` table is based on a lot of experimentation to do a good job compacting otherwise-large tables. Probably a future version will allow to build a much simpler format for tables that are inherently fairly small to begin with.

3.3 Parser Table

Booze-Tools parsers are based on pushdown automata. A simplified version of the algorithm is the function `trial_parse` in the module `boozetools.parsing.shift_reduce`. That version handles neither semantic values nor error recovery. When you feel ready for the rest, check out the `parse` function in the same file.

3.3.1 Top-Level Fields

action
Tells the parser how to respond to a terminal symbol, and when the right-hand side of a production rule is recognized.

breadcrumbs
Used in error reporting; maps states to the symbols that reach those states.

goto
Tells how to respond once a non-terminal symbol has been synthesized.

initial
Tells where to start for each supported language.

nonterminals
A list of the non-terminal symbols used in defining the grammar. This list is useful principally for error-reporting.

rule
Tells how to synthesize a non-terminal symbol from a sequence of symbols from the stack. These are described in greater detail below.

terminals
A list of the string representations of the terminals from the grammar. Useful in error reporting, but you might use it to prepare a reserved-word table.

3.3.2 Encoding of Rules

The format of the parser's `rule` object is probably sub-optimal, but is also:

constructor

List of distinct possible messages from the ends of production rules.

`null` means to just create a tuple of the (non-void) right-hand-side symbols.

Strings that begin with `:` are considered *mid-rule actions*.

All other strings are the names of ordinary *constructor* messages.

line_number
List of the grammar definition source line number for each rule.

rules
List of 4-tuples for each right-hand side.

The 4-tuples for a parser rule are:

Non-terminal index.
This can index into the `nonterminals` list, mentioned earlier.

Length of the rule's right-hand side.

This number of symbols get popped before pushing the non-terminal symbol.

Zero is a valid size: Epsilon rules and mid-rule actions both use it.

Constructor index:

If negative: the stack offset of the semantic value.

If zero or positive: an index into the `constructor` list, mentioned earlier.

Capture list.
This list of integer offsets from top-of-stack (before any popping) describe where to find the arguments for the constructor. In case of a bracketing rule (negative constructor index), the capture-list is meaningless.

Note: Note that all stack-offsets are negative, with -1 being the top-of-stack.

CHAPTER

FOUR

THE ALLEGEDLY MINIMAL LR(1)

Knuth described a simple and correct way to generate an LR(1) parse table, or indeed an LR(k) table for any k. You just do a subset construction on parse-items augmented with a follower-terminal (or k of them). Well and good, but he wrote this down in the 1960s when *computer* was still an honorable profession and the electronic ones were in no danger of carrying out Knuth's algorithm for nontrivial grammars.

Shortly afterwards, the LALR method was discovered. LALR(1) is a clever extension of LR(0) to resolve many shift-reduce conflicts by working out follower-sets. By itself LALR is still a bit clunky, but with a dose of operator-precedence and associativity declarations to resolve remaining ambiguities, the resulting systems were generally quite pleasant. For decades, the pragmatic language hacker relied on LALR because it was fast, acceptably-accurate most-of-the-time, and produced compact tables.

But LALR is not perfect. A cottage industry sprang up for debugging grammars subjected to the LALR treatment.

Then in 2010 someone published the IELR(1) algorithm, which made it practical to use full LR(1) grammars augmented with the same precedence and associativity declarations. There's only one problem: IELR is downright painful to try to understand.

4.1 What "Minimal" even means

Smallest feasible number of states consistent with reasonable generator performance.

I seem to recall reading somewhere that optimizing LR parse tables is hard, in the sense that it's difficult to prove there's not some alternative table with fewer states that recognizes the same language.

Therefore, we shall have to be content with an argument that, if such a thing does indeed exist, then it is infeasible to find.

4.2 My General Concept

My approach to building LR(1) tables without the size blow-up associated with canonical (Knuth-style) LR(1) stems from a simple observation:

- In practice with LALR, conflicts that *would* be resolved in LR(1) are generally confined to small parts of a grammar and relatively few parse states.
- Furthermore, with typical grammars most parse-states have no conflicts even in the LR(0) view.
- Even with LR(1)-style parse-items, there are *equivalence-classes* of terminals with similar behavior.

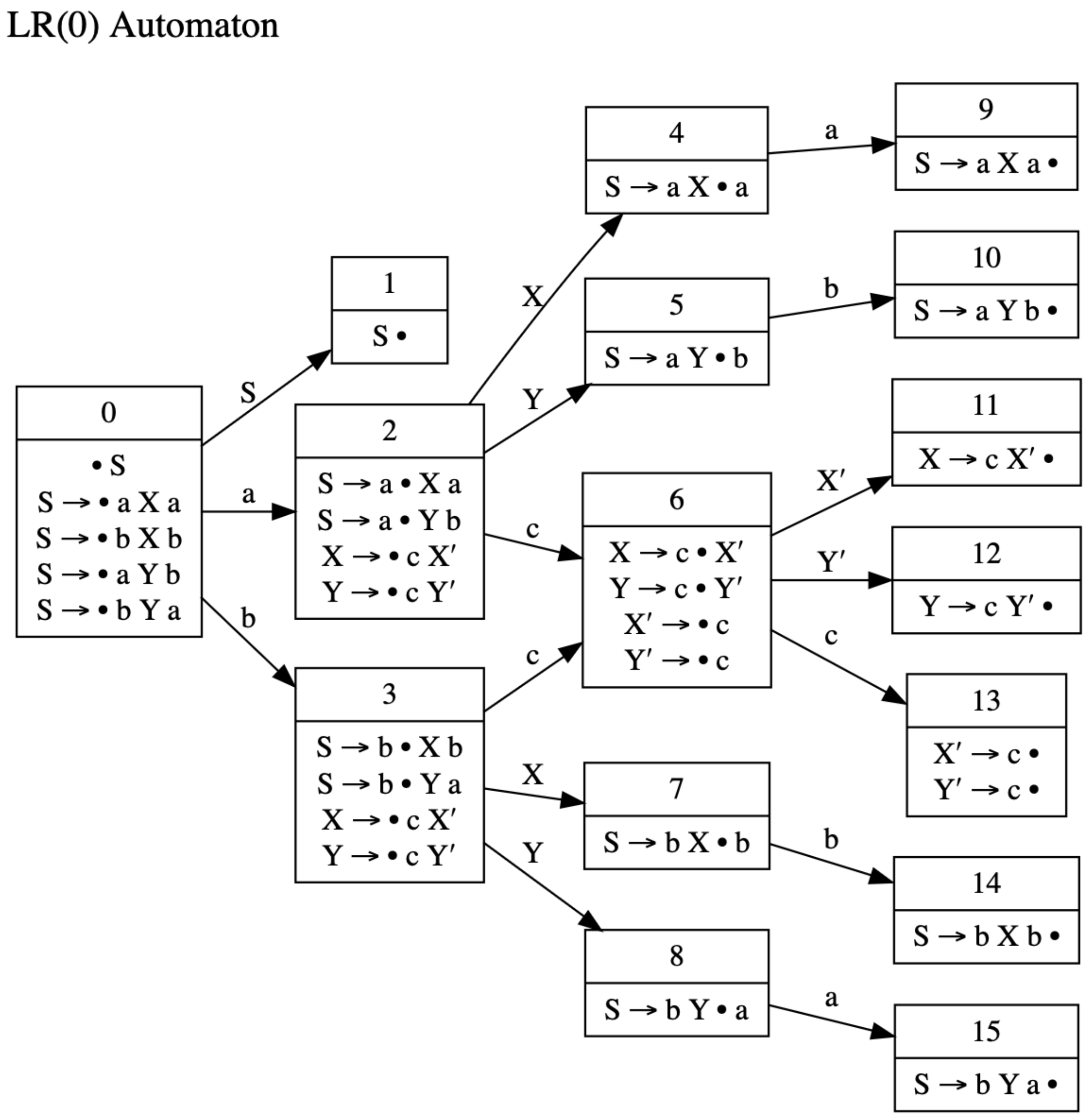
If we could somehow perform LR(0) everywhere that it was good enough, and switch to a more powerful method for those parts of the grammar where strictly necessary, then we should see a nice compromise of speed and size while achieving the full recognition power of the stronger method.

4.3 Case in Point

Consider this grammar:

```
S -> a X a | b X b | a Y b | b Y a
X -> c X'
Y -> c Y'
X' -> c
Y' -> c
```

It has the following LR(0) automaton:



Note: Booze-Tools does not currently support symbols named like `X'` or `Y'` but this is how the example came to me, graphics and all. So this is what I'll run with.

4.4 A Tainted Approach

Have a look at state 13: It's conflicted. It happens to be a reduce/reduce conflict, but this is beside the point.

It happens *also* to be conflicted on a particular set of LR(0) *parse items*. Specifically, those of `X' -> c.` and `Y' -> c.`

With a moment's thought, we can trace back through the table and realize that the *head parse-items* for these conflicts are introduced in LR(0) state 6.

Head Parse-Item
: A parse-item where the dot is in the leftmost position.

So we can call state 6 the **head state** for this conflict. In general there can be more than one head-state for a given conflict, because LR(0) states can have multiple predecessors.

So let us say that symbols `X'` and `Y'` are now officially tainted *specifically in the context of state 6.* Somewhere during the process of elaborating from core-item sets to item-closure, we ran across the need for these particular non-terminals. In fact it's easy to determine if you have the entire item-closure handy: Any item where the dot is just before a tainted symbol (within the same LR(0) state) is also tainted.

Transparent Parse Item:
: Parse items for which the portion of the rule after the dot can possibly produce the empty string. (Transparently, a state with a transparent parse-item has a reduction, and conversely.)

If the successor-item to a parse-item tainted in state Q is transparent, then the head of that parse-item (looking back from state Q) acquires the same taint.

If we take these rules to their logical conclusion (i.e. least fixed-point) then in time we determine that the core-items of states 2 and 3 are all tainted. However, the contagion spreads no further, because the successors to these items are *not* transparent.

4.5 Another Round

We have no further use for the idea of a tainted *symbol* but the tainted *head parse-items* associated with LR(0) states are of particular interest.

Let us have another go at the LR-style subset construction. But this time, we have a special rule:

When considering a core item-set for a parse state, we must first determine the LR(0) *iso-state* and from there the set of tainted head parse-items.

Iso-State
: Consider the set of *iso-items* to the core-items in an LR(1) state. This is guaranteed to correspond with some particular LR(0) state's core item set.

Iso-Item
: Given a parse-item in this round may have a look-ahead token, the Iso-Item is that but without the look-ahead, if any.

Now, the rule is that if an LR(0) head-item in the iso-state is tainted, then we have to construct a set of LR(1)-style head parse-items with a token of look-ahead for this LR(1) state. But if the corresponding head-item in the corresponding LR(0) state is *not* tainted, then evidently this particular head-item does not need the extra resolving power that LR(1)-style items provide.

Finally, we may come to states that are still conflicted. That's OK: We delegate these to the precedence-and-associativity doodad in the usual manner.

4.6 Working Out the Example

4.6.1 Forward-Correctness

Let us assume for the moment that the tainting mechanism described above actually does the right thing. (It needs a proof, but this section should clarify in detail what the proof-obligation exactly is.)

We end up with LR(1) states 2 and 3 like:

```
State 2:
---------
S  ->  a . X a
S  ->  a . Y b
X  ->  . c X' / a
Y  ->  . c Y' / b

State 3:
---------
S  ->  b . X b
S  ->  b . Y a
X  ->  . c X' / b
Y  ->  . c Y' / a
```

Note: I use the / (slash) symbol to set off the look-ahead associated with a parse-item, consistent with Grune and Jacobs.

And clearly this splits LR(0) state 6 into two LR(1) states:

```
State 6a:
----------
X  ->  c . X' / a
Y  ->  c . Y' / b
X' ->  . c / a
Y' ->  . c / b

State 6b:
----------
X  ->  c . X' / b
Y  ->  c . Y' / a
X' ->  . c / b
Y' ->  . c / a
```

It should be clear that if we continue to proceed in this manner, then we end up with a *sufficient* portion of the handle-finding automaton built according to LR(1) principles, so that the resulting parser can do no wrong.

4.6.2 An Interesting Property

LR(1) parsing decides to *shift* or *reduce* based solely on a sequence of terminal symbols. But *goto* actions, for *non*-terminal symbols, do not consider look-ahead at all.

So, let's consider the story around LR(0) states 11 and 12. These states are adequate reducing states in the LR(0) graph. What should become of them in LR(1)?

If we proceed as described above, sooner or later we end up with:

```
State 11a:
-----------
X  ->  x X' . / a

State 11b:
-----------
X  ->  x X' . / b
```

And similar for state 12.

Because the LR(0) iso-state is both adequate and reducing, a parser need not consider look-ahead tokens in this state. Such a parser eagerly reduces out of this kind of state before seeing the look-ahead token. This is desirable for good interactive properties and smaller tables. Eventually, an error token lurking in the input stream will get caught in some less-special state.

So, *an adequate-reducing LR(0)-state need not be split.* If an LR(1) core's iso-state has this property, then avoid creating redundant LR(1) states.

There is probably a way to build on this observation to find a larger class of states – or perhaps parse-items – where it becomes safe to drop the look-ahead. But that's not today's problem.

4.7 Sharpening the Pencil

4.7.1 Big Tables?

The classic argument against LR(1) is that it yields huge tables when given practical grammars. And why is that? We know that every LR(1) state has a corresponding LR(0) state through the *morphism* of dropping the look-ahead tokens out of the core item-sets. So the only viable explanation is that we get lots of distinctive sub-sets where fewer would do the trick.

The method I've described so far should presumably make a dent: Practical grammars may not generally be LR(0), but they probably do have major portions which nevertheless are.

I could propose a refinement on the above technique: Suppose we go ahead and use LALR where it's a non-problem, and then introduce the transitive-closure tainting thing specific to the terminal symbols involved in a particular LALR conflict.

This would change the rules slightly: Given a tainted head-iso-item, one must consider the subset of followers actively engaged in the conflict, along with a stand-in for "everything-else" that LALR got right in the first place.

Note: This was the approach taken in the first version of the minimal-LR(1) code. However, I suspect the tainting logic went wrong when I fixed the LALR mode to be *actually* LALR rather than NQ-LALR, because the relevant code was shared and perhaps too closely so. Either that, or it was insufficient to begin with.

I'm not sure this intermediate step is necessarily worthwhile. LALR does a lot of work on subsets of tokens, and I suspect it's just about the same work-factor if you optimize the representation as I explain in the next section.

In any case, my central conjecture is that reconsidering only the necessary parts of the grammar in LR(1) mode effectively solves the problem of large tables.

4.7.2 Slow Table Generation?

For what I do, I don't see a speed problem. But in any case, let's think it through.

Probably the main driver for speed is the work involved in constructing states. Fewer states generated in the first place might help, and that is a key notion here.

One major factor in that work might be the representation of sets of LR(1) parse-items. In the classic description of canonical LR(1), we talk of constructing and managing a new parse-item for each combination of dotted-rule and relevant look-ahead token. But why manifest that *idea* literally as a distinct scrap of data for each *theoretical* parse-item that propagates through the works?

Suppose we treat subsets of look-ahead tokens as one thing, and the dotted-rule as another. Then we can trivially "shift" an entire set of parse-items in one constant-time operation. I'm assuming you can pass sets around by reference, and that sets have a pre-calculated hashcode rather than re-computing it every time. (If not, then intern the distinct sets.)

4.7.3 Renaming Rules

Rules of the form `A -> B` merit special consideration. In principle you can treat them the same as any other rule, but they do contribute space and time to a parser in that case. If there is no associated semantic action, then it is usually possible to rejigger the parse table to bypass some pointless work. In the process, corresponding parts of the table become unreachable.

Details of this particular optimization are elsewhere. However, it's separate issue from the minimal-LR(1) algorithm. You can apply it regardless of table-generation method.

4.8 Tying Up Loose Ends

At this point, I think it's clear what I need to prove for correctness:

CLAIM:
The set of "tainted" parse-items in the LR(0) automaton are necessary and sufficient to make this monstrosity work.

Trivially, you could taint everything and you'd get canonical LR(1). Somewhere there's a line. I'll take "necessary" on faith for a couple paragraphs, because a miss there still yields a correct parse table, even if a larger one. The "sufficient" part is more interesting. I approach the proof by proving the converse: untainted items have some valid reason why they're fine.

I clarify this below, but mainly:

1. If a parse-item's end-state is LR(0)-adequate, there are no problems.
2. If a parse-item's end-state is not adequate, then its head-item is created with proper relevant look-ahead tokens.
3. If the head-item has a set of look-ahead tokens, these carry through to the item's LR(1) end-state.

CLAIM:
It's safe to use a precedence-and-associativity strategy to disambiguate shift-reduce conflicts. Such a system would do *the right thing* in all circumstances.

By the logic above, any shift-reduce conflict (and then some) will get worked out with precise look-ahead detail. If the conflict still exists in the resulting automaton, then yes of course you can use P&A on it. But if the strategy uses LR(0) items leading up to some state, then there was not a conflict on that state.

CLAIM:
It will be possible to conjure up the correct set of followers for LR(1)-mode head-items.

The algorithm for that part has *almost* nothing to do with the followers of the core item-sets. The only relevance is when the successor-to-what's-followed is transparent, in which case canonical LR(1) includes that item's followers as followers of the new head-item. By construction, this algorithm will have those followers at hand, because the corresponding core-item's head-item will have been tainted, so the core-item will be dragging a follower-set along.

4.9 Beyond LR(1)

It seems possible to try a variant of this idea to expand beyond LR(1). Basically, conflicts still present in the resulting LR(1) handle-finding automaton would result in a second cycle of tainting and then recomputing where specific LR(1) head-items must get LR(2) treatment – and so forth.

A parse engine for LR(k) would be rather more complicated. Again presumably most states don't require the full *k* tokens of look-ahead. And now the look-ahead is sort of a circular buffer that you alternately examine and pull from.

We don't want giant enormous wide parse tables just because there are `n^k` possible k-token look-aheads. Most likely you'd do a prefix tree.

4.10 An Actual Algorithm

The algorithm overall has three high-level phases:

1. Compute the LR(0) graph, and also keep the item-sets (both core and closure) for each LR(0) state.
2. Perform the tainting step.
3. Build the final automaton using the careful mix of LR(0) and LR(1) items as explained.

That tainting step in the middle is a bit tricky. So I propose a multi-step process for it:

a. Associate each LR(0) state with each of its predecessors, thus to save a lot of time later.
b. Starting from the reducing parse-items in each LR(0)-inadequate state, propagate the taint.
c. Treat the propagation step as a transitive-closure problem.

It will be handy to have a quick way to tell which rule (or its left-hand side) is associated with any given parse-item, and the offset from a parse-item to the beginning of its rule.

For any further detail than this, see the code.

4.11 Results and Next Steps

I have implemented the algorithm above instrumented my parser generator to compare the size of parse tables generated with LALR, Canonical LR(1), and this "Minimal" LR(1). Current results are that this algorithm produces *significantly* fewer states than Canonical, but often a good number more than LALR even for grammars that are LALR-compatible:

```
Decaf:
    Canonical: 550 States
    This Thing: 360 States
    LALR: 197 States

Pascal:
    Canonical: 1485 States
    This Thing: 702 States
    LALR: 275 States
```

I see two major approaches to improving on these results. Both try to be more precise about where canonical parse items are necessary.

Propagate follow-sets more precisely.

Right now if a head parse-item is tainted, effectively its entire body is tainted. But the *taint contagion* thing can afflict a parse-item in its middle, which perhaps might mean carrying follow-sets into portions of the automaton where it is not strictly needed.

There is a one-off hack specific to LR(0)-adequate reducing states, but some of their predecessors might be splitting needlessly in the present scheme.

Taint fewer items: Perhaps only those where LALR shows a potential conflict.

Presently any LR(0) parse conflict results in a taint. If there is no LALR conflict on any particular token, then such a state need not be considered tainted. The difference is how the parser behaves in the event of error.

A LALR table table can sometimes reduce a few rules before discovering that it really cannot shift the next token after all. In other words, it lacks the property of detecting errors *immediately*. This may not be that big a deal, but it's something to think about if you want to handle errors nicely.

Finally, there are techniques like unit-rule elimination and shift-reduce instructions. These optimizations can apply regardless of how you construct your table, so I don't consider them specially in this document. They have *not* been applied in the statistics above.

Why the odd name?

- A few programming languages were named for beverages.
- Booze is a beverage.
- I could find no programming language called Booze.

If anything is unclear, please feel free to file an issue at GitHub or contact me via e-mail. Feedback is always welcome. My username in most places is "kjosib" and I have a gmail account.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search