Constructing the LALR(1) CFSM

Michal Young
University of Oregon

v0.2, 2010.02.06

Abstract

Many textbooks describe construction of the LALR(1) context-free state machine (CFSM) with detailed pseudocode that, while it may be a useful guide to producing code, is less helpful than it ought to be for understanding how and why the construction works. This is an attempt to explain the construction well enough that you can actually work examples by hand, by explaining both exactly how it works and why each step is the way it is.

1 Introduction

LR parsing, and specifically LALR(1) parsing, is one of the great success stories of the interaction between theoretical computer science (languages and automata), algorithm design, and a "systems" area of computer science research. LALR(1) parser generators not only fundamentally changed the way compilers are constructed, but they also changed the way languages are designed: Most language designers use an LALR(1) parser generator as a design and debugging tool, treating its reports of shift/reduce and reduce/reduce errors as if they were spelling errors reported by their word processor.

LALR(1) parsing is also beautiful, but it's hard to see the beauty through the fog of a pseudocode presentation. The purpose of this short document is to try to present LALR(1) parse table construction in a way that is clear enough to (a) follow as a guide for performing the construction by hand, or implementing it with a program, (b) understand why it works the way it does, and so remember it, and (c) grasp and recall its beauty even after you've forgotten a few of its operational details, so that you can invent similar algorithms for related problems.

2 Preliminaries: Grammar Analysis

We will make use of two grammar analyses in the LALR(1) constructions below. A non-terminal symbol is *Nullable* if it can derive the empty sequence (even if it can also derive non-empty sequences). The *First* set of a non-terminal symbol is the set of terminal symbols that can be at the beginning of a sequence derived from the non-terminal symbol. The *First* set of a sequence of terminal and non-terminal symbols is the set of terminal symbols that could be at the beginning of a sentence derived from that sequence.

For all of these analyses, we assume that grammars are in a normal form without disjunction. That is, while we might like to write a grammar with productions like

$\mathbf{A} \rightarrow \mathbf{B} \mid \mathbf{C}$

all the rules below assume that this will be written as two productions

$\mathbf{A} \rightarrow \mathbf{B}$

$\mathbf{A} \rightarrow \mathbf{C}$

The definitions of *First* and *Nullable* are the same as for LL(1) parsing. Their clearest definitions are recursive.

2.1 Nullable

Rule 1 (Nullable)**.** *A terminal symbol is not nullable. If S is a non-terminal symbol with productions*

$\boldsymbol{S} \rightarrow \alpha$

$\ldots$

$\boldsymbol{S} \rightarrow \beta$

where $\alpha, \ldots, \beta$ are sequences of zero or more symbols, then, if for any one *of the right hand sides $\alpha, \ldots, \beta$, all of the symbols in that right hand side are nullable, then S itself is nullable.*

Note that this rule applies vacuously when the right hand side of a production

$\mathbf{S} \rightarrow \alpha$

is empty, i.e., S must be nullable if there are no symbols in α.

The recursive rule for nullable can be implemented in an iterative procedure. Initially none of the non-terminal symbols are marked nullable. We check all of the symbols to see if at least one is nullable, and on the first pass we mark those with empty right hand sides. Then we make a second pass through all the non-terminals, and we may mark some more non-terminals because all the symbols in a right hand

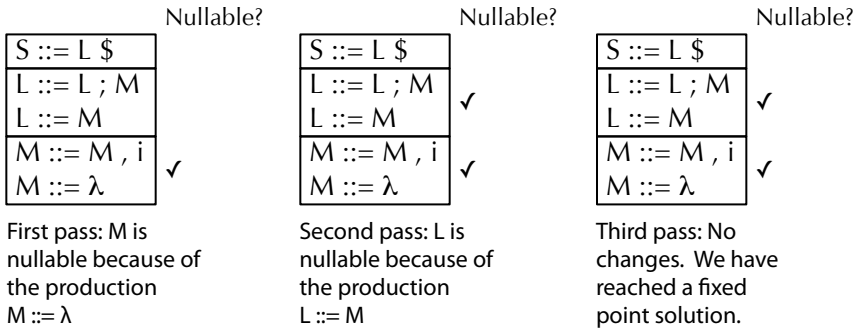

Figure 1: Computing Nullable

side have been marked nullable. We keep doing this until we make a pass through the whole grammar without marking any more nullable non-terminals. When nothing more can be changed by applying these rules, we have reached the solution. (Faster versions of the algorithm apply the same logic but inspect productions in a good order.)

2.2 First

We compute *First* sets on sequences of zero or more symbols.

Rule 2 (First)**.** *For the production*

$S \rightarrow \alpha$

where α is sequence of zero or more symbols, we distingusih three cases for First(α), depending on the form of α:

- *If α is empty, then First(α) is the empty set.*
- *If α is* x β*, and* x *is a terminal symbol, then First(α) is* { x }*.*
- *If α is a single non-terminal symbol **X**, First(**X**) is the union of the First sets of all the right-hand sides of its productions.*
- *If α is **X**β, and **X** is a non-terminal symbol, then*
  - *If **X** is nullable, then First(α) is First(**X**) $\cup$ First(β)*
  - *Otherwise First(α) is First(**X**)*

Like *Nullable* above, we can compute *First* with an iterative procedure. We keep track of the *First* set associated with each non-terminal symbol, from which we can

easily recompute the *First* set of any sequence of symbols, including the right hand sides of individual productions.

We first compute *Nullable* for all the non-terminals of the grammar, so we will know which of the two cases above to apply when we encounter a non-terminal symbol on the right hand side of a production. Then we initialize the *First* set of each non-terminal symbol to the empty set.

For the iteration, we look at each production and compute terminal symbols that should be added to the *First* set of any non-terminal symbol. Adding terminal symbols to the *First* set of one non-terminal may make it necessary to add terminal symbols to another *First* set; we keep checking and applying the rule until no more additions are possible. When no more changes are possible, the *First* sets are correct.

As for *Nullable*, fast algorithms for *First* follow the same logic but inspect productions in a good order. The simple form that I have described them in are called *chaotic iteration* to a fixed point. Algorithms that keep track of dependencies among productions, and inspect only those that depend on a non-terminal that has just changed, are called *work set algorithms*.

3 Basic Parts of the CFSM

3.1 Item

An *item* is a grammar production with a dot somewhere in its right hand side. Think of it as a record of where you are (or might be) in parsing a production. For example, the item

$\mathbf{E} \rightarrow \mathbf{E} \bullet + \mathbf{T}$

means that we are parsing an expression (E), and have already matched one sub-expression; we can finish parsing this production if we see a plus and then a term (T).

A basic concept of LR or bottom-up parsing is that we don't make up our mind which production we are parsing too early, so in many cases we will construct states that contain several items. Each item represents our progress in some *possible* parse of the input so far. Some of the possibilities might represent different points in parsing the same production, like this:

$\mathbf{E} \rightarrow \mathbf{E} \bullet + \mathbf{T}$

$\mathbf{E} \rightarrow \bullet \mathbf{E} + \mathbf{T}$

These are two distinct items. They both describe parsing the input with the same production. The first indicates that we *might* be past the first sub-expression,

and about to see the plus. The second indicates that we might be at the beginning, waiting to see the first sub-expression.

3.2 Lookahead

In LALR(1) parse tables, each item in a state is associated with a set of *lookahead* tokens.[1] We'll see how to generate and propagate them below. For now, the important thing to know is that the meaning of the lookeahead set is "if this item is the correct parse for the input, then the next thing we should see *after* parsing the rest of the production is one of these tokens."

3.3 State

An CFSM *state* is a set of items with lookaheads. (Because several of the items will typically have the same lookahead set, we often group them in drawing the state, but in principle each item has its own lookahead set).

Technically, the state is just the set of items, and the lookaheads are extra information. This is important because of the rule for merging states: Two states made up of the same set of items are really the same state, and we draw it only once, *even if the lookaheads are different.*[2] In case we merge two states with different sets of lookaheads, we keep the lookaheads from both copies, and merge them. In other words, the lookahead sets in a CFSM state are the unions of the what the lookahead sets would be on all the possible paths to that state.

4 Constructing a CFSM State from a Kernel

A *kernel*, just like a CFSM state, is a set of items with their associated lookaheads. It's what we start with to build a state. We construct the whole state by repeated application of a single rule. First we give a version of the rule for LR(0) state construction, and then give a slightly more complicated version for LALR(1) state construction.

Rule 3 (LR(0) State Completion)**.** *If there is an item with the dot immediately before a non-terminal symbol P, like*

[1] The lookahead tokens are like the "follow" set in LL(1) parsing. You can test your understanding by proving that the LALR(1) lookahead set for an item is always a subset of the LL(1) follow set for the non-terminal symbol on the left-hand side of the production.

[2] This is the difference between LR(1) and LALR(1). In LR(1), we consider lookaheads to be part of the state, so we would not merge two states with different lookaheads. This results in more states, which are usually (but not always) just wasted expense, so most parser generators use LALR(1) rather than LR(1).

$\boldsymbol{S} \to \cdots \bullet \boldsymbol{P} \cdots$
then for every production
$\boldsymbol{P} \to \cdots$
in the grammar, add an item
$P \to \bullet \cdots$
to the state.

If applying rule 3 creates an item that is already part of the state (with the dot in the same place — otherwise it's not the same item), then we merge them. We simply apply rule 3 over and over, until all the items we can add are already present, so nothing changes.

Why? Because if we could be about to parse a non-terminal P, we'd better look for all the ways a P could be built from the terminal symbols in the input stream. Think of it as angelic non-determinism in a top-down parse: Instead of calling one procedure to parse a P, and having to decide immediately which of the P productions is the right one to use, we add an item for each of the productions for P, and keep track of all of them at once (marking our place with the dot). Later we may discard some of them as the wrong production, but we don't need to decide yet.

The LALR(1) version of rule 3 is almost the same, except that we need to consider lookaheads at the same time we add items.

Rule 4 (LALR(1) State Completion)**.** *If there is an item with the dot immediately before a non-terminal symbol P, like*

$\boldsymbol{S} \to \cdots \bullet \boldsymbol{P}\alpha \ \{L\}$
(where { L } is the lookahead set, and α is the sequence of zero or more terminal and non-terminal symbols that follow P in the production) then for every production
$\boldsymbol{P} \to \cdots$
in the grammar, add an item
$P \to \bullet \cdots \ \{M\}$
to the state.

The new lookahead set M depends on the sequence of symbols α. M includes First(α), and if α is nullable, then M also includes the elements of L.

We apply rule 4 repeatedly, just like rule 3 above, again stopping when nothing changes. We merge items that are identical except for their lookahead sets, taking the union of their lookahead sets. We are done when *nothing* changes …and that includes the lookahead sets. It is quite common to apply rule 4 a few times without adding any new items, but adding a few symbols to lookahead sets.

5 Adding Transitions

So far we know how to construct a full LALR(1) state from a kernel. Now all we need is to create that transitions from a state. The rule for creating transitions also determines the kernel of the state the transition goes to, from which we can create the rest of the state.

From a state with a set of items

...

$\mathbf{S} \rightarrow \cdots \bullet \mathbf{P}\alpha \ \{L\}$

$\mathbf{T} \rightarrow \cdots \bullet \mathbf{Q}\beta \ \{M\}$

$\mathbf{U} \rightarrow \cdots \bullet \mathbf{P}\gamma \ \{N\}$

...

we will make one transition for each symbol directly following a dot. For example, if there are two items with the dot just before **P** in the set of items shown here, so we make a transition on symbol **P** to a state whose kernel is

$\mathbf{S} \rightarrow \cdots \mathbf{P} \bullet \alpha \ \{L\}$

$\mathbf{U} \rightarrow \cdots \mathbf{P} \bullet \gamma \ \{N\}$

Note that the lookaheads are unchanged by the transition; only the dot has moved. We complete the new state following the rules outlined above, and if it has the same items as an existing state, we merge the states and their lookahead sets. Note that we really only need to compare the kernel of the new state to the kernels of existing states, since all the other items in a state depend on the kernel.[3]

If the symbol we moved the dot over was a terminal symbol, we call the transition a *shift*. If the symbol we moved the dot over was a non-terminal symbol, the transition is a *goto*. The actual parse table data structure produced by a program like Bison or Cup keeps shifts and gotos separate, but conceptually and also for constructing the CFSM, they are exactly the same.

6 Priming the Pump

The kernel of the first state in the CFSM is an item (or set of items) for the start symbol of th grammar, with the dot at the beginning of the right hand side. Typically it will be something like

[3]Also, if the kernels of two states are not the same, it is impossible for the completions of those states to be the same. Can you prove this? It's a simple argument, but if you see it, then you are understanding LALR(1) parse table construction pretty well.

S $\rightarrow$ • **E** \$

Complete this initial state, then generate its transitions, complete each of those states and their transitions, and so on, and you will have the CFSM.