Briefly on Bottom-up

Paola Quaglia
University of Trento

October 8, 2017

Abstract

These short notes provide descriptions and references relative to the construction of parsing tables for SLR(1), for LR(1), and for LALR(1) grammars.

1 Notation and basic definitions

Basic definitions and notational conventions are summerized below.

A context-free grammar is a tuple $\mathcal{G} = (V, T, S, \mathcal{P})$, where the elements of the tuple represent, respectively, the vocabulary of terminal and nonterminal symbols, the set of terminal symbols, the start symbol, and the set of productions. Productions have the shape $A \to \beta$ where $A \in V \setminus T$ is called the *driver*, and $\beta \in V^*$ is called the *body*. The one-step rightmost derivation relation is denoted by "$\Rightarrow$", and "$\Rightarrow^*$" stands for its reflexive and transitive closure. A grammar is said *reduced* if it does not contain any useless production, namely any production that is never involved in the derivation of strings of terminals from S. In what follows we assume grammars be reduced.

The following notational conventions are adopted. The empty string is denoted by ϵ. Lowercase letters early in the Greek alphabet stand for strings of grammar symbols ($\alpha, \beta, \ldots \in V^*$), lowercase letters early in the alphabet stand for terminals ($a, b, \ldots \in T$), uppercase letters early in the alphabet stand for nonterminals ($A, B, \ldots \in (V \setminus T)$), uppercase letters late in the alphabet stand for either terminals or nonterminals ($X, Y, \ldots \in V$), and strings of terminals, i.e. elements of T^*, are ranged over by $w, w_0, \ldots$.

For every α, first(α) denotes the set of terminals that begin strings w such that $\alpha \Rightarrow^* w$. Moreover, if $\alpha \Rightarrow^* \epsilon$ then $\epsilon \in$ first(α). For every A, follow(A) denotes the set of terminals that can follow A in a derivation, and is defined in the usual way.

Given any context-free grammar $\mathcal{G}$, parsing is applied to strings followed by the symbol $\$ \notin V$ used as endmarker. Also, the parsing table is produced for an enriched version of $\mathcal{G}$, denoted by $\mathcal{G}' = (V', T, S', \mathcal{P}')$. The enriched grammar $\mathcal{G}'$ is obtained from $\mathcal{G}$ by augmenting V with a fresh nonterminal symbol S', and by adding the production $S' \to S$ to $\mathcal{P}$.

An LR(0)-item of $\mathcal{G}'$ is a production of $\mathcal{G}'$ with the distinguished marker "$\cdot$" at some position of its body, like, e.g., $A \to \alpha \cdot \beta$. The single LR(0)-item for a production of the shape $A \to \epsilon$ takes the form $A \to \cdot$. The LR(0)-items $S' \to \cdot S$ and $S' \to S \cdot$ are called, respectively, *initial item* and *accepting item*. The LR(0)-item $A \to \alpha \cdot \beta$ is called

- *kernel item* if it is either initial or such that $\alpha \neq \epsilon$,
- *closure item* if it is not kernel, and
- *reducing item* if it is not accepting and if $\beta = \epsilon$.

For a set of LR(0)-items P, kernel(P) is the set of the kernel items in P. By definition, the initial item is the single kernel item of $\mathcal{G}'$ with the dot at the leftmost possible position, and items of the shape $A \to \cdot$ are the only non-kernel reducing items.

An LR(1)-item of $\mathcal{G}'$ is a pair consisting of an LR(0)-item of $\mathcal{G}'$ and of a subset of $T \cup \{\$\}$, like, e.g., $[A \to \alpha \cdot \beta, \{a, \$\}]$. The second component of an LR(1)-item is called lookahead-set and is ranged over by $\Delta, \Gamma, \ldots$. An LR(1)-item is said *initial*, *accepting*, *kernel*, *closure* or *reducing* if so is its first component. For a set P of LR(1)-items, prj(P) is the set of LR(0)-items occurring as first components of the elements of P. Also, function kernel(_) is overloaded, so that for a set of LR(1)-items P, kernel(P) is the set of the kernel items in P.

2 Characteristic automata and parsing tables

Given a context-free grammar $\mathcal{G}$ and a string of terminals w, the aim of bottom-up parsing is to deterministically reconstruct, in reverse order and while reading w from the left,a rightmost derivation of w if the string belongs to the language generated by $\mathcal{G}$. If w does not belong to the language, then parsing returns an error. The computation is carried over on a stack, and before terminating with success or failure, it consists in *shift* steps and in *reduce* steps. A shift step amounts to pushing onto the stack the symbol of w that is currently pointed by the input cursor, and then advancing the cursor. Each reduce step is relative to a specific production of $\mathcal{G}$. A reduce step under $A \to \beta$ consists in popping β off the stack and then pushing A onto it. Such reduction is the appropriate kind of move when, for some α and w_1, the global content of the stack is $\alpha\beta$ and the rightmost derivation of the analyzed string w takes the form

$$S \Rightarrow^* \alpha A w_1 \Rightarrow \alpha\beta w_1 \Rightarrow^* w. \tag{1}$$

A seminal result by Knuth [9] is that for reduced grammars the language of the *characteristic strings*, i.e. of the strings like $\alpha\beta$ in (1), is a regular language. By that, a deterministic finite state automaton can be defined and used as the basis of the finite control of the parsing procedure [3, 4]. This automaton is referred to as the *characteristic automaton*, and is at the basis of the construction of the actual controller of the parsing algorithm, the so-called *parsing table*.

If $\mathcal{Q}$ is the set of states of the characteristic automaton, then the parsing table is a matrix $\mathcal{Q}\times(V\cup\{\$\})$, and the decision about which step to take next depends on the current state and on the symbol read from the parsed word. Various parsing techniques use the same shift/reduce algorithm but are driven by different controllers, which in turn are built on top of distinct characteristic automata.

States of characteristic automata are sets of items. A state P contains the item $A \to \alpha \cdot \beta$ (or an item whose first projection is $A \to \alpha \cdot \beta$) if P is the state reached after recognizing a portion of the parsed word whose suffix corresponds to an expansion of α. Each state of the characteristic automaton is generated from a kernel set of items by closing it up to include all those items that, w.r.t. the parsing procedure, represent the same progress as that expressed by the items in the kernel.

The transition function τ of the automaton describes the evolution between configurations. Every state has as many transitions as the number of distinct symbols that follow the marker "$\cdot$" in its member items. Assume the parser be in state P_n, and let a be the current symbol read from the parsed word. If the entry (P_n, a) of the parsing table is a shift move, then the control goes to the state $\tau(P_n, a)$. If it is a reduction move under $A \to \beta$, then the next state is $\tau(P, A)$ where P is the origin of the path spelling β and leading to P_n. Precisely, suppose that $\beta = Y_1 \dots Y_n$ and let $P \xmapsto{Y} P'$ denote that $\tau(P, Y) = P'$. Then the state of the parser after the reduction $A \to \beta$ in P_n is $\tau(P, A)$ where P is such that $P \xmapsto{Y_1} P_1 \xmapsto{Y_2} \dots \xmapsto{Y_n} P_n$.

The common features of the various characteristic automata used to construct bottom-up parsing tables are listed below.

- Each state in the set of states $\mathcal{Q}$ is a set of items.
- The initial state contains the initial item.
- The set $\mathcal{F}$ of final states consists of all the states containing at least one reducing item.
- The vocubulary is the same as the vocabulary of the given grammar, so that the transition function takes the form $\tau : (\mathcal{Q} \times V) \to \mathcal{Q}$.

As said, in the shift/reduce algorithm, the decision about the next step depends on the current configuration of the parser and on the current input terminal. So, in order to set up a parsing table, it is also necessary to define, for each final state Q and for each reducing item in Q, which set of terminals should trigger the relative reduction. This is achieved by providing an actual definition of the *lookahed function* $\mathcal{LA} : \mathcal{F} \times \mathcal{P} \to \wp(V \cup \{\$\})$. For the argument pair $(P, A \to \beta)$ the lookahead function returns the set of symbols calling for a reduction after $A \to \beta$ when the parser is in state P. E.g., referring to (1) and assuming that P is the state of the parser when $\alpha\beta$ is on the stack, $\mathcal{LA}(P, A \to \beta)$ is expected to contain the first symbol of w_1.

Once the underlying characteristic automaton and lookahead function are defined, the corresponding parsing table is obtained as described below.

Definition 2.1. *Let $\mathcal{Q}$, V, and τ be, respectively, the set of states, the vocabulary, and the transition function of a characteristic automaton. Also, let $\mathcal{LA}_i$ be an actual instance of the lookahead function. Then, the* parsing table *for the pair consisting of the given characteristic automaton and the given lookahead function is the matrix $\mathcal{Q} \times (V \cup \{\$\})$ obtained by filling in each entry (P, Y) after the following rules.*

- *Insert* "Shift Q" *if Y is a terminal and $\tau(P, Y) = Q$.*
- *Insert* "Reduce $A \to \beta$" *if P contains a reducing item for $A \to \beta$ and $Y \in \mathcal{LA}_i(P, A \to \beta)$.*
- *Set to* "Accept" *if P contains the accepting item and $Y = \$$.*
- *Set to* "Error" *if Y is a terminal or \$, and none of the above applies.*

Algorithm 1: Construction of either LR(0)-automaton or LR(1)-automaton (P_0 and closure(_) to be instantiated accordingly)

```
initialize $\mathcal{Q}$ to contain $P_0$;
tag $P_0$ as unmarked;
while there is an unmarked state $P$ in $\mathcal{Q}$ do
    mark $P$ ;
    foreach grammar symbol $Y$ do
        $Tmp \longleftarrow \emptyset$;
                        /* Compute the kernel-set of the $Y$-target of $P$ */
        foreach $A \to \alpha \cdot Y\beta \in P$ do
            add $A \to \alpha Y \cdot \beta$ to $Tmp$;
        if $Tmp \neq \emptyset$ then
                        /* Check whether $\tau(P, Y)$ has already been collected */
            if $Tmp = \text{kernel}(Q)$ for some $Q$ in $\mathcal{Q}$ then
                $\tau(P, Y) \longleftarrow Q$;
            else
                $New\_state \longleftarrow \text{closure}(Tmp)$;
                $\tau(P, Y) \longleftarrow New\_state$;
                add $New\_state$ as an unmarked state to $\mathcal{Q}$ ;
```

- *Set to* "Goto Q" *if* Y *is a nonterminal and* $\tau(P, Y) = Q$.

The table might have multiply-defined entries, mentioning either a shift and a reduce directive (known as a shift/reduce conflict), or multiple reduce directives for different productions (known as a reduce/reduce conflict). If so, then the constructed table cannot possibly drive a deterministic parsing procedure. Consequently, grammar $\mathcal{G}$ is said not to belong to the class of grammars syntactically analyzable by the methodology (choice of automaton and of lookahead function) underlying the definition of the parsing table. Viceversa, if the constructed parsing table contains no conflict, then $\mathcal{G}$ belongs to the class of grammars parsable by the methodology corresponding to the chosen pair of automaton and of instance of $\mathcal{LA}_i$.

Below we focus on SLR(1) grammars, LR(1) grammars, and LALR(1) grammars. Seen as classes of grammars, SLR(1) is strictly contained in LALR(1) which is strictly contained in LR(1).

All the algorithms referred to in these short notes are collected in the Appendix. Some of the algorithms are run on the grammar $\mathcal{G}_1$ below, which is taken from [2], and separates the class SLR(1) from the class LALR(1).

$$\begin{array}{lrcl} \mathcal{G}_1 : & S & \to & L = R \mid R \\ & L & \to & *R \mid \text{id} \\ & R & \to & L \end{array}$$

The language generated by $\mathcal{G}_1$ can be thought of as a language of assignments of r-values to l-values, where an l-value can denote the content of an r-value.

3 SLR(1) grammars

Algorithm 2: Computation of $\text{closure}_0(Q)$

```
function closure_0($P$)
    tag every item in $P$ as unmarked ;
    while there is an unmarked item $I$ in $P$ do
        mark $I$ ;
        if $I$ has the form $A \to \alpha \cdot B\beta$ then
            foreach $B \to \gamma \in \mathcal{P}'$ do
                if $B \to \cdot\gamma \notin P$ then
                    add $B \to \cdot\gamma$ as an unmarked item to $P$ ;
    return $P$ ;
```

The SLR(1) parsing table for $\mathcal{G}$ is constructed from an automaton, called LR(0)-automaton, whose states are sets of LR(0)-items. Correspondingly, function $\mathcal{LA}_i$ is instantiated as follows.

> *For every final state P of the LR(0)-automaton and for every $A \to \beta\cdot \in P$,*
> $\mathcal{LA}_{SLR}(P, A \to \beta) = \text{follow}(A)$.

LR(0)-automata are obtained by applying Alg. 1 after:

- using $\text{closure}_0(\_)$ (see Alg. 2) as $\text{closure}(\_)$ function, and
- taking $P_0 = \text{closure}_0(\{S' \to \cdot S\})$.

The intuition behind the definition of $\text{closure}_0(\_)$ is that, if the parsing procedure progressed as encoded by $A \to \alpha \cdot B\beta$, and if $B \to \gamma \in \mathcal{P}'$, then the coming input can be an expansion of γ followed by an expansion of β. In fact, $\text{closure}_0(P)$ is defined as the smallest set of items that satisfies the following equation:

$$\text{closure}_0(P) = P \cup \{B \to \cdot\gamma \text{ such that } A \to \alpha \cdot B\beta \in \text{closure}_0(P) \text{ and } B \to \gamma \in \mathcal{P}'\}.$$

As an example of application of Alg. 2, the items belonging to $\text{closure}_0(\{S' \to \cdot S\})$ for $\mathcal{G}_1$ are shown below.

$$\begin{array}{rl}
\text{closure}_0(\{S' \to \cdot S\}): & S' \to \cdot S \\
 & S \to \cdot L = R \\
 & S \to \cdot R \\
 & L \to \cdot * R \\
 & L \to \cdot \text{id} \\
 & R \to \cdot L
\end{array}$$

The rationale for Alg. 1 is the following.

- Compute the set of states of the automaton by starting from the initial state P_0 and incrementally adding the targets, under possible Y-transitions, of states already collected.

- To decide which, if any, is the Y-target of a certain state P, first compute in Tmp the set of the kernel items of the Y-target.
- Compare Tmp to the states in the current collection. If, for some collected Q, Tmp and Q have the same kernel items, then take Q as the Y-target of P. If no match is found for Tmp, then add $\text{closure}_0(\mathit{Tmp})$ to the current collection of states.

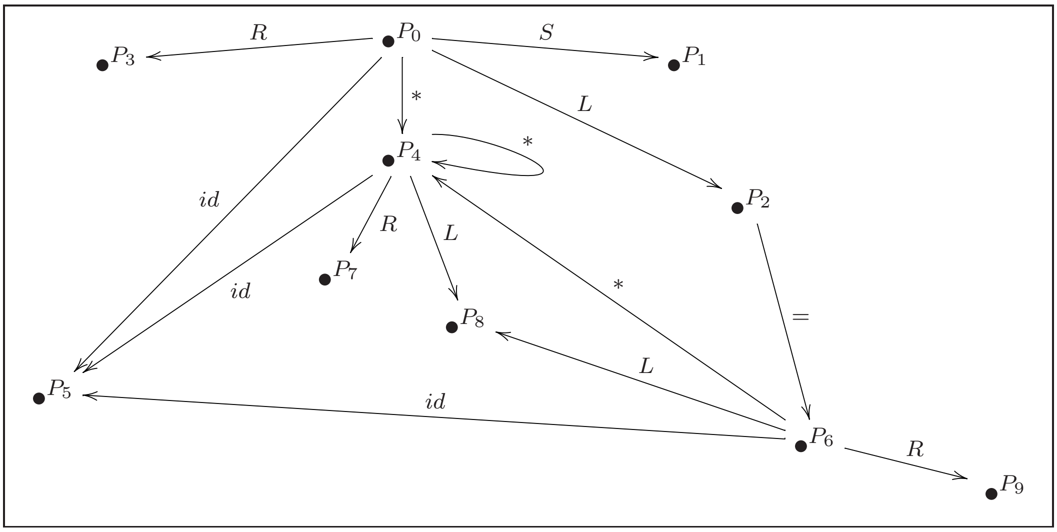

Figure 1: Layout of the LR(0)-automaton for $\mathcal{G}_1$

The layout of the LR(0)-automaton for $\mathcal{G}_1$ is reported in Fig. 1. The accepting item is in state P_1. The final states of the automaton, and the reducing items they contain, are listed below.

State	**Reducing item**
P_2 :	$R \to L\cdot$
P_3 :	$S \to R\cdot$
P_5 :	$L \to \text{id}\cdot$
P_7 :	$L \to *R\cdot$
P_8 :	$R \to L\cdot$
P_9 :	$S \to L = R\cdot$

$\mathcal{G}_1$ is not SLR(1). Indeed, the SLR(1) parsing table for $\mathcal{G}_1$ has a shift/reduce conflict at the entry $(P_2, =)$. This is due to the fact that P_2 has an outgoing transition labelled by $=$ (which induces a shift to P_6), and to the fact that $= \in \text{follow}(R)$ (which induces a reduce after $R \to L$).

4 LR(1) grammars

The LR(1) parsing table for $\mathcal{G}$ is constructed from an automaton, called LR(1)-automaton, whose states are sets of LR(1)-items. Correspondingly, function $\mathcal{LA}_i$ is instantiated as follows.

Algorithm 3: Computation of closure$_1(P)$

```
function closure$_1(P)$
    tag every item in $P$ as unmarked ;
    while there is an unmarked item $I$ in $P$ do
        mark $I$ ;
        if $I$ has the form $[A \to \alpha \cdot B\beta, \Delta]$ then
            $\Delta_1 \longleftarrow \bigcup_{d \in \Delta} \text{first}(\beta d)$ ;
            foreach $B \to \gamma \in \mathcal{P}'$ do
                if $B \to \cdot\gamma \notin \text{prj}(P)$ then
                    add $[B \to \cdot\gamma, \Delta_1]$ as an unmarked item to $P$ ;
                else
                    if $([B \to \cdot\gamma, \Gamma] \in P$ and $\Delta_1 \not\subseteq \Gamma)$ then
                        update $[B \to \cdot\gamma, \Gamma]$ to $[B \to \cdot\gamma, \Gamma \cup \Delta_1]$ in $P$ ;
                        set $[B \to \cdot\gamma, \Gamma \cup \Delta_1]$ as unmarked ;
    return $P$ ;
```

For every final state P of the LR(1)-automaton and for every $[A \to \beta\cdot, \Delta] \in P$, $\mathcal{LA}_{LR}(P, A \to \beta) = \Delta$.

LR(1)-automata are obtained by applying Alg. 1 after:

- using closure$_1(\_)$ (see Alg. 3) as closure$(_)$ function, and
- taking $P_0 = \text{closure}_1(\{[S' \to \cdot S, \{\$\}]\})$.

When applied to an item with projection $A \to \alpha \cdot B\beta$, closure$_1(\_)$ refines closure$_0(\_)$ by propagating the symbols following B to the closure items whose driver is B. By definition, closure$_1(P)$ is the smallest set of items, with smallest lookahead-sets, that satisfies the following equation:

$$\text{closure}_1(P) = P \cup \{[B \to \cdot\gamma, \Gamma] \text{ such that } [A \to \alpha \cdot B\beta, \Delta] \in \text{closure}_1(P) \text{ and } B \to \gamma \in \mathcal{P}' \text{ and } \text{first}(\beta\Delta) \subseteq \Gamma\}.$$

The computation of closure$_1(\{[S' \to \cdot S, \{\$\}]\})$ for $\mathcal{G}_1$ is detailed in the following, where we assume that items are processed in the same order in which they are tagged as unmarked in the collection under construction.

1. First round of **while**
    - $[S' \to \cdot S, \{\$\}]$ taken as I, marked
    - $\Delta_1 = \{\$\}$
    - $[S \to \cdot L = R, \{\$\}]$ added to P, unmarked
    - $[S \to \cdot R, \{\$\}]$ added to P, unmarked.
2. Next round of **while**

- $[S \to \cdot L = R, \{\$\}]$ taken as I, marked
- $\Delta_1 = \{=\}$
- $[L \to \cdot * R, \{=\}]$ added to P, unmarked
- $[L \to \cdot \mathrm{id}, \{=\}]$ added to P, unmarked.

3. Next round of **while**
    - $[S \to \cdot R, \{\$\}]$ taken as I, marked
    - $\Delta_1 = \{\$\}$
    - $[R \to \cdot L, \{\$\}]$ added to P, unmarked.
4. Next round of **while**
    - $[L \to \cdot * R, \{=\}]$ taken as I, marked.
5. Next round of **while**
    - $[L \to \cdot \mathrm{id}, \{=\}]$ taken as I, marked.
6. Next round of **while**
    - $[R \to \cdot L, \{\$\}]$ taken as I, marked
    - $\Delta_1 = \{\$\}$
    - $[L \to \cdot * R, \{=\}]$ updated to $[L \to \cdot * R, \{=, \$\}]$, unmarked
    - $[L \to \cdot \mathrm{id}, \{=\}]$ updated to $[L \to \cdot \mathrm{id}, \{=, \$\}]$, unmarked.
7. Next round of **while**
    - $[L \to \cdot * R, \{=, \$\}]$ taken as I, marked.
8. Last round of **while**
    - $[L \to \cdot \mathrm{id}, \{=, \$\}]$ taken as I, marked.

The layout of the LR(1)-automaton for $\mathcal{G}_1$ is reported in Fig. 2. The accepting item is in state P_1. The final states of the automaton, and the reducing items they contain, are listed below.

State	**Reducing item**
P_2 :	$[R \to L\cdot, \{\$\}]$
P_3 :	$[S \to R\cdot, \{\$\}]$
P_5 :	$[L \to \mathrm{id}\cdot, \{=, \$\}]$
P_7 :	$[L \to *R\cdot, \{=, \$\}]$
P_8 :	$[R \to L\cdot, \{=, \$\}]$
P_9 :	$[S \to L = R\cdot, \{\$\}]$
P_{10} :	$[R \to L\cdot, \{\$\}]$
P_{12} :	$[L \to \mathrm{id}\cdot, \{\$\}]$
P_{13} :	$[L \to *R\cdot, \{\$\}]$

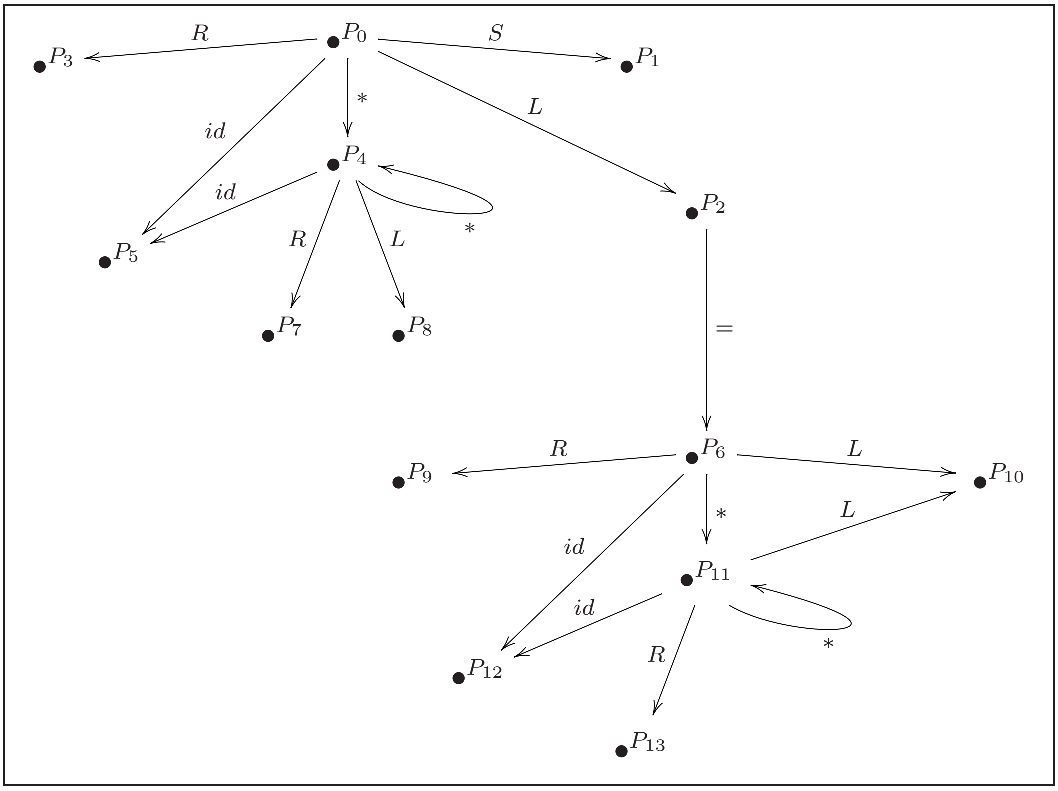

Figure 2: Layout of the LR(1)-automaton for $\mathcal{G}_1$

5 LALR(1) grammars

LALR(1) parsing tables are based on automata whose size is the same as the size of LR(0)-automata. Various algorithms achieve the same goal.

From LR(1)-automata

The less efficient algorithm for the construction of LALR(1) parsing tables is based on the use of *LRm(1)-automata* (for LR(1)-*merged*-automata).

The construction of the LRm(1)-automaton for $\mathcal{G}'$ requires, as pre-processing, the computation of the LR(1)-automaton for $\mathcal{G}'$, say $\mathscr{A}_l$. Then the states and the transitions of the LRm(1)-automaton are defined as follows.

States: Every state of the LRm(1)-automaton for $\mathcal{G}'$ is obtained by merging together all the LR(1)-items belonging to the states of $\mathscr{A}_l$ with the same projection.

Transitions: If the state M of the LRm(1)-automaton is obtained by merging the states $L_1, \ldots, L_k$ of $\mathscr{A}_l$, and if L_1 has a Y-transition to L', then the LRm(1)-automaton has a Y-transition from M to M' where M' is such that $\mathrm{prj}(M') = \mathrm{prj}(L')$.

The LALR(1) parsing table for $\mathcal{G}$ is constructed from the LRm(1)-automaton, and instantiating function $\mathcal{LA}_i$ as follows.

Algorithm 4: Construction of the symbolic automaton

```
$x_0 \longleftarrow$ newVar();
$Vars \longleftarrow \{x_0\}$;
$P_0 \longleftarrow$ closure$_1(\{[S' \rightarrow \cdot S, \{x_0\}]\})$;
initialize $Eqs$ to contain the equation $x_0 \doteq \{\$\}$;
initialize $\mathcal{Q}$ to contain $P_0$;
tag $P_0$ as unmarked;
while there is an unmarked state $P$ in $\mathcal{Q}$ do
    mark $P$ ;
    foreach grammar symbol $Y$ do
                                  /* Compute the kernel-set of the $Y$-target of $P$ */
        $Tmp \longleftarrow \emptyset$;
        foreach $[A \rightarrow \alpha \cdot Y\beta, \Delta]$ in $P$ do
            add $[A \rightarrow \alpha Y \cdot \beta, \Delta]$ to $Tmp$;
        if $Tmp \neq \emptyset$ then
            if prj($Tmp$) = prj(kernel($Q$)) for some $Q$ in $\mathcal{Q}$ then
                                  /* Refine $Eqs$ to let $Q$ be the $Y$-target of $P$ */
                foreach ($[A \rightarrow \alpha Y \cdot \beta, \Delta] \in Tmp$ , $[A \rightarrow \alpha Y \cdot \beta, \{x\}] \in$ kernel($Q$)) do
                    update $(x \doteq \Gamma)$ to $(x \doteq \Gamma \cup \Delta)$ in $Eqs$;
                $\tau(P, Y) \longleftarrow Q$;
            else
                                  /* Generate the $Y$-target of $P$ */
                foreach $[A \rightarrow \alpha Y \cdot \beta, \Delta] \in Tmp$ do
                    $x \longleftarrow$ newVar();
                    $Vars \longleftarrow Vars \cup \{x\}$;
                    enqueue $(x \doteq \Delta)$ into $Eqs$;
                    replace $[A \rightarrow \alpha Y \cdot \beta, \Delta]$ by $[A \rightarrow \alpha Y \cdot \beta, \{x\}]$ in $Tmp$;
                $\tau(P, Y) \longleftarrow$ closure$_1(Tmp)$;
                add $\tau(P, Y)$ as an unmarked state to $\mathcal{Q}$ ;
```

Algorithm 5: Reduced system of equations *REqs* for the variables in *RVars* $\subseteq$ *Vars*

```
inizialize $RVars$ and $REqs$ to $\emptyset$ ;
while  $Eqs$ not empty do
    $x \doteq \Delta \longleftarrow$ dequeue($Eqs$) ;
    if  $\Delta \setminus \{x\} = \{x'\}$ then
        $class(x) \longleftarrow class(x')$ ;
    else
        $class(x) \longleftarrow x$ ;
        add $x$ to $RVars$ ;
foreach  $x \in RVars$ such that $x \doteq \Delta \in Eqs$ do
    update each $x'$ in $\Delta$ to $class(x')$ ;
    add $x \doteq \Delta \setminus \{x\}$ to $REqs$ ;
```

Algorithm 6: Computation of the actual values of variables

```
foreach $x$ do
    $D(x) \longleftarrow 0$ ;
foreach $x$ in $RVars$ do
    if $D(x) = 0$ then
        traverse($x$) ;
where
function traverse($x$)
    push $x$ onto stack $S$ ;
    $depth \longleftarrow$ number of elements in $S$ ;
    $D(x) \longleftarrow depth$ ;
    $val(x) \longleftarrow init(x)$ ;
    foreach $x'$ such that there is an edge in $G$ from $x$ to $x'$ do
        if $D(x') = 0$ then
            traverse($x'$)
        $D(x) \longleftarrow min(D(x), D(x'))$ ;
        $val(x) \longleftarrow val(x) \cup val(x')$ ;
    if  $D(x) = depth$ then
        repeat
            $D(top(S)) \longleftarrow \infty$ ;
            $val(top(S)) \longleftarrow val(x)$ ;
        until  $pop(S) = x$;
```

For every final state P of the LRm(1)-automaton and for every $\{[A \rightarrow \beta\cdot, \Delta_j]\}_j \subseteq P$, $\mathcal{LA}_{LRm}(P, A \rightarrow \beta) = \bigcup_j \Delta_j$.

From smaller automata

The algorithm described in Sec. 4.7.5 of the international edition of [1] is the algorithm implemented in the parser generator `Yacc` [8]. It uses LR(0)-automata as underlying characteristic automata for the contruction of LALR(1) parsing tables. The computation of the lookahead function is then based on a post-processing phase carried on that automaton. The post-processing phase of the `Yacc` algorithm consists in performing closure$_1$-operations that allow the identification of "generated" lookaheads. In various passes, the generated lookaheads are then propagated, along the edges of the LR(0)-automaton, to the appropriate reducing items.

`Bison`, another well-known parser generator [6], applies an algorithm designed by DeRemer and Pennello and presented in [5]. This algorithm is also based on the post-processing of LR(0)-automata. In a nutshell, starting from the state P where the reducing item $A \rightarrow \beta\cdot$ is located, the algorithm by DeRemer and Pennello traverses the automaton to infer which precise subset of the productions of the grammar should be considered when computing the follow-set of A for the item $A \rightarrow \beta\cdot$ in P.

Below, we describe an algorithm based on the construction of specialized *symbolic characteristic automata* [10]. The states of these automata are sets of *symbolic items*, which have the same structure as LR(1)-items. The lookahead-sets of symbolic items, however, can also contain elements from a set $\mathbb{V}$ which is disjoint from $V' \cup \{\$\}$. Elements of $\mathbb{V}$ are called *variables* and are ranged over by $x, x', \ldots$. In what follows, we use $\Delta, \Delta', \ldots, \Gamma, \Gamma', \ldots$ to denote subsets of $\mathbb{V} \cup T \cup \{\$\}$. Also, we let $\text{ground}(\Delta) = \Delta \cap (T \cup \{\$\})$. Moreover, we assume the existence of a function newVar() which returns a fresh symbol of $\mathbb{V}$ at any invocation. The definitions of initial, accepting, kernel, closure, and reducing items are extended to symbolic items in the natural way. Also, functions prj(_) and kernel(_) are overloaded to be applied to sets of symbolic items.

Variables are used to construct on-the-fly a symbolic version of the LRm(1)-automaton. In every state P of the symbolic automaton, the lookahead-set of kernel items is a singleton set containing a distinguished variable, like, e.g. $[A \rightarrow \alpha Y \cdot \beta, \{x\}]$. On the side, an equation for x collects all the contributions to the lookahead-set of $A \rightarrow \alpha Y \cdot \beta$ coming from the items with projection $A \rightarrow \alpha \cdot Y\beta$ which are located in the states Q_i with a Y-transition to P. When a new state P is generated and added to the current collection, closure$_1$(_) symbolically propagates the lookaheads encoded by the variables associated with the kernel items of P to the closure items of the state. When the construction of the symbolic automaton is over, the associated system of equations over variables is resolved to compute, for every variable x, the subset of $T \cup \{\$\}$ that is the actual value of x, denoted by $val(x)$. The evaluation of variables, in turn, is used to actualize lookahead-sets. In particular, function $\mathcal{LA}_i$ is instantiated as follows.

For every final state P of the symbolic automaton and for every $[A \rightarrow \beta\cdot, \Delta] \in P$, $\mathcal{LA}_{LALR}(P, A \rightarrow \beta) = \text{ground}(\Delta) \cup \bigcup_{x \in \Delta} val(x)$.

Globally, the procedure for collecting all the elements needed to set up the LALR(1) parsing table consists in the following steps.

State	*Items (kernel in purple)*	*Eqs*
P_0 :	$[S' \to \cdot S, \{x_0\}]$	$x_0 \doteq \{\$\}$
	$[S \to \cdot L = R, \{x_0\}]$	
	$[S \to \cdot R, \{x_0\}]$	
	$[L \to \cdot * R, \{=, x_0\}]$	
	$[L \to \cdot \text{id}, \{=, x_0\}]$	
	$[R \to \cdot L, \{x_0\}]$	
P_1 :	$[S' \to S \cdot, \{x_1\}]$	$x_1 \doteq \{x_0\}$
P_2 :	$[S \to L \cdot = R, \{x_2\}]$	$x_2 \doteq \{x_0\}$
	$[R \to L \cdot, \{x_3\}]$	$x_3 \doteq \{x_0\}$
P_3 :	$[S \to R \cdot, \{x_4\}]$	$x_4 \doteq \{x_0\}$
P_4 :	$[L \to * \cdot R, \{x_5\}]$	$x_5 \doteq \{=, x_0\} \cup \{x_5\} \cup \{x_7\}$
	$[R \to \cdot L, \{x_5\}]$	
	$[L \to \cdot * R, \{x_5\}]$	
	$[L \to \cdot \text{id}, \{x_5\}]$	
P_5 :	$[L \to \text{id} \cdot, \{x_6\}]$	$x_6 \doteq \{=, x_0\} \cup \{x_5\} \cup \{x_7\}$
P_6 :	$[S \to L = \cdot R, \{x_7\}]$	$x_7 \doteq \{x_2\}$
	$[R \to \cdot L, \{x_7\}]$	
	$[L \to \cdot * R, \{x_7\}]$	
	$[L \to \cdot \text{id}, \{x_7\}]$	
P_7 :	$[L \to *R \cdot, \{x_8\}]$	$x_8 \doteq \{x_5\}$
P_8 :	$[R \to L \cdot, \{x_9\}]$	$x_9 \doteq \{x_5\} \cup \{x_7\}$
P_9 :	$[S \to L = R \cdot, \{x_{10}\}]$	$x_{10} \doteq \{x_7\}$

Figure 3: Symbolic automaton for $\mathcal{G}_1$: content of states, and of *Eqs*

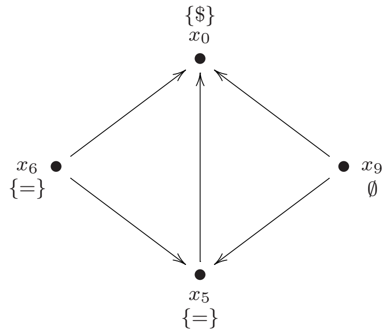

Figure 4: Dependency graph for the computation of the actual values of x_0, x_5, x_6, and x_9

1. Construct the symbolic automaton applying Alg. 4, and get the set *Vars* of variables generated for the construction, and the list *Eqs* of equations installed for those variables.

2. Partition the variables in *Vars* into equivalence classes, so that, instead of computing the actual values of all the variables, it is sufficient to evaluate one variable per class.

   To get the partition, run Alg. 5. It returns a reduced system of equations *REqs* over the variables in the subset *RVars* of *Vars*. Also, Alg. 5 associates with every $x \in \mathit{Vars}$ a class representative, denoted by $\mathit{class}(x)$.

3. Resolve the reduced system of equations *REqs* for the variables in *RVars*.

   To do that, first set up a graph G whose nodes represent the variables in *RVars*. If *REqs* contains the equation $x \doteq \Delta$, then the node for x in G is associated with the inital value $\mathit{init}(x) = \mathrm{ground}(\Delta)$. Also, the node for x has an outgoing edge to each of the nodes representing the variables in Δ.

   Then, run Alg. 6 on G to compute $\mathit{val}(x)$ for all the variables in *RVars*. For every $x \in \mathit{Vars} \setminus \mathit{RVars}$, set $\mathit{val}(x) = \mathit{val}(\mathit{class}(x))$.

   Alg. 6 is a depth-first search algorithm [11] for the computation of the reflexive and transitive closure of relations [7, 5]. The values associated with the farthest nodes are accumulated with the values of the nodes found along the way back to the origin of the path. The visit is organized in such a way that strongly connected components, if any, are recognized on-the-fly and traversed only once.

The application of Alg. 4 to $\mathcal{G}_1$ results in a symbolic automaton with the same layout as that of the LR(0)-automaton in Fig. 1. The content of the states of the symbolic automaton, and the associated system of equations *Eqs* are reported in Fig. 3.

Running Alg. 5 for the equations of the symbolic automaton for $\mathcal{G}_1$ results in the reduced system shown below.

Eqs	$\mathit{class}(x)$	*RVars*	*REqs*
$x_0 \doteq \{\$\}$	x_0	x_0	$x_0 \doteq \{\$\}$
$x_1 \doteq \{x_0\}$	x_0		
$x_2 \doteq \{x_0\}$	x_0		
$x_3 \doteq \{x_0\}$	x_0		
$x_4 \doteq \{x_0\}$	x_0		
$x_5 \doteq \{=, x_0, x_5, x_7\}$	x_5	x_5	$x_5 \doteq \{=, x_0\}$
$x_6 \doteq \{=, x_0, x_5, x_7\}$	x_6	x_6	$x_6 \doteq \{=, x_0, x_5\}$
$x_7 \doteq \{x_2\}$	x_0		
$x_8 \doteq \{x_5\}$	x_5		
$x_9 \doteq \{x_5, x_7\}$	x_9	x_9	$x_9 \doteq \{x_0, x_5\}$
$x_{10} \doteq \{x_7\}$	x_0		

The actual values of variables are obtained by applying Alg. 6 to the graph shown in Fig. 4. Specifically, $\mathit{val}(x_0) = \mathit{val}(x_1) = \mathit{val}(x_2) = \mathit{val}(x_3) = \mathit{val}(x_4) = \mathit{val}(x_7) = \mathit{val}(x_{10}) = \{\$\}$, and $\mathit{val}(x_5) = \mathit{val}(x_6) = \mathit{val}(x_8) = \mathit{val}(x_9) = \{=, \$\}$.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[3] Frank DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, MIT, Cambridge, Mass., 1969.

[4] Frank DeRemer. Simple LR(k) Grammars. *Commun. ACM*, 14(7):453–460, 1971. URL: `http://doi.acm.org/10.1145/362619.362625`.

[5] Frank DeRemer and Thomas J. Pennello. Efficient Computation of LALR(1) Look-Ahead Sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, 1982. URL: `http://doi.acm.org/10.1145/69622.357187`.

[6] Charles Donnelly and Richard Stallman. Bison: The Yacc-compatible Parser Generator (Ver. 3.0.4). 2015. URL: `http://www.gnu.org/software/bison/manual/bison.pdf`.

[7] J. Eve and Reino Kurki-Suonio. On Computing the Transitive Closure of a Relation. *Acta Inf.*, 8:303–314, 1977. URL: `http://dx.doi.org/10.1007/BF00271339`.

[8] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Tech. Rep. CSTR 32, Bell Laboratories, Murray Hill, N.J., 1974. URL: `http://dinosaur.compilertools.net/`.

[9] Donald E. Knuth. On the Translation of Languages from Left to Rigth. *Information and Control*, 8(6):607–639, 1965. URL: `http://dx.doi.org/10.1016/S0019-9958(65)90426-2`.

[10] Paola Quaglia. Symbolic Lookaheads for Bottom-up Parsing. In *Proc. 41st Int. Symposium on Mathematical Foundations of Computer Science, MFCS 2016*, volume 58 of *LIPIcs*, pages 79:1–79:13, 2016. URL: `http://dx.doi.org/10.4230/LIPIcs.MFCS.2016.79`.

[11] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. URL: `http://dx.doi.org/10.1137/0201010`.