AUTOMATON-BASED PARSING FOR LEXICALIZED GRAMMARS

Roger Evans
University of Brighton
Roger.Evans@itri.brighton.ac.uk

David Weir
University of Sussex
David.Weir@cogs.susx.ac.uk

Abstract

In wide-coverage lexicalized grammars many of the elementary structures have substructures in common. This means that during parsing some of the computation associated with different structures is duplicated. This paper explores ways in which the grammar can be precompiled into finite state automata so that some of this shared structure results in shared computation at run-time.

1 Introduction

This paper investigates grammar precompilation techniques aimed at improving the parsing performance of lexicalized grammars. In a wide-coverage lexicalized grammar, such as the XTAG grammar (XTAG-Group, 1995), many of the elementary structures[1] have substructures in common. If such structures are viewed as independent by a parsing algorithm, the computation associated with their shared structure may be duplicated. This paper explores ways in which the grammar can be precompiled so that some of this shared structure results in shared computation at run-time.

We assume as a starting point a conventional kind of parser for lexicalized grammars (such as Vijay-Shanker and Joshi (1985), Schabes (1990), Vijay-Shanker and Weir (1993) for LTAG), although the techniques described here are not restricted to the specific details of any particular parsing algorithm. Our approach is based on the observation that the process of traversing an elementary structure (ES) during parsing can be expressed as a finite state automaton (FSA), and hence that the whole parsing process can be viewed in terms of interacting FSA's[2]. By doing this, it becomes possible to apply standard FSA optimisation techniques (such as determinisation and minimisation) to sets of ES's, and to vary control strategy on a per-ES basis to achieve optimum results.

We shall begin by describing how an ES can be mapped onto an automaton, and how a conventional parsing algorithm can be adapted to work with such automata. Then, using a family of related ES's we shall give two examples of optimisation of the processing of ES's through a deterministic merging of their corresponding automata, and show how different control strategies yield different optimisations. We will illustrate the approach using Lexicalized Tree Adjoining Grammars (Joshi and Schabes, 1991), but we note that the techniques described here can also be applied to other lexicalized grammar formalisms[3].

2 Automaton-based parsing of LTAG

Generally speaking, parsing algorithms for LTAG have the property that a traversal is made through the nodes of elementary trees (the 'elementary structures' of an LTAG grammar). In bottom-up parsers, for example, this

[1] We use the term *elementary structures* to mean the basic components of grammatical description. In LTAG (Joshi and Schabes, 1991), for example, the elementary structures are initial and auxiliary trees. A grammar consists of a finite set of elementary structures that are built into derived structures using the composition operations of the formalism.

[2] Cf. Alshawi (1996) for a similar approach. but with different objectives and a different style of formalism. The approach we present here is also reminiscent of LR parsing in that the FSA used by an LR-parser to recognize viable prefixes can be viewed as the result of determinizing a nondeterministic finite state automaton constructed by linking with ϵ-transitions deterministic automata for the individual productions.

[3] The second author is currently involved in the development of a wide-coverage grammar and automaton-based parser using the formalism described in Rambow, Vijay-Shanker, and Weir (1995).

involves traversing the trees from the anchor of the tree up to its root as more and more of the input is spanned to the right and left of the anchor. Hence, the recognition of an elementary tree can be seen to involve a sequence of computation steps: eg, "find an NP to the right, then find a PP to the right and then find an NP to the left". The particular sequence of steps is determined by the tree structure and the labels attached to substitution (and adjunction) nodes. We shall call each of these steps an **elementary computation step** (or ECS) and the sequence corresponding to the processing of an entire elementary tree an **elementary computation**.

Consider, for example, the ditransitive tree given in Figure 1. A bottom-up parser will initially anchor this tree on an actual ditransitive verb in the input and start its traversal at the 'v' node. If a noun phrase is detected to the right of the verb, the first ECS will be to consume it and move to the 'np1' node. A preposition to the right of that will generate a second ECS, moving to the 'p' node, then an additional noun phrase will take it to the 'np2' node. Here the traversal continues to the 'pp' node and then the 'vp' node with no further inputs (and so no elementary computation steps), and so the final ECS consumes a noun phrase to the left, taking the parser to the 'np3' node, and from there straight to the 's' node.

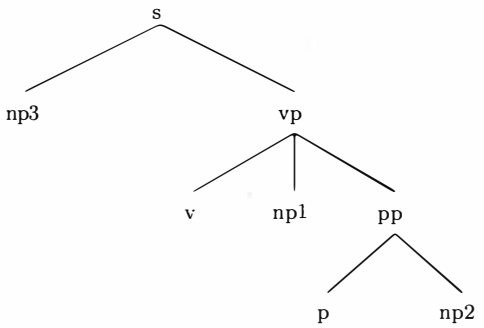


Figure 1: A ditransitive verb tree

This description of the processing clearly invites representation as a finite state automaton, and a possible automaton is given in Figure 2. This automaton has five states, with each transition between states corresponding to an ECS. The input language for the automaton consists of tokens representing parse table states that trigger the ECS ("NP to the left", notated as $\overleftarrow{\text{np}}$, for example). The additional loop attached to the fourth node allows for optional adjunctions at the 'vp' node (notated as $\overleftrightarrow{\text{vp}}$, and triggered by the presence of a complete auxiliary tree enveloping the completed 'vp' subtree). These introduce arbitrarily many additional ECS's without changing the state in the automaton, that is, without traversing the underlying elementary tree at all.

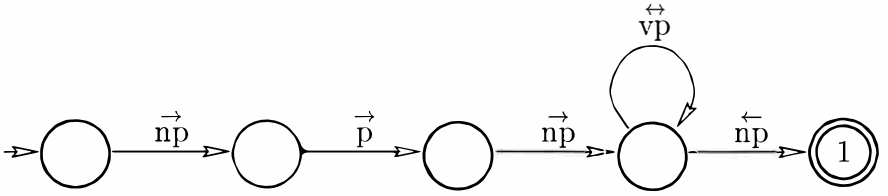


Figure 2: Finite state automaton for the ditransitive verb tree

Automata such as this can be 'executed' by a parser in a manner similar to a conventional tree traversal approach. The parser manipulates a table associating automaton states with segments of the input string (represented as quadruples of indices, or pairs of indices together with a stack of tree addresses – see Vijay-Shanker and Weir (1993)). The table is first populated with initial states for all the automata associated with trees anchored by the words in the input string. Thereafter, elementary computation steps are triggered by the presence of final states in the table. Final states are labelled with the tree they represent (such as the label '1' in Figure 2), and from this the parser can recover the information required to licence a new ECS – the root category of an initial tree (to licence a substitution) or the head and foot node categories of an auxiliary tree (to licence an adjunction). The parser looks for adjacent table entries encoding automata states that are waiting

for this ECS trigger (on the appropriate side). When such an adjacent entry is found, the ECS is executed by passing the trigger as input to the automaton, making a transition to a new state in the automaton, and then creating a new table entry for the new state spanning a suitably enlarged input segment.

The control strategy for identifying and executing ECS's is a detail of the parsing algorithm that need not concern us here. Parsing terminates when no further ECS's can be generated. At that point any table entries containing a final state of some automaton and spanning the entire input correspond to parses. Reading off a parse tree may involve further work, however. Notice that the automaton in Figure 2 does not directly encode all the structure of the tree, and could in fact correspond to a number of well-formed trees. But the tree label associated with its final state does provide the additional information we need to derive a tree from the parse table[4].

As presented so far, this transformation from trees to automata is straightforward and we will not dwell on the details of its application to any specific parsing algorithm here. Instead we will focus on the potential for exploiting the automaton-based representation more fully. For although the transformation is straightforward it is not completely trivial: there is a significant difference between the tree-based and the automaton-based parsers, and it is a difference in the distribution of *control* information. In the tree-based parser, the trees are passive data structures which are traversed in an order specified by the parsing algorithm. In the automaton-based parser, some of this control is vested in the automata themselves, rather than the parsing algorithm. The parser still controls the overall order of execution of ECS's, but the effect of an ECS on a particular automaton is determined by the automaton, not by the parsing algorithm.

There are two consequences of this that we will pursue in the rest of this paper:

- The parser can operate on any automata over the ECS input language – it does not depend on one particular mapping from trees to individual automata. This means we can use standard techniques for merging automata for several trees together into optimal combined representations, resulting in automata that implement several trees at once. This results in greater parsing efficiency, without having to change the parser or the grammar.

- Each automaton encodes a control strategy for traversing the corresponding tree (or set of trees). In the example above we chose one particular traversal, but others are also possible. There is no requirement that the same strategy be used in all the automata, since the parser just follows the automata specifications blindly. So we can consider different strategies giving different optimisations for different parts of the grammar.

3 Some Examples

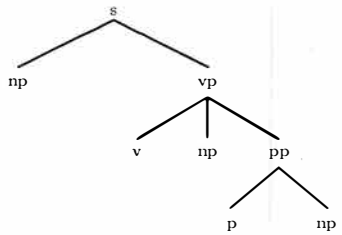
In this section we shall explore these ideas in more detail, by looking at a particular set of related trees, and some of the automata that can be derived from them. The trees we shall use are all potentially anchored by the same surface word so it is likely that they would all need to be considered together during a parse. We will describe automata corresponding to two different traversals of these trees, and show the different results that are obtained by determinising the union of all the automata in each case.

3.1 The Trees

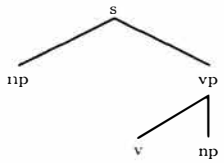
The set of trees we shall consider are a selection from the ditransitive verb family, including the basic ditransitive itself and some (but by no means all) of its regular syntactic and semantic alternations. One feature of these trees is that they can all be anchored by the same word (a present participle or gerund form, such as 'giving'), so that a parser may need to consider all of them simultaneously during parsing.

Tree 1 is the basic ditransitive form itself, which we take to include a noun phrase and a prepositional phrase. We assume here that 'v' is the lexical anchor and that adjunction can only occur at the 'vp' node. Note that although this tree is potentially complete for finite verb forms, our example sentence uses a present participle, which requires the adjunction of an auxiliary verb to form a valid sentence. In our example sentences, we shall indicate such additional words, not contained in the actual tree, but putting them in parentheses. Other adjunctions (of adverbial modifiers etc.) are also possible. Trees 2 and 3 are the regular transitive and intransitive alternants that any ditransitive verb has (where the missing complements are left unspecified).

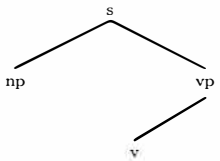
[4]For a simple recognizer, it is sufficient to label the final states with the category information the parser actually requires to identify ECS's – see also Section 3.2 below.



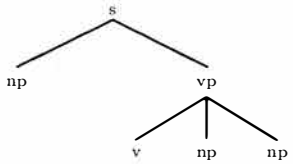
Tree 1: Ditransitive
Mary (is) giving presents to the children



Tree 2: Transitive
Mary (is) giving presents



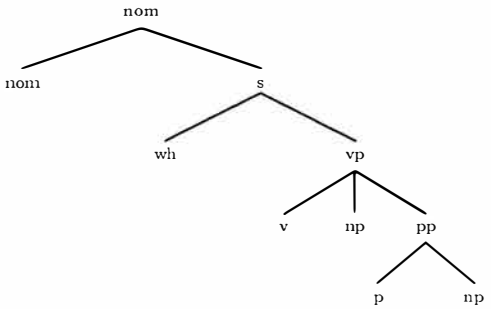
Tree 3: Intransitive
Mary (is) giving



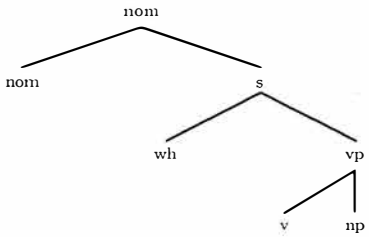
Tree 4: Dative
Mary (is) giving the children presents

Tree 4 assumes that we are in the specific subclass of ditransitives that can undergo dative movement, and provides the appropriate double-np subcategorisation for it.

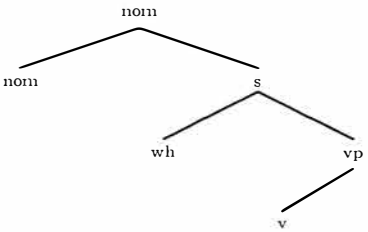
Trees 5–8 represent the wh-relative versions of Trees 1–4. Note that these are in fact auxiliary trees which can be adjoined into noun phrases. As well as having extra structure to support this, the subject position is marked as a 'wh' form. As before, adjunctions can take place at the 'vp' node.



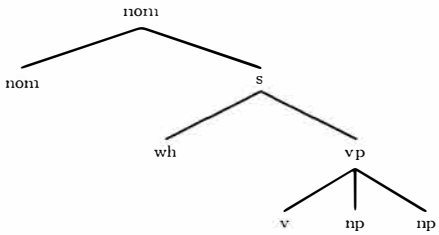
Tree 5: WH-ditransitive
(someone) who (is) giving presents to the children



Tree 6: WH-transitive
(someone) who (is) giving presents



Tree 7: WH-intransitive
(someone) who (is) giving

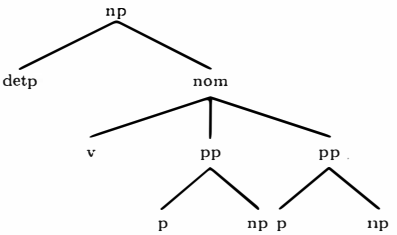


Tree 8: WH-dative
(someone) who (is) giving the children presents

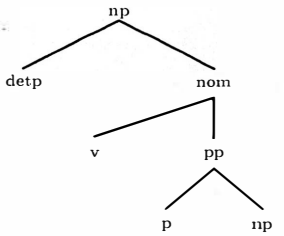
Trees 9–11 provide nominal variants of the ditransitive, transitive and intransitive forms, using a gerund. Here, the object noun phrase (when present) has become a prepositional phrase, and the trees represent a noun-phrase with a determiner rather than a sentence with a subject. Adjunction can only occur at the 'nom' node[5].

In all we have eleven trees, all anchored in the same word, and all clearly duplicating structure. There are many other trees in this family including those for topicalized sentences and wh-sentences. By mapping these

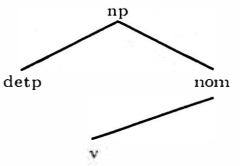
[5]This is not the only possible analysis of gerunds of course, but it is adequate for present illustrative purposes.



Tree 9: Ditransitive gerund
the giving of presents to the children



Tree 10: Transitive gerund
the giving of presents



Tree 11: Intransitive gerund
the giving

eleven into automata, we aim to collapse some of this structure using standard automata techniques, and so make parsing of this tree set more efficient. In general there is more than one way to do this, and the following two subsections illustrate two possibilities.

3.2 Bottom-up traversal

Our first example takes these eleven trees and applies the same traversal as we introduced above – a bottom-up traversal that builds complete local trees working from the anchor outwards before moving to higher structure. Figure 3 shows the resulting eleven automata.

The automata transitions are labelled according to the convention for elementary computation steps introduced above. $\overleftarrow{\text{np}}$ denotes substitution of a noun phrase on the left, $\overrightarrow{\text{np}}$ denotes substitution of a noun phrase on the right and $\overleftrightarrow{\text{vp}}$ denotes adjunction at a verb phrase node. The final states are labelled with the number of the tree that they complete.

The first step in optimising these automata is to union them into a single (nondeterministic) automaton by introducing a new initial state and ϵ-transitions from it to the start states of each of the above automata. Once this has been achieved, we can then reduce and determinize the resulting automaton, and this gives us the automaton in Figure 4. Here, as well as the labels on the final states, we have labelled non-final states with the list of tree numbers (in italic script) whose EC passes through that state. This is not relevant to the parser, but may make the diagram easier to follow.

How does this automaton compare with the original eleven? It is difficult to compare the behaviour of one automaton with that of eleven operating in concert without committing to a particular parsing algorithm. So it may be more useful to compare our determinized automaton with the single union automaton described in the previous paragraph. Even then, it is not clear what metrics might be lead to useful comparisons, but the following three (interrelated) metrics seem to provide interesting insights:

1. the total number of states in the automaton, as a basic indicator of complexity.

2. the average number of trees per state: each state represents a point in the traversal of one or more of the trees. By averaging the number of trees associated with each state, we get a measure of the amount of sharing that has been achieved. In each of the original automata this figure is 1 (and in the union it is close to 1, the additional start state introduces a little sharing), but higher numbers reflect more sharing of structure.

3. the representational expansion factor, that is, the ratio of the number of tree/state combinations compared with the original automata. Larger values for this ratio represent the introduction of *additional* structure not present in the original representation (for example to ensure determinism).

The table in Figure 4 gives values for these metrics for the union automaton and the determinized automaton. We can see from these figures that we have nearly halved the number of states with every state doing work for

two trees on average. The expansion factor reflects the introduction of new states to cope with the optionality of adjunction in a deterministic setting. The third line of results is for a version of the determinized automaton which only performs recognition, included for interest although we have not included the automaton itself. As we noted above, a recognizer need only distinguish the category information associated with final states, not the licencing tree, and this results in additional sharing of structure. In this case the improvement is even more marked – a fourfold improvement in size on the original automata.

3.3 Left-right traversal

The English grammar used in the XTAG system (XTAG-Group, 1995) has the property that foot nodes of auxiliary trees are always at the extreme left or extreme right of the tree. This means that when adjunction takes place all the nodes of the adjoined tree lie to the right or to the left of the node at which adjunction occurred. For grammars with this property we can distinguish between *left adjunction* (where the foot is on the extreme right of the tree) and *right adjunction* (where it is on the extreme left), and assume that these are the only kinds of adjunction that occur, that is, that no *two-sided* adjunctions occur.

The previous example used a traversal of the trees which built complete structures bottom-up before moving to parent nodes. If no two-sided adjunctions are permitted, we can consider a different traversal, which attempts to work left first and then right, independently of local tree structure. Using the same trees and starting at the 'v' anchor node, this time we look first for left adjunctions at the parent 'vp', then for a subject, and only then do we look for complements and finally right adjunctions. For the same eleven trees introduced above, the set of automata resulting from this traversal is given in Figure 5, and the determinized version in Figure 6. We use ↓ vp to denote the left adjunction of a 'vp' auxiliary tree and vp ↓ to denote right adjunction of a 'vp' auxiliary tree. As in Figure 4 we have annotated states with the numbers of trees whose EC passes through that state, but notice that in this case some states are final for one tree, but non-final for others.

The values for the three metrics introduced above are also given in the table in Figure 6. From these figures we see that something slightly different has happened. Although the determinized automata have the same number of states in the two examples, in this second case the amount of sharing is also higher, but this is counterbalanced by the expansion factor. Looking closely at the automaton, we see that there is more effective sharing of state (because the left hand sides of our trees are more similar than the subcategorisation frames, roughly speaking), but we also had to introduce more states to cope with the splitting up of the adjunction into two phases. The figures for the recognizer are still significantly better, but there seems to have been less scope for compression, again due to the need for expansion to cope with the adjunctions.

4 Discussion

We have presented a formulation of lexicalized grammar parsing that lends itself to a range of precompilation optimisations, and given examples of two such precompilations. One issue that we have not yet made clear is exactly which sets of trees should be precompiled in this way. In a lexicalized grammar, the usual distinction between the lexicon and the grammar is reflected in the distinction within the lexicon between the Elementary Structures Database and the Word Database:

- The Elementary Structures Database (ESD) is a set of unanchored ES's (typically several hundred). An unanchored ES is one in which no lexical item has been associated. Collections of related ES's are often organized into families.
- The Word Database (WD) is an association of words (perhaps several hundred thousand) with those elements of the ESD that they can anchor. Typically a word will anchor all the members of a family, and may anchor structures from several families.

This architecture has interesting consequences for our precompilation approach. The set of ES's associated with a given lexical token seems an obvious target for FSA optimisation. However, many tokens share their set of ES's with other tokens, often many other tokens. The distinction between the ESD and the WD allows us to optimize structure in the ESD only, so that these tokens can continue to share the optimized structures. On the other hand some tokens have overlapping but non-identical ES sets. ES's in the intersection may be integrated into two different automata, possibly in different ways, optimized to the particular requirements of each token.

But this apparent duplication of grammatical structure has no processing disadvantage, since only one of the two representations will ever be associated with any given input token (though they may both be active in the parse as a whole).

We saw above that alternative traversals of the underlying elementary structures leads to different automata, and hence different merging of elementary computations. Because the parser operates directly on the automata, different choices of traversal are equally valid, and will result in different parsing strategies. In fact, there is no reason why different parts of a single grammar should not use different traversals. This might be motivated by the linguistic structure (strategies for verbal or nominal subcategorisation might differ from those for coordination, for example), or by the relative sizes of the automata themselves, but a particularly interesting approach would be to try and optimise performance on a statistical basis.

Time is wasted during parsing when locally licensed ECS's are made in situations that cannot lead to complete parses. Let us call these *useless* steps. since the order of ECS's in an elementary computation varies from one traversal to another, it is possible that some traversals will lead to less occurrences of useless steps than others. It should, in principle, be possible to use the information provided by a large parsed corpus to select traversals that minimize the number of useless steps that will arise.

Finally, it is interesting to note the duality between the kind of optimisation proposed here and the more traditional representational optimisation through the expression of linguistic generalisations. We have previously argued (Evans, Gazdar, and Weir, 1995) that lexicalized grammars can benefit from the representational tools developed for lexical representation (such as inheritance hierarchies, exceptions and lexical rules). The present work does not exploit such generalisations at all, but rather assumes a completely flat base grammar. The implicit architecture promoted here views optimisation for parsing as independent of linguistic generalisation. To a large extent this is a consequence of the underlying parsing framework (which also cannot exploit lingusitic generalisations). It would be interesting further work to explore the relationship between these linguistic and parsing optimisations, to see, for example, whether linguistic generalisations might inform the optimisation process, or conversely whether they might arise as emergent properties of parsing optimisations carried out for other reasons, or else to conclude that the two are indeed best treated as independent aspects of language structure.

References

Alshawi, Hiyan. 1996. Head automata and bilingual tilings: Translation with minimal representations. In *34th Meeting of the Association for Computational Linguistics (ACL'96)*, pages 167–176.

Evans, Roger, Gerald Gazdar, and David Weir. 1995. Encoding lexicalized tree adjoining grammars with a nonmonotonic inheritance hierarchy. In *33rd Meeting of the Association for Computational Linguistics (ACL'95)*, pages 77–84.

Joshi, Aravind K. and Yves Schabes. 1991. Tree-adjoining grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Definability and Recognizability of Sets of Trees*. Elsevier.

Rambow, Owen, K. Vijay-Shanker, and David Weir. 1995. D-Tree Grammars. In *33rd Meeting of the Association for Computational Linguistics (ACL'95)*, pages 151–158.

Schabes, Yves. 1990. *Mathematical and Computational Aspects of Lexicalized Grammars*. Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

Vijay-Shanker, K. and Aravind Joshi. 1985. Some computational properties of tree adjoining grammars. In *23rd Meeting of the Association for Computational Linguistics (ACL'85)*, pages 82–93.

Vijay-Shanker, K. and David Weir. 1993. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636.

XTAG-Group, The. 1995. A lexicalized tree adjoining grammar for English. Technical Report IRCS Report 95-03, The Institute for Research in Cognitive Science, University of Pennsylvania.

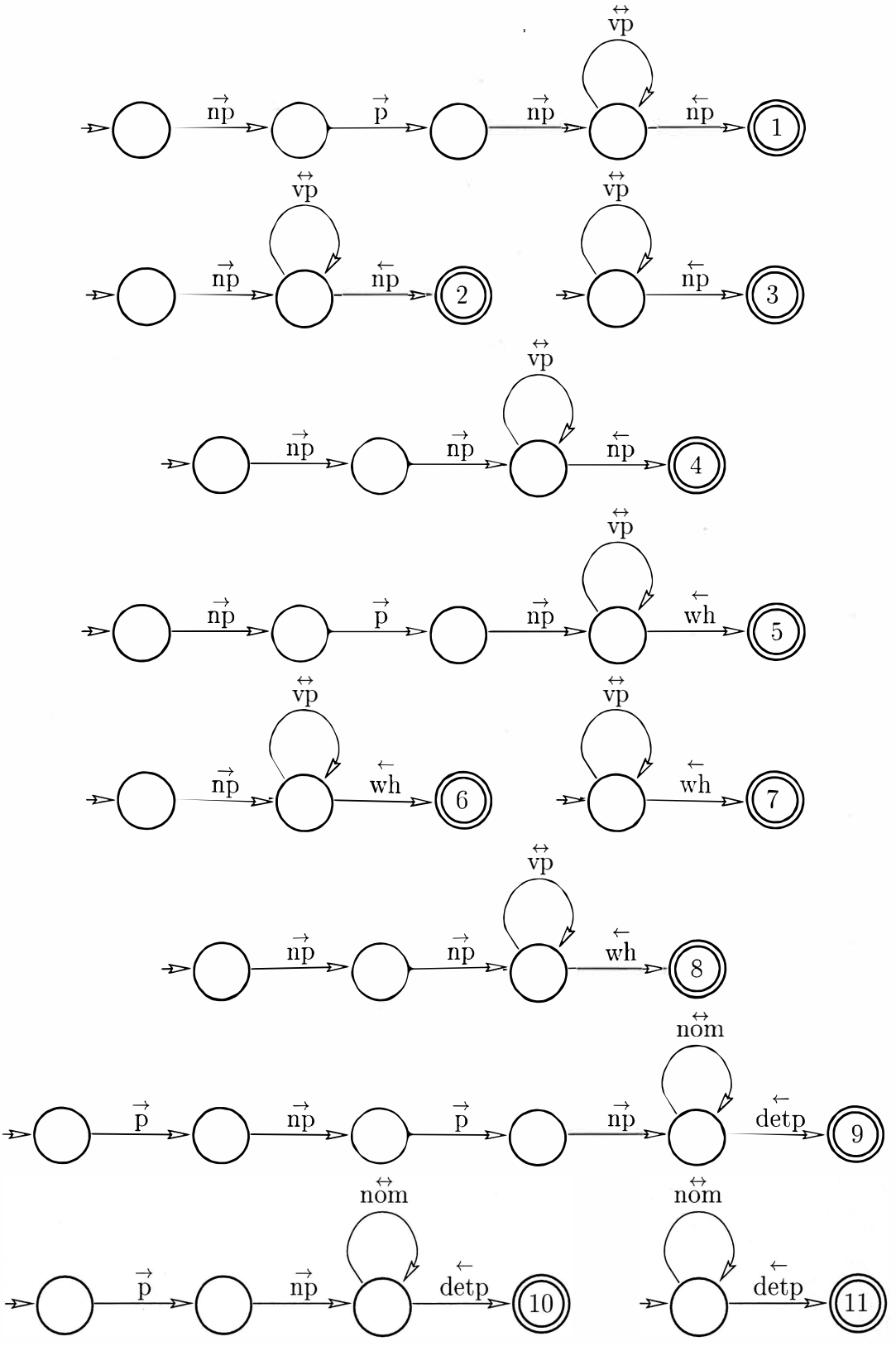
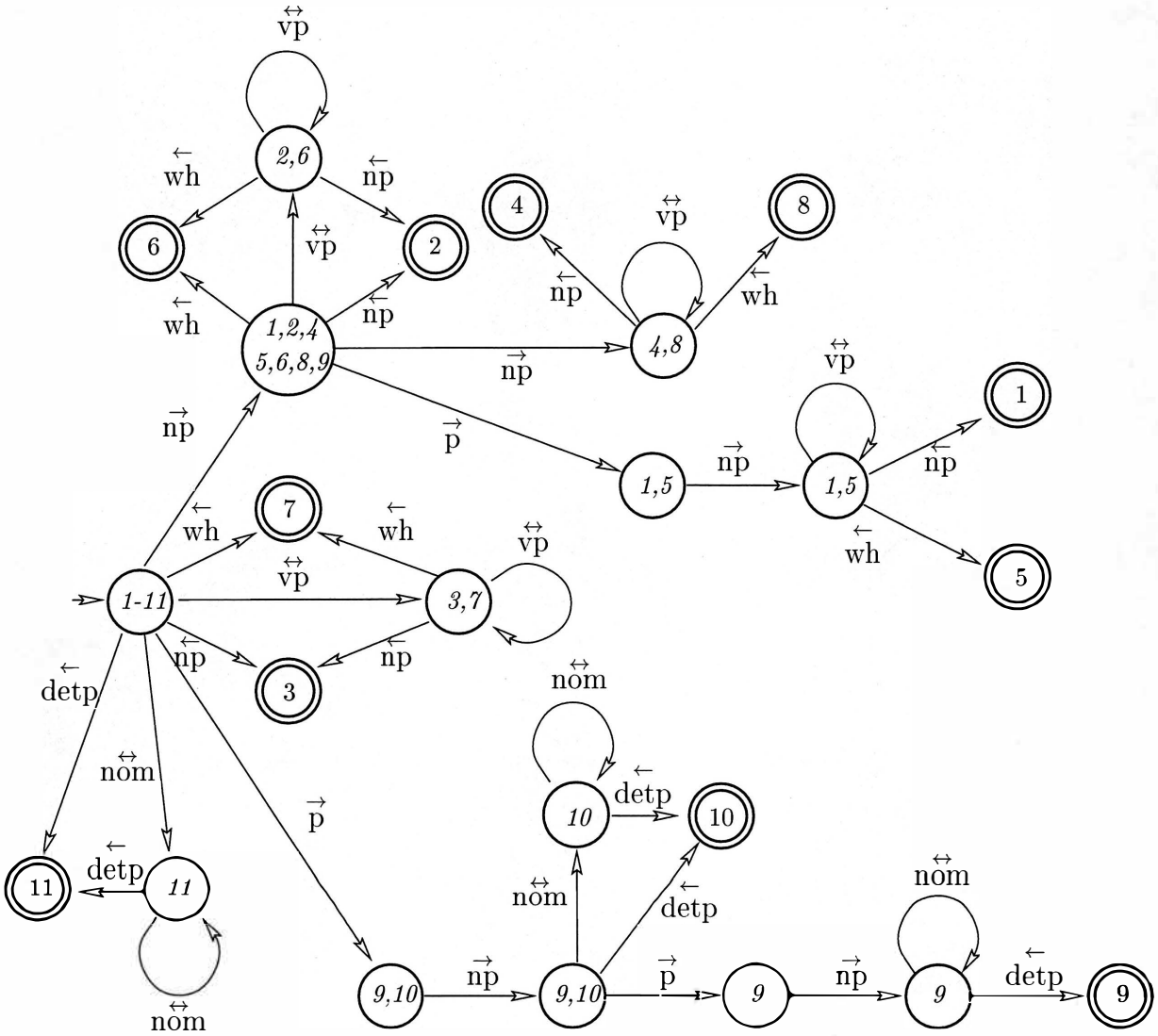


Figure 3: Bottom-up automata for the eleven trees



Metrics for BU Automaton	*States*	*Trees per state*	*Expansion factor*
Union of eleven automata	41	1.244	1.025
Determinized automaton (parser)	24	1.92	1.15
Determinized automaton (recognizer)	11	4	1.1

Figure 4: Bottom-up deterministic automaton for all eleven trees

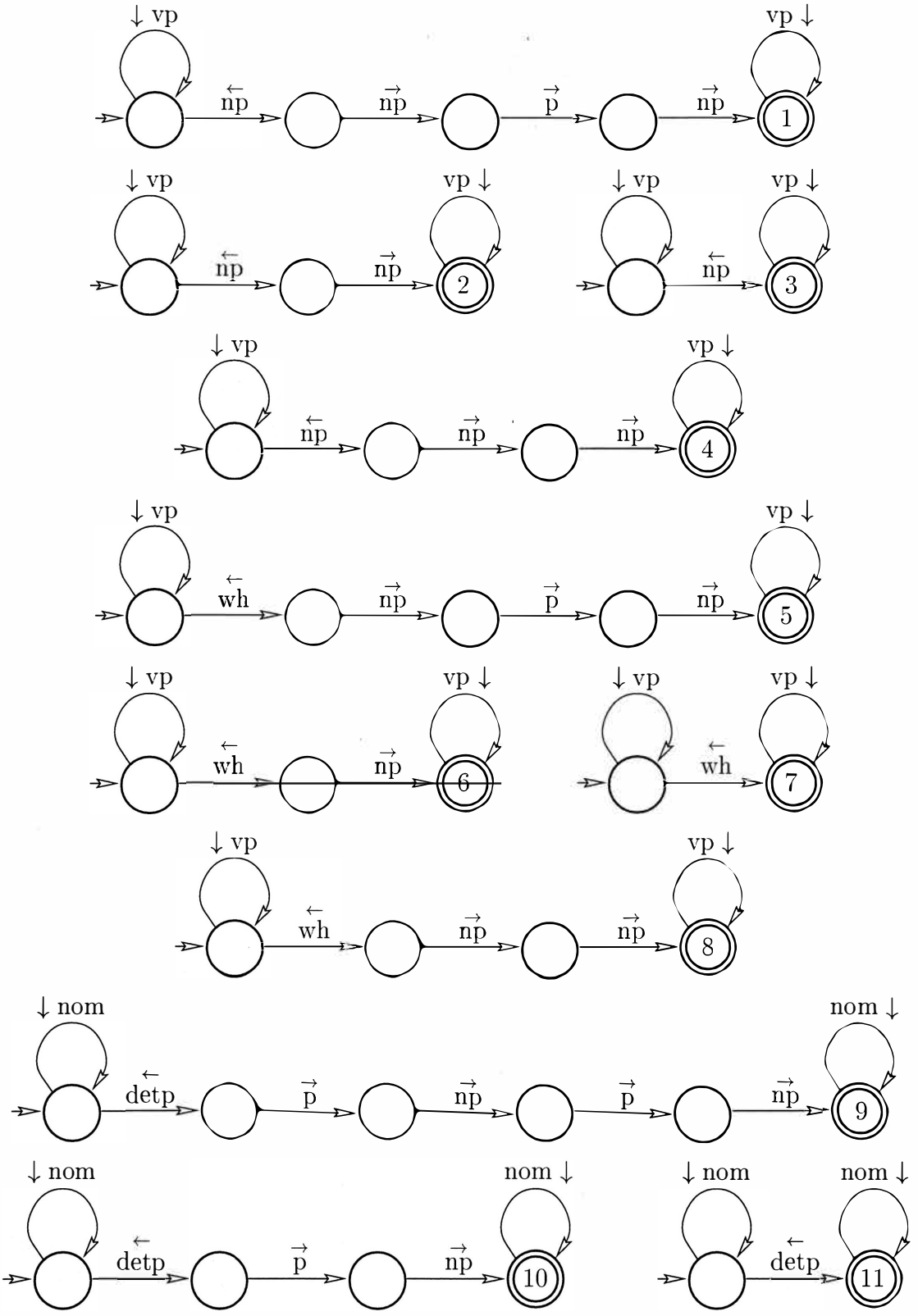
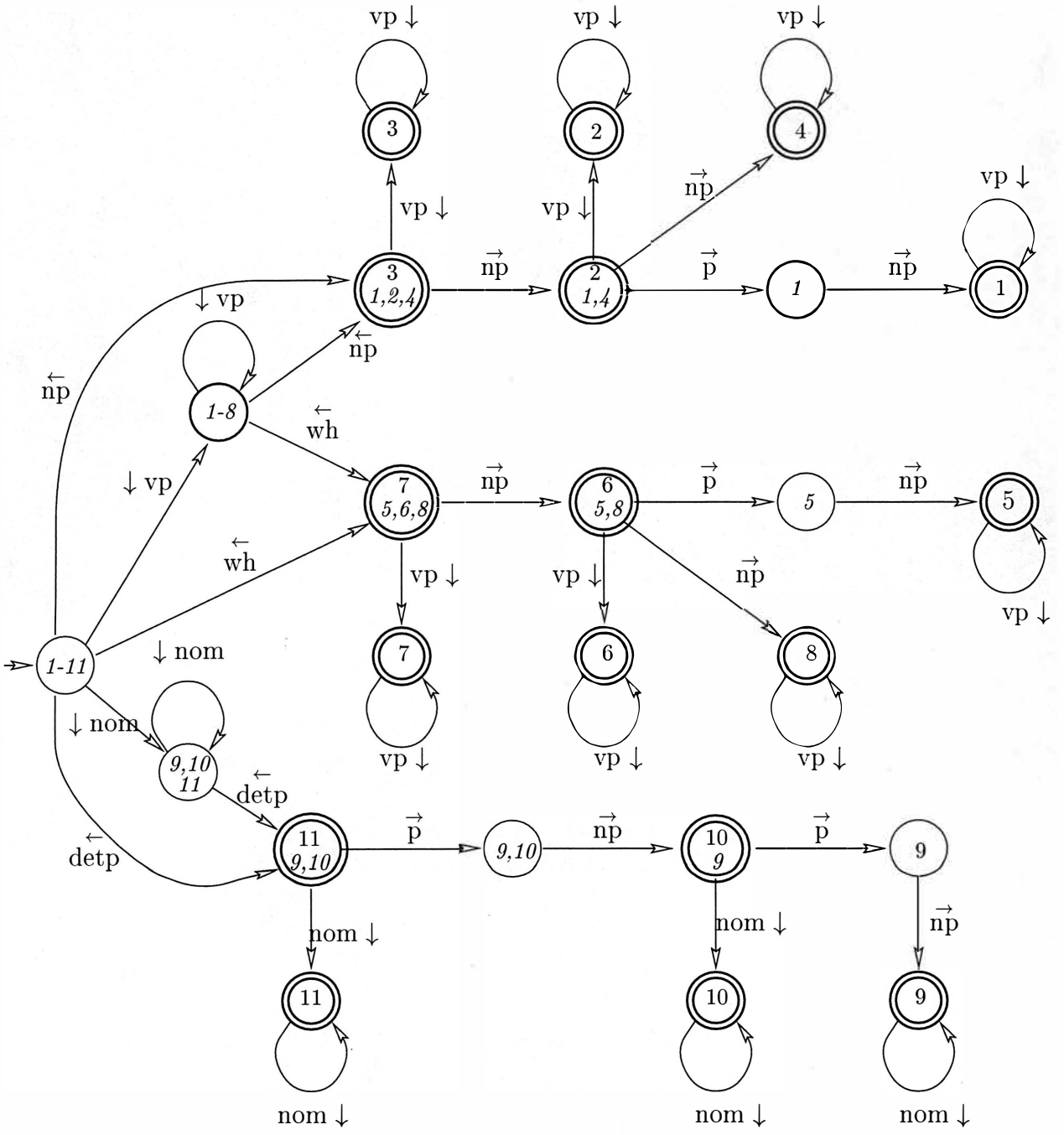


Figure 5: Left-right automata for the eleven trees



Metrics for L-R automaton	*States*	*Trees per state*	*Expansion factor*
Union of eleven automata	41	1.244	1.025
Determinized automaton (parser)	24	2.375	1.425
Determinized automaton (recognizer)	16	3.562	1.425

Figure 6: Left-right deterministic automaton for all eleven trees