

Michael W. Daniels  
May 14, 1999

## On The Extension of UNICORN for Lexical Rule Processing

### 1. Introduction

Feature structure-based grammar formalisms have become increasingly important in recent times. Many current linguistic theories can be expressed in terms of feature structure-based grammars, and they are particularly well-suited for computational implementation. In many of these theories, the lexicon has two parts. The first is a set, often structured in some way, of lexical entries. These lexical entries completely encapsulate the linguistic properties of the lexeme being described – for instance, its pronunciation, meaning, and valence properties. The second component of the lexicon is a set of lexical rules that encapsulate regular correspondences within the lexicon. For instance, a theory might posit one lexical entry for each verb and then use a lexical rule to describe how inflected verb forms relate to uninflected verb forms. These lexical rules must therefore describe the correspondences between all parts of a lexical entry: the phonological descriptions, the orthographic descriptions, the semantic descriptions, and the syntactic descriptions.

The UNICORN parser[2] was created by Dale Gerdemann and Erhard Hinrichs at the University of Illinois as a tool for working with feature-structure grammars and investigating their properties. It represents the lexicon as a simple list of lexical entries, however, and therefore lacks the economy afforded by lexical rules. For instance, separate lexical entries are required for *dog* and *dogs*, *cat* and *cats*, *bird* and *birds*, and any other singular/plural noun pair. For any verb, separate entries are required for the third-person singular, the simple past, the present participle, the past participle, and so on. While this can make the task of parsing more efficient, it slows down grammar development considerably and increases the likelihood of error. It is therefore advantageous for a parsing system to be able to process lexical rules.

This paper describes the changes necessary to add such lexical rule functionality to UNICORN. Section 2 begins by describing the nature and foundations of context-free parsing, on which UNICORN is based. Section 3 describes how Earley's algorithm was adapted to handle feature-structure grammars. In section 4, I show how the relationship between Earley's algorithm and UNICORN's parsing algorithm can be extended to allow UNICORN to process lexical rules. Finally, section 5 describes the architecture necessary to handle morphological correspondences. Appendix A summarizes the sample grammar used to illustrate the parsing algorithms.

### 2. Context-free Parsing

To understand the motivations behind the extensions of UNICORN described here, it is necessary to understand the relationship between UNICORN's parsing algorithm and Earley's algorithm for context-free parsing.

## 2.1 Definitions

A context-free grammar (CFG) is formally defined[3] as a quadruple  $G = \langle N, T, P, S \rangle$ , where  $N$  is a set of non-terminal symbols,  $T$  a set of terminal symbols,  $P$  the set of rules in the grammar (represented as a relation from  $N$  onto  $(N \cup T)^*$ ), and  $S$  the distinct (non-terminal) start symbol (sometimes referred to as the *axiom*).

We write  $\alpha A \gamma \rightarrow \alpha \beta \gamma$  (that is, a given string *directly derives* another) when  $(A, \beta) \in P$  (i.e., when a rule  $A \rightarrow \beta$  exists in the grammar). If a string  $\alpha_1$  directly derives  $\alpha_2$  and  $\alpha_2$  directly derives  $\alpha_3$ , then  $\alpha_1$  is said to derive  $\alpha_3$  in two steps. To say that  $\alpha_1 \Rightarrow \alpha_k$  (“derives”) is to say that there exists some  $k$  for which  $\alpha_1$  derives  $\alpha_k$  in  $k$  steps. Then the *language* generated by  $G$  is the set of all strings  $w \in T^*$  for which  $S \Rightarrow w$ .

## 2.2 Earley’s Algorithm

Earley’s algorithm[1] for context-free parsing (the task of enumerating the rules involved in deriving  $w$  from  $S$ ) is based on the manipulation of *states* and *statesets* (sets of states). Throughout the parsing process, a set of statesets will be maintained, each corresponding to a particular symbol in the input string. The algorithm will process each stateset in turn and add new states to either the current stateset or the next stateset.

A state is an annotated rule that describes the current status of a parse. If  $A \rightarrow \beta$  is a rule in the grammar, then its states will be of the form  $A \rightarrow \gamma.\delta, n$ , where  $\gamma$  refers to the recognized part of the string and  $\delta$  to the as-yet unrecognized portion. In all cases,  $\beta$  must be the concatenation of  $\gamma$  with  $\delta$ . The integer  $n$  represents the *origin* stateset: the stateset from which the state was generated. If  $\gamma$  is empty, we refer to the state as an *initial state*; if  $\delta$  is empty, we refer to the state as a *final state*.

The input to Earley’s algorithm consists of two parts: a grammar (as described above) and an input string. To begin, the algorithm adds a new rule  $S' \rightarrow S$  (where  $S$  is the original start symbol) to the grammar and sets  $S'$  as the start symbol for the grammar. This ensures that the starting symbol appears on the left-hand side of only one rule, avoiding the premature termination that would otherwise occur upon recognizing an embedded  $S$ . The algorithm then initializes the first stateset with the state  $S' \rightarrow .S, 0$  and appends an explicit end marker (here,  $\$$ ) to the input string.

Then, starting with the newly-added state, the algorithm examines each state in turn. Based on the nature of the symbol immediately to the right of the dot (the *active* symbol), the algorithm applies one of three procedures to the current state, adding additional states to either the current or future statesets—at no point will states ever be added to an already-processed stateset. These three procedures are the predictor, scanner, and completer. Each takes a state as input and adds new states to various statesets according to

the nature of the input state. The parser decides which procedure applies by examining the active symbol.

If the active symbol is a non-terminal, the *predictor* applies. For every rule whose left hand side is identical to the active symbol, the predictor adds the initial state corresponding to that rule to the current state set. This new state has its origin set to the current stateset.

If the active symbol is a terminal, the *scanner* applies. It compares the current stateset's input token to the active symbol in the state. If they are identical, a new state is created in the next stateset with the dot advanced one position and the origin unchanged. If they differ, we simply move to the next state.

If there is no active symbol (and this is therefore a final state), the *completer* applies. For each state in the origin whose active symbol is the left hand side of the current state, the completer creates a new state in the current stateset with the dot advanced one position and the origin unchanged. If the left-hand symbol in the rule is  $S'$  (the new start symbol) and the current stateset's input token is the end marker, then the completer has been applied to the initial rule and the string is successfully parsed. Otherwise, execution continues normally.

After states have been added, the next state in the current stateset is processed. This continues until all the states in the current stateset have been processed, at which point the next stateset is processed. If at any point the states in the current stateset have been exhausted, and the next stateset is empty, the parse fails.

The execution trace in (1) summarizes the following illustration of how Earley's algorithm parses the string  $xyx$  with the rules in (2).

	$I_o$	state	created by
Stateset 0 — Input: x			
0	0	$S' \rightarrow . S$	initial
1	0	$S \rightarrow . S A$	predict(0,1)
2	0	$S \rightarrow . x$	predict(0,2)
Stateset 1 — Input: y			
3	0	$S \rightarrow x .$	scan(2)
4	0	$S' \rightarrow S .$	complete(3,0)
5	0	$S \rightarrow S . A$	complete(3,1)
6	1	$A \rightarrow . y$	predict(5,4)
7	1	$A \rightarrow . A S$	predict(5,3)
Stateset 2 — Input: x			
8	1	$A \rightarrow y .$	scan(6)
9	0	$S \rightarrow S A .$	complete(8,5)
10	1	$A \rightarrow A . S$	complete(8,7)
11	0	$S' \rightarrow S .$	complete(9,0)
12	0	$S \rightarrow S . A$	complete(9,1)
13	2	$S \rightarrow . x$	predict(10,2)
14	2	$S \rightarrow . S A$	predict(10,1)
15	2	$A \rightarrow . y$	predict(12,4)
16	2	$A \rightarrow . A S$	predict(12,3)
Stateset 3 — Input: \$			
17	2	$S \rightarrow x .$	scan(13)
18	1	$A \rightarrow A S .$	complete(17,10)
19	2	$S \rightarrow S . A$	complete(17,14)
20	0	$S \rightarrow S A .$	complete(18,5)
21	1	$A \rightarrow A . S$	complete(18,7)
22	3	$A \rightarrow . y$	predict(19,4)
23	3	$A \rightarrow . A S$	predict(19,3)
24	0	$S' \rightarrow S .$	complete(20,0)
25	0	$S \rightarrow S . A$	complete(20,2)
26	3	$S \rightarrow . x$	predict(21,2)
27	3	$S \rightarrow . S A$	predict(21,1)

(1)

0 :  $S' \rightarrow S$ 1 :  $S \rightarrow S A$ 2 :  $S \rightarrow x$ 3 :  $A \rightarrow A S$ 4 :  $A \rightarrow y$ 

(2)

Line 0 is the initial state. Since the non-terminal  $S$  is to the right of the dot, the predictor applies, and lines 1 and 2 (corresponding to rules 1 and 2) are added to stateset 0 with origin 0. The predictor also applies to line 1, but the states it would add are identical to lines 1 and 2, so no action is taken. For line 2, the terminal  $x$  is active and the scanner applies. Since the active symbol  $x$  matches the current stateset's input, line 3 is added to stateset 1 with origin 0.

The completer applies to line 3, since there is no active symbol (the dot is at the end of the rule). Since line 3 has origin 0, the completer examines stateset 0 for rules where the symbol  $S$  (the left-hand side of line 3) is the active symbol. In this case, lines 0 and 1 match this condition, and therefore lines 4 and 5 are added to the current stateset. Line 4 represents a possible completion of the initial state, as its left-hand symbol is  $S'$ . Since the current stateset's input is not the end marker  $\$,$  this state cannot be completed. When line 5 is processed, the scanner adds lines 6 and 7. Here, the origin is set to 1, the value of the current stateset. Lines 6 through 23 are processed similarly.

At line 24, the completer is again applied to the initial state. At this point, the current stateset's input is the end marker and the algorithm terminates, indicating a successful recognition.

While Earley's algorithm has been presented as a mechanism for parsing context-free grammars, it may also be extended to handle more complex types of grammars.

### 3. Feature-structure Parsing

#### 3.1 Definitions

In general, unification grammars are based on relations between *feature structures*. A feature structure is a directed acyclic graph (DAG) whose arcs are labelled with *attributes* and whose nodes are labelled with *values*. Figure 3 depicts a sample feature structure.



Formally, such a DAG is defined[2] as a sextuple  $\langle N, L, A, \delta, \alpha, r \rangle$  where  $N$  is a set of *nodes*,  $L$  a set of *attributes* (that is, arc labels),  $A$  a set of *atoms*,  $\delta$  the connection function that maps node-attribute pairs onto destination nodes,  $\alpha$  the valuation function that maps nodes onto atoms (representing the arcs in the DAG), and  $r$  the designated *root* node. In addition, there must be a path from the root node to every other node in the graph.

It is common to represent feature structures by an *attribute-value matrix* (AVM). Figure (4) depicts the AVM corresponding to (3).

$$\left[ \begin{array}{l} \mathbf{x0} \\ \mathbf{x1} \end{array} \left[ \begin{array}{l} \mathbf{maj} \quad \boxed{1} \ v \\ \mathbf{syn} \quad \boxed{2} \ \mathbf{sg} \\ \mathbf{maj} \quad \boxed{1} \\ \mathbf{syn} \quad \boxed{2} \end{array} \right] \right] \quad (4)$$

Here, the boxed 1 represents the fact that both `maj` arcs point to the same node; the boxed 2 similarly represents the sharing of a node by both `syn` arcs. This notation can also be used to describe classes of feature structures by leaving some or all of the values unspecified. For instance, (5) represents the class of feature structures that include (3) as well as those where, for instance, both `syn` attributes have the value `pl`.

$$\left[ \begin{array}{l} \mathbf{x0} \\ \mathbf{x1} \end{array} \left[ \begin{array}{l} \mathbf{maj} \quad \boxed{1} \ v \\ \mathbf{syn} \quad \boxed{2} \\ \mathbf{maj} \quad \boxed{1} \\ \mathbf{syn} \quad \boxed{2} \end{array} \right] \right] \quad (5)$$

### 3.2 Augmenting Earley's Algorithm

A feature structure grammar, then, has two parts. The first is a set of feature structures that form the rules of the grammar, describing how a composite linguistic entity is related to the entities corresponding to its components. UNICORN represents these as CFG rules augmented so that feature structures have taken the place of terminals and non-terminals. They have the general form of (3): the  $X_0$  attribute's value is the parent feature structure, corresponding to the left-hand side of a CFG rule, and the remaining attributes' values are the feature structures for the daughters, corresponding to the right-side of a CFG rule.

The second component of a feature structure grammar is a set of feature structures representing lexical entries. These typically have one attribute whose value can be matched against the input (i.e. some sort of phonological or orthographical representation) and other attributes that describe the lexical item itself, depending on the intended use of the grammar and the underlying linguistic theory.

Since the feature-structure grammar is represented as an augmentation of a context-free grammar, it is possible to construct a feature structure parsing algorithm from a context-free parsing algorithm; indeed, this is the approach Gerdemann and Hinrich took in adapting Earley's algorithm. The nature of this construction is a direct consequence of the replacement of atomic symbols with complex feature structures: for context-free parsing, the tests for state identity and symbol identity are trivial operations, requiring a single comparison. For feature-structure parsing, however, the analogous comparisons are more complex.

The predictor uses a state identity test to avoid adding redundant states to statesets. When an identical state already exists in that stateset (as in line 1 in (1)), it does not add that state to the stateset again. With a feature structure grammar, we need to test for *subsumption* instead. In essence, feature structure  $A$  subsumes another feature structure  $B$  when  $A$  and  $B$  do not conflict in any way and  $A$  is more general than  $B$ . The feature-structure predictor, then, will not add a state to a stateset when a subsuming DAG is already present in that stateset.

Symbol identity tests are used by all three of the state-adding procedures. The predictor looks for rules in the grammar whose left-hand symbol matches the active non-terminal in the current state. The scanner checks to see that the active terminal matches the active input symbol. The completer looks for states in the origin stateset whose active symbols match the left-hand symbol of the current state. In all these cases, we need to replace this identity test with *unification*. In essence, the unification of two feature structures  $A$  and  $B$  (written  $A \sqcup B$ ) is the most general feature structure subsumed by both  $A$  and  $B$ . The feature-structure predictor, scanner, and completer, then, will add new states when the active feature structure successfully unifies with a feature structure from other states or the set of rules in the grammar.

### 3.3 Restriction

By replacing state and symbol identity tests with the more complex subsumption and unification tests, the execution time of the algorithm increases considerably. A technique known as *restriction* optimizes the application of the subsumption and unification tests, compensating for this increase.

Restriction uses a function (a *restrictor*) from DAGs onto DAGs to reduce the amount of time spent performing such tests. The tests are performed on the *restricted DAGs* (RDs) first; if the test succeeds, then it is performed on the actual DAGs, but if it fails, there is no need to consider the actual DAGs.

The exact nature of the optimal restrictor depends heavily on the nature of the grammar itself. In a grammar designed for UNICORN, it will consist of a set of *paths*. A path is a connected series of attributes in the DAG (written as a list of attribute labels separated with vertical bars). For example,  $X0|MAJ$  is a path to the node labelled  $v$  in (3).

The output of this kind of restrictor is an RD consisting of the restrictor’s paths and their values. For complex values, we use empty brackets ( $[]$ ) to represent the complex wildcard, a symbol that unifies with any complex value but not with any atomic value. For example, if we apply the restrictor (6) to the sub-DAG  $X_0$  of rule (39) in Appendix A (reproduced here as (7)), we get (8).

$$\langle MAJ, SYN|FORM, SYN|COMPS, SYN|SPR \rangle \tag{6}$$

$$\left[ \begin{array}{l} \text{maj} \\ \text{syn} \\ \text{surf} \end{array} \begin{array}{l} \text{v} \\ \left[ \begin{array}{l} \text{form} \\ \text{agr} \\ \text{comps} \\ \text{spr} \end{array} \right] \\ \text{plurv}(\left[ \right]) \end{array} \begin{array}{l} \\ \left[ \begin{array}{l} \text{finite} \\ \left[ \begin{array}{l} \text{cls} \\ \text{non} \end{array} \right] \\ \left[ \right] \\ \left[ \right] \end{array} \right] \\ \left[ \right] \end{array} \right] \quad (7)$$

$$\left[ \begin{array}{l} \text{maj} \\ \text{syn} \end{array} \begin{array}{l} \text{v} \\ \left[ \begin{array}{l} \text{form} \\ \text{comps} \\ \text{spr} \end{array} \right] \end{array} \begin{array}{l} \\ \left[ \begin{array}{l} \text{base} \\ \left[ \right] \\ \left[ \right] \end{array} \right] \end{array} \right] \quad (8)$$

Unifying two RDs is a linear-time operation, as each RD contains a finite number of paths having atomic values. Since there are a finite number of RDs, restriction has the effect of dividing the infinite feature structure space into a finite number of equivalence classes. Then, as it is impossible for feature structures in different equivalence classes to unify, the algorithm can recognize pairs of DAGs that “obviously” wouldn’t unify without actually performing the unification test.

### 3.4 UNICORN’s parser

The basic algorithm used by UNICORN follows the outline of Earley’s algorithm, maintaining a similar collection of statesets. Each stateset contains, in addition to its states, a list of all RDs that have been used to make predictions in this stateset (the *predictors list*). Each state contains a DAG representing an instantiation of a rule in the grammar, the location of the dot, a pointer to the state’s origin, and two RDs: one acting as a backpointer, set when the state is created, and one as a forward pointer, set when the predictor is applied to the state.

The input to the UNICORN parser consists of a grammar and an input string, as for Earley’s algorithm. The user also provides the parser with a *goal* DAG, representing the desired root of the final parse tree. The initial state is constructed by embedding this goal DAG under the  $X_0$  and  $X_1$  attributes of the initial DAG. For instance, if the goal is  $\left[ \text{maj} \quad \text{s} \right]$ , the initial DAG would be as described in figure 9.

$$\left[ \begin{array}{l} \text{x0} \\ \text{x1} \end{array} \begin{array}{l} \mathbb{I} \\ \mathbb{I} \end{array} \left[ \begin{array}{l} \text{maj} \\ \text{s} \end{array} \right] \right] \quad (9)$$

The backpointer of the initial state is given the distinct value \$ to mark it as initial; the dot is placed before  $X_1$ .

Unlike Earley’s algorithm for context-free grammars, the predictor and scanner will always both be applied to non-final states. For prediction, the parser constructs an RD from the active feature structure. If this RD is subsumed by any RD on the predictors list, processing continues to the next state. Otherwise, this RD is unified with the RD for each rule in the grammar. If they unify, the actual DAGs for the rule and the active feature structure are unified. If this operation succeeds, a state is added to the current stateset whose DAG is the result of the unification. The dot is set to the initial position, the origin to the current stateset and the backpointer to the RD that generated this state. This RD is also added to the prediction list for this stateset and becomes the forward pointer of the state that generated this prediction.

When the scanner is applied, it attempts to unify the active feature structure with each lexical item in the grammar matching the current input word (using the RDs to filter possibilities as before). If the unification succeeds, a new state is created from the result of the unification with the same origin and backpointer as the current state but with the dot advanced one position.

To determine to which stateset this new state must be added, the length of the scanned phrase is added to the current stateset’s index. Thus a three-word phrase scanned in stateset 2 would cause a new state to be added to stateset 5; a zero-length phrase (a trace) would cause a new state to be added to the current stateset. This allows the parser to use the same mechanisms to scan phrasal lexical items (proper names, for example) as it uses to scan individual words.

Finally, the completer is applied whenever the current state is final. For each state in the origin stateset whose forward pointers match the current state’s backpointer (the potential *ancestors*), the completer unifies the  $X_0$  attribute of the current state with the active feature structure in the potential ancestor. For each success, a new state is created from the result of the unification with the dot advanced one position. The new state retains the ancestor’s origin as well as its back- and forward pointers, and is added to the current stateset.

To illustrate this process, some of the states involved in the parse of (10) in the grammar given in appendix A will be shown.

The swimmer splashes the dogs. (10)

Before the parse begins, the parser pre-processes the grammar by applying the restrictor (6) to the  $X_0$  sub-DAG of each rule and to each lexical entry. The resulting RDs are stored for future reference.

Here, the goal has been provided as a verb phrase whose *comps* and *spr* attributes are nil. Figure 11 shows the initial state (the backpointer is represented as \$ to encode the fact that this is the initial state).

$$\left\langle 0, \begin{bmatrix} x0 & \boxed{1} \\ x1 & \boxed{1} \end{bmatrix} \begin{bmatrix} \text{maj} & v \\ \text{syn} & \begin{bmatrix} \text{spr} & \text{nil} \\ \text{comps} & \text{nil} \end{bmatrix} \end{bmatrix} \right\rangle, 1, 0, \$, \begin{bmatrix} \text{maj} & v \\ \text{syn} & \begin{bmatrix} \text{comps} & \text{nil} \\ \text{spr} & \text{nil} \end{bmatrix} \end{bmatrix} \right\rangle \quad (11)$$

In this and the following figures, the individual states are represented with the components in the following order: stateset number, rule feature structure, dot position, originating stateset, backpointer, forward pointer.

In applying the predictor to (11), the parser begins by applying the restrictor (6) to the active feature structure in this state: the  $X_1$  attribute. Since the parser has not yet applied the predictor to any state, the list of predictors for this stateset is empty. The forward pointer of the current state is therefore set to the resulting RD and added to the predictors list. Next, the parser looks for rules in the grammar whose RDs unify with the current forward pointer. In this case, both rules (12) and (13) meet this criterion. Both of these rules are then unified with the active feature structure (by extracting it from the state and embedding it under an  $X_0$  attribute in a new DAG first) to produce states (14) and (15).

$$\left[ \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \begin{bmatrix} \text{maj} & \boxed{4} \\ \text{syn} & \begin{bmatrix} \text{form} & \boxed{2} \\ \text{comps} & \boxed{3} \\ \text{agr} & \boxed{5} \\ \text{spr} & \boxed{6} \end{bmatrix} \\ \text{surf} & \text{append}(\boxed{7}, \boxed{8}) \end{bmatrix} \right] \quad (12)$$

$$\left[ \begin{array}{l} x1 \\ x2 \end{array} \begin{bmatrix} \text{maj} & \boxed{4} \\ \text{syn} & \begin{bmatrix} \text{form} & \boxed{2} \\ \text{comps} & \langle \boxed{1}, \boxed{3} \rangle \\ \text{agr} & \boxed{5} \\ \text{spr} & \boxed{6} \end{bmatrix} \\ \text{surf} & \boxed{7} \end{bmatrix} \right]$$

$$\left[ \begin{array}{l} x2 \\ \end{array} \begin{bmatrix} \boxed{1} & \text{surf} & \boxed{8} \end{bmatrix} \right]$$

$$\left[ \begin{array}{l}
x0 \\
x1 \\
x2
\end{array} \right] = \left[ \begin{array}{l}
\left[ \begin{array}{l}
\text{maj } \boxed{3} \\
\left[ \begin{array}{l}
\text{form } \boxed{2} \\
\text{comps nil} \\
\text{agr } \boxed{4} \\
\text{spr nil}
\end{array} \right] \\
\text{surf } \text{append}(\boxed{5}, \boxed{6})
\end{array} \right] \\
\boxed{1} \left[ \begin{array}{l}
\text{syn } \left[ \begin{array}{l}
\text{agr } \boxed{4} \\
\text{spr nil}
\end{array} \right] \\
\text{surf } \boxed{5}
\end{array} \right] \\
\left[ \begin{array}{l}
\text{maj } \boxed{3} \\
\left[ \begin{array}{l}
\text{form } \boxed{2} \\
\text{comps nil} \\
\text{agr } \boxed{4} \\
\text{spr } \boxed{1}
\end{array} \right] \\
\text{surf } \boxed{6}
\end{array} \right]
\end{array} \right] \quad (13)$$

$$\left\langle 0, \left[ \begin{array}{l}
x0 \\
x1 \\
x2
\end{array} \right] \right\rangle, 1, 0, \left[ \begin{array}{l}
\left[ \begin{array}{l}
\text{maj } \boxed{4} v \\
\left[ \begin{array}{l}
\text{form } \boxed{2} \\
\text{comps } \boxed{3} \text{ nil} \\
\text{agr } \boxed{5} \\
\text{spr } \boxed{6} \text{ nil}
\end{array} \right] \\
\text{surf } \text{append}(\boxed{7}, \boxed{8})
\end{array} \right] \\
\left[ \begin{array}{l}
\text{maj } \boxed{4} \\
\left[ \begin{array}{l}
\text{form } \boxed{2} \\
\text{comps } \langle \boxed{1}, \boxed{3} \rangle \\
\text{agr } \boxed{5} \\
\text{spr } \boxed{6}
\end{array} \right] \\
\text{surf } \boxed{7}
\end{array} \right] \\
\boxed{1} \left[ \text{surf } \boxed{8} \right]
\end{array} \right] \left[ \begin{array}{l}
\left[ \begin{array}{l}
\text{maj } v \\
\left[ \begin{array}{l}
\text{comps nil} \\
\text{spr nil}
\end{array} \right]
\end{array} \right] \\
\left[ \begin{array}{l}
\text{maj } v \\
\left[ \begin{array}{l}
\text{comps } \square \\
\text{spr nil}
\end{array} \right]
\end{array} \right]
\end{array} \right] \right\rangle \quad (14)$$

$$\left\langle 0, \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \left[ \begin{array}{l} \begin{array}{l} \text{maj} \quad \boxed{3} \quad v \\ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \\ \text{spr} \quad \text{nil} \end{array} \\ \text{surf} \quad \text{append}(\boxed{5}, \boxed{6}) \end{array} \\ \begin{array}{l} \boxed{1} \\ \text{surf} \quad \boxed{5} \end{array} \\ \begin{array}{l} \text{maj} \quad \boxed{3} \\ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \\ \text{spr} \quad \boxed{1} \end{array} \\ \text{surf} \quad \boxed{6} \end{array} \right] \right\rangle, 1, 0, \left[ \begin{array}{l} \text{maj} \quad v \\ \text{syn} \quad \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \end{array} \end{array} \right], \left[ \text{syn} \quad \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \end{array} \right] \right\rangle \quad (15)$$

The rule instantiations in states (14) and (15) show that the information from the rule has been combined (through unification) with the information provided by the goal DAG (i.e. that the parent's major category (**maj**) is **v** and that its **spr** and **comps** attributes have a value of **nil**). The parser next applies the scanner to state (11), but none of the lexical entries unify with the current forward pointer. Processing moves on to state (14); here, the predictor produces state (16) from rule (12).

$$\left\langle 0, \left[ \begin{array}{l} x0 \left[ \begin{array}{l} \text{maj} \quad \boxed{4} \ v \\ \text{syn} \left[ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \boxed{3} \langle \boxed{1}, \text{nil} \rangle \\ \text{agr} \quad \boxed{5} \\ \text{spr} \quad \boxed{6} \ \text{nil} \end{array} \right] \\ \text{surf} \ \text{append}(\boxed{7}, \boxed{8}) \end{array} \right] \\ x1 \left[ \begin{array}{l} \text{maj} \quad \boxed{4} \\ \text{syn} \left[ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \langle \boxed{1}, \boxed{3} \rangle \\ \text{agr} \quad \boxed{5} \\ \text{spr} \quad \boxed{6} \end{array} \right] \\ \text{surf} \quad \boxed{7} \end{array} \right] \\ x2 \quad \boxed{1} \left[ \text{surf} \quad \boxed{8} \right] \end{array} \right] , 1, 0, \left[ \begin{array}{l} \text{maj} \quad v \\ \text{syn} \left[ \begin{array}{l} \text{comps} \quad \boxed{\phantom{1}} \\ \text{spr} \quad \text{nil} \end{array} \right] \end{array} \right], \left[ \begin{array}{l} \text{maj} \quad v \\ \text{syn} \left[ \begin{array}{l} \text{comps} \quad \boxed{\phantom{1}} \\ \text{spr} \quad \text{nil} \end{array} \right] \end{array} \right] \right\rangle \quad (16)$$

Note that (14) differs from (16) in that it is an instantiation of (12) based on an earlier prediction, rather than on the goal DAG; here, the only difference is in the value of the parent's `comps` attribute.

The scanner produces no new states when applied to (14). When processing state (15), the predictor uses rule (13) to add state (17) to the current stateset.

$$\left\langle 0, \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \left[ \begin{array}{l} \left[ \begin{array}{l} \text{maj} \quad \boxed{3} \\ \left[ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \\ \text{spr} \quad \text{nil} \end{array} \right] \\ \text{surf} \quad \text{append}(\boxed{5}, \boxed{6}) \end{array} \right] \\ \left[ \begin{array}{l} \boxed{1} \\ \text{syn} \\ \left[ \begin{array}{l} \text{spr} \quad \text{nil} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \end{array} \right] \\ \text{surf} \quad \boxed{5} \end{array} \right] \\ \left[ \begin{array}{l} \text{maj} \quad \boxed{3} \\ \left[ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \\ \text{spr} \quad \boxed{1} \end{array} \right] \\ \text{surf} \quad \boxed{6} \end{array} \right] \end{array} \right] , 1, 0, \left[ \text{syn} \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \end{array} \right] \right], \left[ \text{syn} \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \end{array} \right] \right] \right\rangle \quad (17)$$

State (17) differs from (15) in that the parent's major category is left unspecified. When the scanner is applied to state (15), it successfully unifies the state's forward pointer with the RD for *the* (the current input token). The scanner then unifies the lexical entry for *the* with the current state (embedding the lexical entry under the  $X_1$  attribute of a new DAG first), producing state (18), which is then added to stateset 1.

$$\left\langle 1, \left[ \begin{array}{l} x0 \left[ \begin{array}{l} \text{maj} \quad \boxed{3} \text{ v} \\ \left[ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \end{array} \right] \\ \text{syn} \\ \left[ \begin{array}{l} \text{agr} \quad \boxed{4} \\ \text{spr} \quad \text{nil} \end{array} \right] \\ \text{surf} \quad \text{append}(\boxed{5}, \boxed{6}) \end{array} \right] \\ x1 \quad \boxed{1} \left[ \begin{array}{l} \text{maj} \quad \text{det} \\ \left[ \begin{array}{l} \text{spr} \quad \text{nil} \\ \text{comps} \quad \text{nil} \end{array} \right] \\ \text{syn} \\ \left[ \begin{array}{l} \text{agr} \quad \boxed{4} \\ \text{spr} \quad \text{nil} \end{array} \right] \\ \text{surf} \quad \boxed{5} \text{ 'the' } \end{array} \right] \\ x2 \left[ \begin{array}{l} \text{maj} \quad \boxed{3} \\ \left[ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \end{array} \right] \\ \text{syn} \\ \left[ \begin{array}{l} \text{agr} \quad \boxed{4} \\ \text{spr} \quad \boxed{1} \end{array} \right] \\ \text{surf} \quad \boxed{6} \end{array} \right] \end{array} \right], 2, 0, \left[ \begin{array}{l} \text{maj} \quad \text{v} \\ \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \end{array} \right] \\ \text{syn} \end{array} \right], \left[ \begin{array}{l} \text{maj} \quad \text{v} \\ \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \boxed{\quad} \end{array} \right] \\ \text{syn} \end{array} \right] \right] \right\rangle \quad (18)$$

State (18) therefore differs from (15) in the values for the first daughter’s attributes: the major category and surface form (`surf`) are now recognized.

When the parser constructs the forward pointer for state (16) (in order to begin prediction), it recognizes that it is subsumed by an RD that has previously been used to make predictions: the forward pointer in (14). At this point, the predictor simply halts; no further processing occurs. The scanner, however, is still applied to this state; it does not find lexical items that unify with the active feature structure.

In state (17), the forward pointer is also subsumed by an element on the predictor list (in this case, that of state (15)) and no further predictions are made. The scanner, however, successfully unifies the forward pointer with the RD of the lexical item *the*. It therefore adds state (19) to stateset 1.

$$\left\langle 1, \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \left[ \begin{array}{l} \begin{array}{l} \text{maj} \quad \boxed{3} \\ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \end{array} \\ \text{syn} \\ \begin{array}{l} \text{agr} \quad \boxed{4} \\ \text{spr} \quad \text{nil} \end{array} \\ \text{surf} \quad \text{append}(\boxed{5}, \boxed{6}) \end{array} \\ \begin{array}{l} \text{maj} \quad \text{det} \\ \begin{array}{l} \text{spr} \quad \text{nil} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \end{array} \\ \text{surf} \quad \boxed{5} \text{'the'} \end{array} \\ \begin{array}{l} \text{maj} \quad \boxed{3} \\ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \\ \text{spr} \quad \boxed{1} \end{array} \\ \text{surf} \quad \boxed{6} \end{array} \end{array} \right], 2, 0, \left[ \text{syn} \begin{array}{l} \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \end{array} \end{array} \right], \left[ \text{syn} \begin{array}{l} \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \boxed{\phantom{1}} \end{array} \end{array} \right] \right\rangle \quad (19)$$

State (19) differs from (17) in its values for the first daughter and from (18) in leaving the parent's major category unspecified.

In state (18), the dot is at  $X_2$ , reflecting the fact that the lexical item for  $X_1$  has already been scanned. The predictor will automatically apply here; as this is a new stateset, the predictor list is empty. (Both of these predictions lead to dead-ends, however, and are omitted here.) The scanner applies to this state, but fails to find any lexical items whose RDs unify with the forward pointer.

The predictor similarly creates two dead-end states from state (19). When the scanner is applied to (19), however, it finds the lexical item *swimmer*, whose RD unifies with this state's forward pointer, and adds state (20) to stateset 2.

$$\left\langle 2, \left[ \begin{array}{l} x0 \left[ \begin{array}{l} \text{maj} \quad \boxed{3} \text{ n} \\ \text{syn} \left[ \begin{array}{l} \text{form} \quad \boxed{2} \text{ base} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \left[ \begin{array}{l} \text{cls} \quad \text{tsg} \\ \text{num} \quad \text{sg} \end{array} \right] \\ \text{spr} \quad \text{nil} \end{array} \right] \\ \text{surf} \quad \text{append}(\boxed{5}, \boxed{6}) \end{array} \right] \\ \\ x1 \quad \boxed{1} \left[ \begin{array}{l} \text{maj} \quad \text{det} \\ \text{syn} \left[ \begin{array}{l} \text{spr} \quad \text{nil} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \end{array} \right] \\ \text{surf} \quad \boxed{5} \text{ 'the' } \end{array} \right] \\ \\ x2 \left[ \begin{array}{l} \text{maj} \quad \boxed{3} \\ \text{syn} \left[ \begin{array}{l} \text{form} \quad \boxed{2} \\ \text{comps} \quad \text{nil} \\ \text{agr} \quad \boxed{4} \\ \text{spr} \quad \boxed{1} \end{array} \right] \\ \text{surf} \quad \boxed{6} \text{ 'swimmer' } \end{array} \right] \end{array} \right] , 3, 0, \left[ \text{syn} \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \end{array} \right] \right], \left[ \text{syn} \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \boxed{\quad} \end{array} \right] \right] \right\rangle \quad (20)$$

Notice that in state (20), the unification of the lexical information for the second daughter has added new information to the description of the parent's attributes; in particular, values are now present for the `maj`, `form`, and `agr` attributes.

Since state (20) is a final state, the predictor and scanner do not apply. Instead, the completer recognizes that this state originated from stateset 0. It examines that stateset and finds three states whose forward pointers unify with (20)'s backpointer (i.e. three potential ancestors): (11), (15), and (17). For each of these, the parent node is extracted and unified with the active feature structure in the ancestor states. For (11), this unification fails. With (15) and (17), however, the unification succeeds. The completer constructs new states in the current stateset out of these states by leaving the origin and backpointer the same and advancing the dot by one position. The results are (21) and (22), respectively.

$$\left\langle 2, \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \left[ \begin{array}{l} \left[ \begin{array}{l} \text{maj } \boxed{3} \text{ v} \\ \left[ \begin{array}{l} \text{form } \boxed{2} \\ \text{comps nil} \\ \text{agr } \boxed{4} \left[ \begin{array}{l} \text{cls tsg} \\ \text{num sg} \end{array} \right] \\ \text{spr nil} \end{array} \right] \\ \text{surf append}(\boxed{5}, \boxed{6}) \end{array} \right] \\ \left[ \begin{array}{l} \text{maj n} \\ \left[ \begin{array}{l} \text{spr nil} \\ \text{comps nil} \\ \text{agr } \boxed{4} \\ \text{form base} \end{array} \right] \\ \text{surf } \boxed{5} \text{ 'the swimmer' } \end{array} \right] \\ \left[ \begin{array}{l} \text{maj } \boxed{3} \\ \left[ \begin{array}{l} \text{form } \boxed{2} \\ \text{comps nil} \\ \text{agr } \boxed{4} \\ \text{spr } \boxed{1} \end{array} \right] \\ \text{surf } \boxed{6} \end{array} \right] \end{array} \right] \right. \\ \left. , 2, 0, \left[ \begin{array}{l} \text{maj v} \\ \left[ \begin{array}{l} \text{comps nil} \\ \text{spr nil} \end{array} \right] \end{array} \right], \left[ \begin{array}{l} \text{maj v} \\ \left[ \begin{array}{l} \text{comps nil} \\ \text{spr } \square \end{array} \right] \end{array} \right] \right\rangle \quad (21)$$

$$\left\langle 2, \left[ \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \right], 0, \left[ \begin{array}{l} \text{syn} \\ \text{syn} \\ \text{syn} \end{array} \right], \left[ \begin{array}{l} \left[ \begin{array}{l} \text{comps} \text{ nil} \\ \text{spr} \text{ nil} \end{array} \right] \\ \left[ \begin{array}{l} \text{comps} \text{ nil} \\ \text{spr} \text{ nil} \end{array} \right] \\ \left[ \begin{array}{l} \text{comps} \text{ nil} \\ \text{spr} \text{ nil} \end{array} \right] \end{array} \right] \right\rangle \quad (22)$$

Detailed description of the parse tree structure in equation (22):

- The root node is a list:  $\langle 2, [x0, x1, x2], 0, [\text{syn}, \text{syn}, \text{syn}], [\dots] \rangle$ .
- Node  $x0$  is a list:  $[ \text{maj } \boxed{3}, \text{syn } [ \text{form } \boxed{2}, \text{comps } \text{nil}, \text{agr } \boxed{4} [ \text{cls } \text{tsg}, \text{num } \text{sg} ], \text{spr } \text{nil} ], \text{surf } \text{append}(\boxed{5}, \boxed{6}) ]$ .
- Node  $x1$  is a list:  $[ \boxed{1}, \text{syn } [ \text{maj } \text{n}, \text{spr } \text{nil}, \text{comps } \text{nil}, \text{agr } \boxed{4}, \text{form } \text{base} ], \text{surf } \boxed{5} \text{'the swimmer'} ]$ .
- Node  $x2$  is a list:  $[ \text{maj } \boxed{3}, \text{syn } [ \text{form } \boxed{2}, \text{comps } \text{nil}, \text{agr } \boxed{4}, \text{spr } \boxed{1} ], \text{surf } \boxed{6} ]$ .
- The list of lists  $[\dots]$  contains three elements:  $[ \text{comps } \text{nil}, \text{spr } \text{nil} ], [ \text{comps } \text{nil}, \text{spr } \text{nil} ], [ \text{comps } \text{nil}, \text{spr } \text{nil} ]$ .

The rest of the parse proceeds similarly. As before, once the completer successfully applies to the initial state, the parse succeeds. If at any point there are no states left to process in any future stateset, the parse fails.

#### 4. Lexical Rule Implementation

In a feature structure grammar, a lexical rule must completely describe the correspondences between the two forms related by the rule. For instance, a grammar might posit (23) as the lexical rule relating passive verbs to their base lexical entries.

$$\left[ \begin{array}{l}
x0 \\
x1
\end{array} \right] \left[ \begin{array}{l}
\left[ \begin{array}{l}
\text{maj} \quad \boxed{1} \text{ v} \\
\text{syn} \quad \left[ \begin{array}{l}
\text{form} \quad \text{ppar} \\
\text{comps} \quad \boxed{3} \\
\text{spr} \quad \boxed{2}
\end{array} \right] \\
\text{surf} \quad \text{ppar}(\boxed{4})
\end{array} \right] \\
\left[ \begin{array}{l}
\text{maj} \quad \boxed{1} \\
\text{syn} \quad \left[ \begin{array}{l}
\text{form} \quad \text{base} \\
\text{comps} \quad \langle \boxed{2}, \boxed{3} \rangle
\end{array} \right] \\
\text{surf} \quad \boxed{4}
\end{array} \right]
\end{array} \right] \quad (23)$$

Here, the  $X_0$  attribute describes the passive form, while the  $X_1$  attribute describes the base form (i.e. the form given in the lexicon). The rule states that the surface form (the value of the `surf` attribute) of the passive is related to the form of the base verb through the `ppar` function (as described in Section 5). Similarly, the rule specifies the relationship between the `spr` and `comps` attributes of each form: the specifier of the passive form is the first complement of the base form, while the remaining complements of the base form become the complements of the passive form. The two forms share the same major category (`maj`), and each receives an appropriate `form` value: `ppar` for the passive form and `base` for the base form.

As described so far, UNICORN cannot accurately parse with a grammar containing lexical rules: it expects to be able to directly match lexical entries to input tokens. As some input tokens might only be produced as the output of a lexical rule, they will never be found in the lexicon, and they will be impossible to scan.

#### 4.1 Context-free Lexical Rules

To see how this capability can be added to UNICORN, consider the context-free equivalent to a lexical rule: a rule that has terminal symbols on both sides of the rule, as in the set of rules in (24).

$$\begin{array}{l}
0 : S' \rightarrow S \\
1 : S \rightarrow A A \\
2 : A \rightarrow a \\
3 : A \rightarrow b \\
4 : b \rightarrow c
\end{array} \quad (24)$$

The only change we need make to Earley's algorithm is to allow the predictor to apply to states whose active symbols are terminals. A trace of the parse for the string  $bc$  is given in (25).

	$I_o$	state	created by
Stateset 0 — Input: b			
0	0	$S' \rightarrow . S$	initial
1	0	$S \rightarrow . A A$	predict(0,1)
2	0	$A \rightarrow . a$	predict(1,2)
3	0	$A \rightarrow . b$	predict(1,3)
4	0	$b \rightarrow . c$	predict(3,4)
Stateset 1 — Input: c			
5	0	$A \rightarrow b .$	scan(3)
6	0	$S \rightarrow A . A$	complete(5,1)
7	1	$A \rightarrow . a$	predict(6,2)
8	1	$A \rightarrow . b$	predict(6,3)
9	1	$b \rightarrow . c$	predict(8,4)
Stateset 2 — Input: \$			
10	1	$b \rightarrow c .$	scan(9)
11	1	$A \rightarrow b .$	complete(10,8)
12	0	$S \rightarrow A A .$	complete(11,6)
13	0	$S' \rightarrow S .$	complete(12,1)

(25)

Line 4 in this trace represents a lexical rule prediction. When applying the scanner to line 3, the parser recognizes that the symbol  $b$  is a terminal. Under the original Earley’s algorithm, the predictor would not apply to this state. Here, the predictor recognizes that a lexical rule exists whose left-hand side matches the active symbol, and therefore adds line 4 to stateset 0. This also occurs when the parser processes line 8, predicting line 9. The rest of the parse proceeds exactly as in section 2.

#### 4.2 Feature Structure Lexical Rules

As mentioned before, UNICORN expects to directly match input symbols against the surface forms of the lexical entries, before that lexical entry is unified with the active symbol. To accurately parse with lexical rules, however, we must defer this input scanning until after the unifications, when the surface form can be accurately matched to the output of the morphological functions.

To accomplish this, each of the three state-adding components of the parser is modified. The modifications to the predictor are minor: any state predicted from a lexical rule is now tagged as a lexical state; any state predicted from a non-lexical rule is now tagged as a non-lexical state.

The scanner is unchanged with respect to non-lexical states (scanning preserves the lexical/non-lexical status of a state). With lexical states, however, the input scanning is omitted. Instead, when the scanner applies to a lexical state, we produce a new state from every lexical item that successfully unifies with the active feature structure.

Non-lexical states are also completed the same way as before, as are lexical states with lexical ancestors. When the completer encounters a lexical state with a non-lexical ancestor, the surface form of the parent

node of the current state must match the current input token for completion to succeed. This has the effect of deferring input matching until all lexical rule application has completed

To illustrate these changes, consider the parse presented in section 3. When the new predictor applies to state (22) (repeated here as (26)), it will produce state (27) (among others) from rule (28).

$$\left\langle 2, \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \left[ \begin{array}{l} \begin{array}{l} \text{maj } \boxed{3} \\ \begin{array}{l} \text{form } \boxed{2} \\ \text{comps } \text{nil} \\ \text{agr } \boxed{4} \begin{array}{l} \text{cls } \text{tsg} \\ \text{num } \text{sg} \end{array} \\ \text{spr } \text{nil} \end{array} \\ \text{surf } \text{append}(\boxed{5}, \boxed{6}) \end{array} \\ \begin{array}{l} \text{maj } \text{n} \\ \begin{array}{l} \text{spr } \text{nil} \\ \text{comps } \text{nil} \\ \text{agr } \boxed{4} \\ \text{form } \text{base} \end{array} \\ \text{surf } \boxed{5} \text{'the swimmer'} \end{array} \\ \begin{array}{l} \text{maj } \boxed{3} \\ \begin{array}{l} \text{form } \boxed{2} \\ \text{comps } \text{nil} \\ \text{agr } \boxed{4} \\ \text{spr } \boxed{1} \end{array} \\ \text{surf } \boxed{6} \end{array} \end{array} \right] \right. \\ \left. , 2, 0, \text{nl}, \left[ \text{syn } \begin{array}{l} \text{comps } \text{nil} \\ \text{spr } \text{nil} \end{array} \right], \left[ \text{syn } \begin{array}{l} \text{comps } \text{nil} \\ \text{spr } \boxed{\phantom{1}} \end{array} \right] \right\rangle \quad (26)$$

$$\left\langle 2, \left[ \begin{array}{l} x0 \left[ \begin{array}{l} \text{maj } \boxed{3} \text{ v} \\ \text{form } \boxed{2} \\ \text{agr } \boxed{4} \left[ \begin{array}{l} \text{cls tsg} \\ \text{num sg} \end{array} \right] \\ \text{comps nil} \\ \text{syn} \left[ \begin{array}{l} \text{maj n} \\ \text{spr nil} \\ \text{comps nil} \\ \text{agr } \boxed{4} \\ \text{form base} \\ \text{surf 'the swimmer'} \end{array} \right] \\ \text{surf append}(\boxed{6}, \boxed{7}) \end{array} \right] \\ x1 \left[ \begin{array}{l} \text{maj } \boxed{3} \\ \text{form } \boxed{2} \\ \text{comps } \langle \boxed{1} \rangle \\ \text{spr } \boxed{5} \\ \text{agr } \boxed{4} \\ \text{surf } \boxed{6} \end{array} \right] \\ x2 \boxed{1} \left[ \text{surf } \boxed{7} \right] \end{array} \right] , 1, 2, nl, \left[ \text{syn } \left[ \begin{array}{l} \text{comps nil} \\ \text{spr } \boxed{\phantom{1}} \end{array} \right] \right], \left[ \text{syn } \left[ \begin{array}{l} \text{comps } \boxed{\phantom{1}} \\ \text{spr } \boxed{\phantom{1}} \end{array} \right] \right] \rangle \quad (27)$$

$$\left[ \begin{array}{l} x0 \left[ \begin{array}{l} \text{maj } \boxed{4} \\ \text{form } \boxed{2} \\ \text{comps } \boxed{3} \\ \text{agr } \boxed{5} \\ \text{spr } \boxed{6} \\ \text{surf append}(\boxed{7}, \boxed{8}) \end{array} \right] \\ x1 \left[ \begin{array}{l} \text{maj } \boxed{4} \\ \text{form } \boxed{2} \\ \text{comps } \langle \boxed{1}, \boxed{3} \rangle \\ \text{agr } \boxed{5} \\ \text{spr } \boxed{6} \\ \text{surf } \boxed{7} \end{array} \right] \\ x2 \boxed{1} \left[ \text{surf } \boxed{8} \right] \end{array} \right] \quad (28)$$

State (27) therefore represents an instantiation of (28) in which the parent's attributes are more fully specified;

values are now present for the `maj`, `agr`, `comps`, and `spr` attributes. When the predictor is applied to (27), it uses rule (29) to produce state (30).

$$\left[ \begin{array}{l} x0 \\ x1 \end{array} \right] \left[ \begin{array}{l} \text{maj } \boxed{3} v \\ \text{syn } \left[ \begin{array}{l} \text{form } \text{finite} \\ \text{agr } \left[ \begin{array}{l} \text{cls } \text{tsg} \end{array} \right] \\ \text{comps } \boxed{1} \\ \text{spr } \boxed{2} \end{array} \right] \\ \text{surf } \text{sing}(\boxed{4}) \end{array} \right] \right] \quad (29)$$

$$\left\langle 2, \left[ \begin{array}{l} x0 \\ x1 \end{array} \right] \left[ \begin{array}{l} \text{maj } \boxed{3} v \\ \text{syn } \left[ \begin{array}{l} \text{form } \text{finite} \\ \text{agr } \boxed{5} \left[ \begin{array}{l} \text{cls } \text{tsg} \\ \text{num } \text{sg} \end{array} \right] \\ \text{comps } \langle \boxed{1} \rangle \\ \text{spr } \boxed{2} \left[ \begin{array}{l} \text{maj } n \\ \text{syn } \left[ \begin{array}{l} \text{spr } \text{nil} \\ \text{comps } \text{nil} \\ \text{agr } \boxed{5} \\ \text{form } \text{base} \end{array} \right] \\ \text{surf } \text{'the swimmer'} \end{array} \right] \end{array} \right] \\ \text{surf } \text{sing}(\boxed{4}) \end{array} \right] \right] \left[ \begin{array}{l} \text{maj } \boxed{3} \\ \text{syn } \left[ \begin{array}{l} \text{form } \text{base} \\ \text{comps } \langle \boxed{1} \rangle \\ \text{spr } \boxed{2} \end{array} \right] \\ \text{surf } \boxed{4} \end{array} \right] \right] \left[ \begin{array}{l} \text{maj } v \\ \text{syn } \left[ \begin{array}{l} \text{form } \text{base} \\ \text{comps } \square \\ \text{spr } \square \end{array} \right] \end{array} \right] \left[ \begin{array}{l} \text{comps } \square \\ \text{spr } \square \end{array} \right] \right] \left[ \begin{array}{l} \text{maj } v \\ \text{syn } \left[ \begin{array}{l} \text{form } \text{base} \\ \text{comps } \square \\ \text{spr } \square \end{array} \right] \end{array} \right] \right\} \quad (30)$$

State (30) similarly represents a fuller instantiation of (29) in which values for `agr | num`, `comps`, and `spr` have been unified in. From (30), the scanner unifies the  $X_1$  node with the lexical entry for *splash*, producing state (31).



$$\left\langle 2, \left[ \begin{array}{l} x0 \\ x1 \\ x2 \end{array} \right] \right\rangle, 2, 2, nl, \left[ \begin{array}{l} \text{syn} \\ \text{surf} \end{array} \right] \left[ \begin{array}{l} \text{comps} \\ \text{spr} \end{array} \right] \left[ \begin{array}{l} \text{nil} \\ \square \end{array} \right] \left[ \begin{array}{l} \text{maj} \\ \text{syn} \end{array} \right] \left[ \begin{array}{l} n \\ \text{comps} \\ \text{spr} \end{array} \right] \left[ \begin{array}{l} \text{nil} \\ \text{nil} \\ \text{nil} \end{array} \right] \right\rangle \quad (32)$$

Detailed description of the diagram: The diagram shows a hierarchical feature structure. On the left, a large vertical bracket groups three nodes: x0, x1, and x2. Node x0 is a list containing:

- maj: 3 v
- form: 2 finite
- agr: 4 [cls tsg, num sg]
- comps: nil
- syn: [maj n, spr nil, comps nil, agr 4, form base, cse nom]
- spr: 5 syn
- surf: 'the swimmer'

Node x1 is a list containing:

- maj: 3
- form: 2
- comps: <1>
- spr: 5
- agr: 4
- surf: 6 'splashes'

Node x2 is a list containing:

- 1
- syn: [maj n, comps nil, spr nil, cse acc]
- surf: 7

To the right of this structure is the expression ', 2, 2, nl, [syn [comps nil], [spr □], [maj n, syn [comps nil, spr nil]]]'. This is followed by a large right-pointing arrow and the label '(32)'.

Here, the parent has received values for the `form` and `syn|spr|syn|cse` attributes, the first daughter has its `surf` attribute specified, and the second daughter now has values for `maj`, and `syn`. The rest of the parse is constructed similarly.

### 4.3 A Potential Problem

The deferral of input checking when the scanner is applied to a lexical state can be a potential problem. As presented here, the scanner must, in effect, attempt to unify the active feature structure with every compatible lexical item. In a lexicon with a thousand transitive verbs, then, the scanner would have to create a thousand states every time it prepared to scan a transitive verb, rather than just those whose surface forms match the input (and would be retrievable through a simple hash table query).

In some sense, this is simply an unavoidable consequence of the top-down nature of Earley’s algorithm. The simplest top-down parser for a grammar with  $n$  words and an input of  $k$  tokens simply enumerates the  $k^n$  possible sentences and checks to see which of these matches the input, outputting the respective parse tree. All top-down parsing algorithms are refinements of this strategy in which the possibilities are kept as few as possible, usually by collapsing as many possibilities as possible into one prediction. It is therefore natural to expect this explosion of states.

On the other hand, it is similarly natural to expect that we can sufficiently collapse these states into a small number of predictions. This can be done in two different (but not mutually exclusive) ways. One method would be to design the grammar in such a way that the chosen restrictor was more effective in selecting a subrange of lexical items. In some cases, simply adding more paths to the restrictor would have this effect.

The other method would involve changing the way words are stored in the lexicon. Instead of a list of words with corresponding feature structures, the lexicon would consist of several feature structures for word classes and a list of lexical items belonging to each class. For instance, “singular noun” and “intransitive verb” might be two word classes. Under this system, the scanner would simply find an appropriate word class feature structure to unify with the active symbol; the word lists would not be accessed until completion. As many linguistic theories posit some sort of structured lexicon, this is a natural modification to make.

## 5. Finite-State Morphology

The modifications described in section 4 successfully allow the relationship between base and derived-form feature structures to be represented in the grammar. This section describes how the parser interprets a value like `ppar(splash)`.

Morphological relationships are described in terms of morphological correspondence functions (i.e. `sing` in (31)). These functions have two components. The first is an exception list. This handles irregular forms (for instance, *be* → *is*). The second part specifies an input to the finite-state transducer (for instance, `__+s`) for regular cases. Thus the input to the function is first matched against the items on the exception list. If it matches any of them, the function simply returns the corresponding output form. Otherwise, the input to the transducer is formed by replacing the `_` in the output pattern with the input.

The transducer then processes this input according to a user-provided set of morphological rules. These rules are represented in the form of finite state tables. From these tables, a list of *feasible pairs* (possible correspondences between surface and underlying forms) is formed. Upon receiving an input string, the transducer runs all possible surface forms for the given underlying form through each rule. Only those forms which every finite state machine accepts are legitimate outputs.

## 6. Conclusion

The ability to parse with lexical rules is crucial to a proper implementation of many modern linguistic theories. As feature-structure grammars become increasingly popular for linguistic modelling, it is important that they also gain this ability. Once a mechanism for handling surface form correspondence has been incorporated into the parser, only a simple set of changes to the parser is required to accomplish this.

## A. Sample Grammar

### A.1 Lexicon

(Here, (33) is the version used in the Section 3 (non-lexical rule) grammar; (34) is the version used in the Section 4 (lexical rule) grammar.)

$$\left[ \begin{array}{l} \text{maj} \quad v \\ \text{surf} \quad \text{splashes} \\ \left[ \begin{array}{l} \text{form} \quad \text{finite} \\ \left[ \begin{array}{l} \text{maj} \quad n \\ \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \\ \text{syn} \quad \left[ \begin{array}{l} \text{cse} \quad \text{nom} \\ \text{agr} \quad \left[ \text{cls} \quad \text{tsg} \right] \end{array} \right] \end{array} \right] \end{array} \right] \\ \text{comps} \quad \left\langle \begin{array}{l} \left[ \begin{array}{l} \text{maj} \quad n \\ \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \\ \text{cse} \quad \text{acc} \end{array} \right] \end{array} \right] \\ \left[ \begin{array}{l} \text{syn} \quad \left[ \begin{array}{l} \text{cse} \quad \text{acc} \end{array} \right] \end{array} \right] \end{array} \right\rangle, \text{nil} \end{array} \right] \end{array} \right] \end{array} \right] \quad (33)$$

$$\left[ \begin{array}{l} \text{maj} \quad v \\ \text{surf} \quad \text{splash} \\ \left[ \begin{array}{l} \text{form} \quad \text{base} \\ \left[ \begin{array}{l} \text{maj} \quad n \\ \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \\ \text{syn} \quad \left[ \begin{array}{l} \text{cse} \quad \text{nom} \end{array} \right] \end{array} \right] \end{array} \right] \\ \text{comps} \quad \left\langle \begin{array}{l} \left[ \begin{array}{l} \text{maj} \quad n \\ \left[ \begin{array}{l} \text{comps} \quad \text{nil} \\ \text{spr} \quad \text{nil} \\ \text{cse} \quad \text{acc} \end{array} \right] \end{array} \right] \\ \left[ \begin{array}{l} \text{syn} \quad \left[ \begin{array}{l} \text{cse} \quad \text{acc} \end{array} \right] \end{array} \right] \end{array} \right\rangle, \text{nil} \end{array} \right] \end{array} \right] \end{array} \right] \quad (34)$$

$$\begin{bmatrix} \text{maj} & \text{det} \\ \text{surf} & \text{the} \\ \text{syn} & \begin{bmatrix} \text{spr} & \text{nil} \\ \text{comps} & \text{nil} \end{bmatrix} \end{bmatrix} \quad (35)$$

$$\begin{bmatrix} \text{maj} & \text{n} \\ \text{surf} & \text{swimmer} \\ \text{syn} & \begin{bmatrix} \text{spr} & \begin{bmatrix} \text{maj} & \text{det} \\ \text{cls} & \text{tsg} \\ \text{num} & \text{sg} \end{bmatrix} \\ \text{agr} & \\ \text{comps} & \text{nil} \\ \text{form} & \text{base} \end{bmatrix} \end{bmatrix} \quad (36)$$

## A.2 Non-Lexical Rules

$$\begin{bmatrix} \text{x0} & \begin{bmatrix} \text{maj} & \boxed{4} \\ \text{syn} & \begin{bmatrix} \text{form} & \boxed{2} \\ \text{comps} & \boxed{3} \\ \text{agr} & \boxed{5} \\ \text{spr} & \boxed{6} \end{bmatrix} \\ \text{surf} & \text{append}(\boxed{7}, \boxed{8}) \end{bmatrix} \\ \text{x1} & \begin{bmatrix} \text{maj} & \boxed{4} \\ \text{syn} & \begin{bmatrix} \text{form} & \boxed{2} \\ \text{comps} & \langle \boxed{1}, \boxed{3} \rangle \\ \text{agr} & \boxed{5} \\ \text{spr} & \boxed{6} \end{bmatrix} \\ \text{surf} & \boxed{7} \end{bmatrix} \\ \text{x2} & \boxed{1} \begin{bmatrix} \text{surf} & \boxed{8} \end{bmatrix} \end{bmatrix} \quad (37)$$

$$\left[ \begin{array}{l}
 x0 \left[ \begin{array}{l}
 \text{maj } \boxed{3} \\
 \text{syn } \left[ \begin{array}{l}
 \text{form } \boxed{2} \\
 \text{comps nil} \\
 \text{agr } \boxed{4} \\
 \text{spr nil}
 \end{array} \right] \\
 \text{surf } \text{append}(\boxed{5}, \boxed{6})
 \end{array} \right] \\
 x1 \boxed{1} \left[ \begin{array}{l}
 \text{syn } \left[ \begin{array}{l}
 \text{agr } \boxed{4} \\
 \text{spr nil}
 \end{array} \right] \\
 \text{surf } \boxed{5}
 \end{array} \right] \\
 x2 \left[ \begin{array}{l}
 \text{maj } \boxed{3} \\
 \text{syn } \left[ \begin{array}{l}
 \text{form } \boxed{2} \\
 \text{comps nil} \\
 \text{agr } \boxed{4} \\
 \text{spr } \boxed{1}
 \end{array} \right] \\
 \text{surf } \boxed{6}
 \end{array} \right]
 \end{array} \right] \tag{38}$$

### A.3 Lexical Rules

$$\left[ \begin{array}{l}
 x0 \left[ \begin{array}{l}
 \text{maj } \boxed{3} v \\
 \text{syn } \left[ \begin{array}{l}
 \text{form } \text{finite} \\
 \text{agr } \left[ \text{cls } \text{non} \right] \\
 \text{comps } \boxed{1} \\
 \text{spr } \boxed{2}
 \end{array} \right] \\
 \text{surf } \text{plurv}(\boxed{4})
 \end{array} \right] \\
 x1 \left[ \begin{array}{l}
 \text{maj } \boxed{3} \\
 \text{syn } \left[ \begin{array}{l}
 \text{form } \text{base} \\
 \text{comps } \boxed{1} \\
 \text{spr } \boxed{2}
 \end{array} \right] \\
 \text{surf } \boxed{4}
 \end{array} \right]
 \end{array} \right] \tag{39}$$

$$\begin{array}{l}
 \left[ \begin{array}{l}
 \text{x0} \\
 \text{x1}
 \end{array} \right]
 \begin{array}{l}
 \left[ \begin{array}{l}
 \text{maj} \quad \boxed{3} \text{ v} \\
 \text{syn} \quad \left[ \begin{array}{l}
 \text{form} \quad \text{finite} \\
 \text{agr} \quad \left[ \text{cls} \quad \text{tsg} \right] \\
 \text{comps} \quad \boxed{1} \\
 \text{spr} \quad \boxed{2}
 \end{array} \right] \\
 \text{surf} \quad \text{sing}(\boxed{4})
 \end{array} \right] \\
 \left[ \begin{array}{l}
 \text{maj} \quad \boxed{3} \\
 \text{syn} \quad \left[ \begin{array}{l}
 \text{form} \quad \text{base} \\
 \text{comps} \quad \boxed{1} \\
 \text{spr} \quad \boxed{2}
 \end{array} \right] \\
 \text{surf} \quad \boxed{4}
 \end{array} \right]
 \end{array}
 \end{array}
 \quad (40)$$

$$\begin{array}{l}
 \left[ \begin{array}{l}
 \text{x0} \\
 \text{x1}
 \end{array} \right]
 \begin{array}{l}
 \left[ \begin{array}{l}
 \text{maj} \quad \boxed{1} \text{ n} \\
 \text{syn} \quad \left[ \begin{array}{l}
 \text{form} \quad \boxed{3} \\
 \text{agr} \quad \left[ \begin{array}{l}
 \text{cls} \quad \text{non} \\
 \text{num} \quad \text{pl}
 \end{array} \right] \\
 \text{comps} \quad \boxed{4} \\
 \text{spr} \quad \boxed{5}
 \end{array} \right] \\
 \text{surf} \quad \text{plurn}(\boxed{2})
 \end{array} \right] \\
 \left[ \begin{array}{l}
 \text{maj} \quad \boxed{1} \\
 \text{syn} \quad \left[ \begin{array}{l}
 \text{form} \quad \boxed{3} \\
 \text{agr} \quad \left[ \begin{array}{l}
 \text{num} \quad \text{sg} \\
 \text{cls} \quad \text{tsg}
 \end{array} \right] \\
 \text{comps} \quad \boxed{4} \\
 \text{spr} \quad \boxed{5}
 \end{array} \right] \\
 \text{surf} \quad \boxed{2}
 \end{array} \right]
 \end{array}
 \end{array}
 \quad (41)$$

## Works Cited

- [1] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102, 1970.
- [2] Dale D. Gerdemann. Parsing and generation of unification grammars. Technical Report Cognitive Science CS-91-06, Beckman Institute, University of Illinois, 1991.
- [3] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.