The Formalism and Implementation of PATR-II

RESEARCH ON INTERACTIVE ACQUISITION AND USE OF KNOWLEDGE

Final Report
Covering the Period July 3, 1980 to Nov. 30, 1983

November 1983

Principal Investigators:
Barbara J. Grosz, Program Director, Natural Language
Mark E. Stickel, Senior Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

Attention: Commander Ronald B. Ohlander

SRI Project 1694

Preparation of this paper was supported by the Defense Advanced Research Pr
jects Agency under Contract N00039-80-C-0575 with the Naval Electronic Sy
tems Command.

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025-3493
Telephone: (415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-2046
Telex: 334 486

4. The Formalism and Implementation of PATR-II

This section was written by Stuart Shieber, Hans Uszkoreit, Fernando Pereira, Jane Robinson, and Mabry Tyson.

4.1. The PATR-II Formalism

4.1.1 Motivation for the Formalism

The goal of natural-language processing is simple: to enable computers to participate in dialogues with humans in their language in order to make the computers more useful to their creators. The pursuit of this objective, however, has been a difficult task for at least two reasons: first, the phenomenon of human language is not as well understood as is popularly supposed; second, the tools for teaching computers what we do know about human language are still quite primitive. The solution of these problems falls into the research domains of linguistics and computer science, respectively.

Similar problems have previously arisen in the field of computer science. With the advent of digital computers, the need for effective ways of communicating with computers, other than by means of patch panels, became quickly evident. The "black art" of programming-language design has improved greatly over the years and much is now known about effective communication with computers. In particular, the criteria for good programming languages are their *power*, *utility*, and, in the case of research languages, *simplicity and mathematical well-foundedness*. Note that only the first of these can be measured objectively; in fact, the power of most current programming languages is equivalent to that of a Turing machine. However, the basic fact is that more power is considered better. On the other hand, the other two criteria are inherently subjective, which is why programming language design is an art rather than a science. Utility, in fact, is usually a relative measure, relative to the purposes the language is

designed for. SNOBOL is a useful language for string manipulation, but awkward at best for, say, matrix manipulation. This is because the primitives supplied by SNOBOL do not match the common underlying operations of matrix handling.

Among the evaluation criteria for programming languages that have been assiduously promoted in the recent past is the aforementioned criterion of simplicity. Other trends have been in the direction of declarative languages, languages emphasizing structured programming and modularity, and the like. The design of a grammar formalism embodies the same problems as the design of a programming language simply because it aspires to the same goal, i.e., effective communication of information to a computer. Thus, the same criteria can be applied: the formalism should be as powerful as possible, should incorporate the types of primitives that natural-language grammar writers find they need, should be simple and mathematically well-founded. Trends from programming-language design, such as declarativeness and modularity, can also be applied to the problem of designing grammar formalisms for computers.

Theoretical linguists have been concerned with designing grammatical formalisms that provide the tools for expressing universal and language-specific generalizations in a concise and transparent fashion. One of their main objectives in this task is to constrain the power of their formalisms in concurrence with the cross-linguistic set of constraints upon syntactic and semantic phenomena that are found in natural language.

A radical but widespread opinion regarding the choice of an appropriate formalism is that it should embody all nonaccidental regularities that are observed in all languages, i.e., those that belong to universal grammar. For instance, if all languages are thought to possess coordination, this fact should derive from the formalism. If, on the other hand, no language in the world has the word "famakupa," which would be phonologically well-formed in many languages, we can then regard this observation as an accidental fact that will be represented in the set of particular grammars.

The PATR-II formalism as a tool for grammar writing does not attempt to encode most of the statements of universal grammar. It is based on the generally accepted view that sentences have structure, and it provides for structures that are more complex than phrase structure

trees. Not only do the regularities of specific languages have to be encoded by the user of the formalism—either in the proposed rules or in stipulations with regard to usage constraints—but so do most cross-linguistic generalizations, including constraints on generative power. The cross-linguistic generalization and constraints can be reflected in a selected implementation or usage notation. We shall discuss an example of such a notation later.

4.1.2 Design of the Formalism

We now describe the formalism that underlies the implementations of PATR-II. In some sense, this is the "operational semantics" of a PATR-II grammar. Certain implementations may make use of certain abbreviations or conventions, but the operation of such implementations is defined in terms of this simple underlying formalism. Thus, the formalism bears the same relation to PATR-II implementations as, say, pure LISP does to MACLISP.

The basic operation in PATR-II is ***unification***, an extended pattern-matching technique that was first used in logic and theorem-proving research and has been arousing considerable interest of late in the linguistics community. Rather than unifying logic terms, however, PATR unification operates on directed acyclic graphs (DAG).[1] DAGs can be atomic symbols or sets of label/value pairs whose labels (also called attributes or features) are atomic symbols or other DAGs (i.e., subDAGs). Since two labels can point to the same DAG, the term graph is used rather than tree. DAGs are notated either by drawing the labeled graph structure itself or, as in this paper, notating the sets of label/value pairs in square brackets ([ ]), with the labels separated from their values by a colon (:), e.g.,

```
[cat: v
 head: [aux: false
        form: nonfinite
        voice: active
        trans: [pred: eat
                arg1: <f1134>
                      []
                arg2: <f1138>
```

[1] Technically, these are rooted, unordered, directed, acyclic graphs with labeled arcs. See Appendix A for a more formal definition of PATR-II grammars and accompanying notions.

```
                    []]]
syncat: [first: [cat: np
                 head: [trans: <f1134>]]
         rest: [first: [cat: np
                        head: [trans: <f1138>]]
                rest: <f1140>
                      lambda]
         tail: <f1140>]]
```

Note that the re-entrant structure, where two arcs point to the same node, is notated by labeling the DAG with an arbitrary label (in angle brackets (<>)) and then using that label for future references to the DAG.

Associated with each lexical entry in the lexicon is a set of DAGs.[2] The root of each DAG will have an arc labeled *cat* whose value will be the category of the associated lexical entry. Other arcs may encode information about the syntactic features, translation, or syntactic subcategorization of the entry.

PATR-II grammars consist of rules with a context-free phrase structure portion and a series of unifications on the DAGs associated with the constituents taking part in the use of the rule. The grammar rules notate how constituents can be built up to form new constituents with associated DAGs. The right side of the rule lists the *cat* values of the DAGs associated with the child constituents; the left side, the *cat* of the parent. Other unifications specify equivalences that must exist among the various DAGs and subDAGs of the parent and children. Thus, the formalism uses only one representation (DAGs) for lexical, syntactic, and semantic information, and just one operation (unification) on this representation.

By way of example, we present a small grammar for a fragment of English, accompanied by a lexicon associating words with DAGs.

$S \rightarrow NP\ VP$

$$<VP\ agr> = <NP\ agr>$$

$VP \rightarrow V\ NP$

$$<VP\ agr> = <V\ agr>$$

[2]We shall postpone until later any discussion as to how this association is encoded or implemented.

Uther:

$$\begin{aligned} \langle cat \rangle &= np \\ \langle agr\ number \rangle &= singular \\ \langle agr\ person \rangle &= third \end{aligned}$$

Arthur:

$$\begin{aligned} \langle cat \rangle &= np \\ \langle agr\ number \rangle &= singular \\ \langle agr\ person \rangle &= third \end{aligned}$$

knights:

$$\begin{aligned} \langle cat \rangle &= v \\ \langle agr\ number \rangle &= singular \\ \langle agr\ person \rangle &= third \end{aligned}$$

This grammar (plus lexicon) admits the two sentences "Uther knights Arthur" and "Arthur knights Uther." The phrase structure associated with the first of these is

[S [NP Uther] [VP [V knights] [NP Arthur]]]

The VP rule requires that the *agr* feature of the DAG associated with the VP be the same as (unified with) the *agr* of the V. Thus the VP's *agr* feature will have as its value the *same* node as the V's *agr* and, hence, the same values for the *person* and *number* features. Similarly, by the unification associated with the S rule, the NP will have the same *agr* value as the VP and, consequently, the V. We have thus encoded a form of subject-verb agreement.

4.1.3 Power of the Formalism

PATR-II grammars, as just presented, are extremely powerful; in fact, they are equivalent to Turing machines. We therefore present a straightforward constraint upon their power that guarantees decidability, a constraint Pereira [98] calls *off-line parsability*. Off-line parsability requires that there be no nonproductive recursive chains of rules in the grammar, i.e., chains that can consume no input. Recursive chains of unary rules, or chains of rules in which all but one nonterminal in each rule can derive the empty string, are thus eliminated. In the case of context-free grammars, removing such rules does not change the power of the formalism.

PATR-II grammars, however, are restricted by this constraint—the specific effect of which is to render the parsing problem decidable.

Nonetheless, the power of PATR-II grammars remains great. Appendix B presents grammars for the non-context-free triple-counting language $a^n b^n c^n$ and the non-indexed language $a^{2^{2^n}}$. It remains an open question whether there are interesting further constraints on PATR-II and other unification-based formalisms that reduce the parsing problem significantly without unduly constricting generative capacity. We should keep in mind that evaluation of such constraints requires aesthetic judgments, not scientific ones.

4.1.4 Future Research to Improve the Formalism

The formalism is broad and powerful enough to handle most—indeed, probably all—phenomena in the syntax and semantics of natural language. It has also turned out to be well suited for the classes of phenomena considered so far. Most of the research will have to be done in the area of choosing appropriate strategies for application of the formalism. However, there is a class of phenomena that might justify some extension or modification of the formalism: the phenomena of free or variable word order.

Although the formalism is powerful enough to deal with word order variability, there is a strong feeling on our side that it should be possible to express free variation more directly. We plan to work out the necessary modifications in the near future and, to this end, we hope to be able to use results of a proposed parallel research project for studying such word order variation.

One direction in which the formalism might be extended to allow for word order variability is the relaxation of constraints on possible feature values. Let us assume that these values can be nonatomic, i.e., that they can be sets or sequences. Let us furthermore regard the permutations of verb complements as an example of order variation. By allowing structure in the feature system, we can encode much more information about possible VP structures. One example follows, but the possibilities are endless (literally so, since, by doing this, we move from the realm of context-free languages to the realm of Turing computability). Suppose the

range R of the Syncat [see Section 4.3.4.1.] feature included atomic symbols and all sets and sequences of elements of R. Also suppose that we define an operation $\ominus$ acting on compound elements of R such that

$$< \alpha, \beta_1, \ldots, \beta_n > \ominus \gamma = < \alpha \ominus \gamma, \beta_1, \ldots, \beta_n >$$
$$\{\alpha, \beta_1, \ldots, \beta_n\} \ominus \gamma = < \alpha \ominus \gamma, \{\beta_1, \ldots, \beta_n\} >$$

Now we can write a grammar as follows:

$$VP \rightarrow VP\ COMP$$
$$Syncat(VP_1) = Syncat(VP_2) \ominus Form(COMP)$$
$$VP \rightarrow V$$
$$Syncat(VP) = Syncat(V)$$

This structuring allows us a way of expressing free word order in the subcategorizations. Thus, if a verb subcategorizes for $\{< \alpha\beta > \gamma\}$, it allows argument structures of $\alpha\beta\gamma$ and $\gamma\alpha\beta$ but not $\beta\alpha\gamma$ or $\alpha\gamma\beta$. Using similar techniques, ID/LP could be encoded in a $\ominus$ operator working on complex structures. In fact, this is basically how the ID/LP direct parsing algorithm of Shieber [117] works.

4.2. Some Uses of the Formalism: The Current PATR-II Grammar Design

In this section we present some ideas concerning different uses of the formalism and describe our own current usage. Although most of the techniques presented here represent our current use of the basic formalism, they should not be identified with the formalism itself, which allows for quite different strategies of grammar writing.

It should be mentioned that many syntactic constructs not discussed in this introduction to the formalism are currently handled by existing grammars for our implemented system. Among these are $\overline{S}$ complements, active, passive, "there" insertion, extraposition, raising and equi constructions, etc. (See Appendix D for a more complete PATR-II grammar, Appendix E for a transcript of the parsing system using the grammar.)

Before we start explaining our use of the formalism let us emphasize once more the considerable freedom it allows for writing a grammar. The only label with any special significance

in the formalism is the arc label *cat*. This is a consequence of the decision to use traditional context-free phrase structure rules to create part of the syntactic and semantic structure. Everything else, including the appropriate category symbols, has to be designed by the grammar writer. Part of the process of writing a grammar, therefore, involves deciding on a set of arc labels (attributes, features) that are used to encode pertinent information about constituents.

4.2.1 Feature Percolation

Linguistic formalisms often provide a technique for percolating a large set of features from a given child to its parent, for instance, by means of the *head feature convention* in GPSG or the ↑=↓ equation in LFG. Grouping of features in this way can be accomplished in PATR-II by placing the features on a subDAG of the DAG of the child under a special attribute, say *head*, and then unifying the *head* attribute of parent and child with a unification of the form *<parent head> = <child head>*. Agreement features and case, are examples of features that could be percolated in this way. Thus, the previous sample grammar might be extended to allow *head* feature percolation as follows:

S → NP VP

<S head> = <VP head>
<NP head agr> = <S head agr>

VP → V NP

<VP head> = <V head>

Other:

<cat> = np
<head agr number> = singular
<head agr person> = third

Arthur:

<cat> = np
<head agr number> = singular
<head agr person> = third

knights:

<cat> = v

$$\begin{aligned} \langle \textit{head agr number} \rangle &= \textit{singular} \\ \langle \textit{head agr person} \rangle &= \textit{third} \end{aligned}$$

4.2.2 Semantics

The meaning of a constituent, a segment of logical form, needs to be recorded somewhere in the DAG associated with it. For reasons of modularity, we would like this encoding to be separable from the other portions of the DAG that encode syntactic information. To encode meanings with no extra apparatus, we shall use the following encoding of logical-form fragments. A predicate applied to several arguments, for instance—$f(a, b, c)$—will be encoded with the arcs *pred* and arg_i, respectively. A constant will be notated with the feature *ref*. Thus, the fragment above would be encoded as

```
[pred: f
 arg1: [ref: a]
 arg2: [ref: b]
 arg3: [ref: c]]
```

More evocative names for the argument positions could be used, e.g., *agent, patient, goal,* though we will not use them here.

Note that the translation of a parent constituent is often associated with the translation of a specific child constituent (with other child translations adding further information). For instance, the translation of a VP will be identical to that of the child V, with complements supplying translations assigned to the arguments. We can therefore make *trans* a *head* feature and allow the standard *head* feature mechanism to distribute it appropriately. Adding translations to our small grammar, we get:

$S \rightarrow NP\ VP$

$$\begin{aligned} \langle \textit{S head} \rangle &= \langle \textit{VP head} \rangle \\ \langle \textit{NP head agr} \rangle &= \langle \textit{S head agr} \rangle \\ \langle \textit{S head trans arg1} \rangle &= \langle \textit{NP head trans} \rangle \end{aligned}$$

$VP \rightarrow V\ NP$

$$\begin{aligned} \langle \textit{VP head} \rangle &= \langle \textit{V head} \rangle \\ \langle \textit{VP head trans arg2} \rangle &= \langle \textit{NP head trans} \rangle \end{aligned}$$

Uther:

<*cat*>	=	*np*
<*head agr number*>	=	*singular*
<*head agr person*>	=	*third*
<*head trans ref*>	=	*uther'*

Arthur:

<*cat*>	=	*np*
<*head agr number*>	=	*singular*
<*head agr person*>	=	*third*
<*head trans ref*>	=	*arthur'*

knights:

<*cat*>	=	*v*
<*head agr number*>	=	*singular*
<*head agr person*>	=	*third*
<*head trans pred*>	=	*knight'*

This grammar will admit the same sentences as previously, yielding the translations (in prefix notation) *knight'(uther', arthur')*, and *knight'(arthur', uther')* respectively.

4.2.3 Coordinating Syntax and Semantics

The previous grammar performs a *de facto* coordination of syntax and semantics by requiring that the (syntactically) preverbal NP play the (semantic) role of first argument, and that the postverbal complement play the role of second argument. Such a direct one-time mapping is difficult to maintain, and various theories have solved this problem in different ways. In general, the solution requires adding one more *degree of freedom* in the mapping. GPSG obtains this degree of freedom because intensional-logic operators are able to act as combinators, reordering arguments. These operators are introduced through metarules (though they could have been introduced by lexical rules). LFG uses an intermediate representation to provide the additional degree of freedom, mapping syntactic objects onto a set of arbitrary labels—*SUBJ, OBJ, OBJ2*, etc.—and then mapping these in turn to argument positions.

Either of these solutions could be modeled in PATR-II, though our actual technique (which will be presented after subcategorization is discussed) is slightly different from both.

We offer an example of the LFG style solution at this juncture. An LFG grammar unifies the preverbal and postverbal NPs as the values of *subject* and *object*, respectively. If one declares these to be *head* features, they will be unified with the *subject* and *object* features of the V itself. The lexical entry for the V will then perform the second half of the mapping, i.e., from grammatical function to argument position.

$S \rightarrow NP\ VP$

$$\begin{aligned} \langle S\ head \rangle &= \langle VP\ head \rangle \\ \langle NP\ head\ agr \rangle &= \langle S\ head\ agr \rangle \\ \langle S\ head\ subject \rangle &= \langle NP\ head \rangle \end{aligned}$$

$VP \rightarrow V\ NP$

$$\begin{aligned} \langle VP\ head \rangle &= \langle V\ head \rangle \\ \langle VP\ head\ object \rangle &= \langle NP\ head \rangle \end{aligned}$$

Uther:

$$\begin{aligned} \langle cat \rangle &= np \\ \langle head\ agr\ number \rangle &= singular \\ \langle head\ agr\ person \rangle &= third \\ \langle head\ trans\ ref \rangle &= uther' \end{aligned}$$

Arthur:

$$\begin{aligned} \langle cat \rangle &= np \\ \langle head\ agr\ number \rangle &= singular \\ \langle head\ agr\ person \rangle &= third \\ \langle head\ trans\ ref \rangle &= arthur' \end{aligned}$$

knights:

$$\begin{aligned} \langle cat \rangle &= v \\ \langle head\ agr\ number \rangle &= singular \\ \langle head\ agr\ person \rangle &= third \\ \langle head\ trans\ pred \rangle &= knight' \\ \langle head\ trans\ arg1 \rangle &= \langle subject\ trans \rangle \\ \langle head\ trans\ arg2 \rangle &= \langle object\ trans \rangle \end{aligned}$$

It is now clear how a lexical rule might be written for passivization: it merely changes the roles of *subject* and *object* in the lexical entry in the appropriate way. The ability to perform such redirection of grammatical and semantic functions provides the requisite extra degree of freedom. Before presenting an alternative solution to the degree-of-freedom problem, we must discuss the related problems of verb phrase structure and subcategorization.

4.2.4 Verb Phrase Structure and Subcategorization

4.2.4.1 Nested versus Flat Structure

Various alternatives have been suggested for handling verb phrase structures in a grammar for English. The proposed methods fall into two main categories: flat structure and nested structure. The flat structure is epitomized by the treatment in GPSG. We shall start with this technique.

Suppose we have a GPSG of the form

$$< 1, VP \rightarrow V\ \alpha_1 \cdots \alpha_m >$$
$$< 2, VP \rightarrow V\ \beta_1 \cdots \beta_n >$$
etc.

This grammar generates flat verb-phrase structures in which the verb and all of its complements are siblings. In GPSG we get appropriate subcategorization by associating with the rule some distinguishing feature (in the nontechnical sense) then associating that feature with any verbs that subcategorize for the rule. (This association acts like a virtual pointer between verbs and rules.) The feature in the case of GPSG is the rule number.

Two points deserve mention. First, the rule number technique in GPSG is *outside* the feature system.[3] But, since there is presumably only a finite number of verb phrase rules, there is no reason that the rule number could not have status as a normal feature (in the technical sense). A PATR-II grammar using this technique would look like this:

$VP \rightarrow V\ \alpha_1 \cdots \alpha_m$
$\quad <V\ syncat> = 1$

$VP \rightarrow V\ \beta_1 \cdots \beta_n$
$\quad <V\ syncat> = 2$

and so on.

---

[3]Actually, the most recent versions of GPSG have abandoned the distinction between rule numbers and features.

Second, rule numbers are only one way of distinguishing rules. Any other distinguishing feature of rules could be used. In particular, if no two rules share the same right-hand side, the right-hand sides could themselves be used as the subcategorization, as in the following grammar:

$$VP \rightarrow V\ \alpha_1 \cdots \alpha_m$$
$$<V\ syncat> = [\alpha_1, \ldots, \alpha_m]$$

$$VP \rightarrow V\ \beta_1 \cdots \beta_n$$
$$<V\ syncat> = [\beta_1, \ldots, \beta_n]$$

Of course, we have introduced notation here that is not found in the PATR-II formalism, namely, lists. Before explaining this, let us be even more free with notation. We could make the grammar still more concise by taking advantage of the fact that DAGs carry their category "on their sleeve," so to speak.

$$VP \rightarrow V\ \alpha_1 \cdots \alpha_m$$
$$<V\ syncat> = \bigoplus_{i=1}^{m} [< \alpha_i\ cat >]$$

$$VP \rightarrow V\ \beta_1 \cdots \beta_n$$
$$<V\ syncat> = \bigoplus_{i=1}^{n} [< \beta_i\ cat >]$$

where $\bigoplus$ denotes the repeated use of the list concatenation operator $\oplus$.

Note that all the unifications of the rules in this sample VP grammar are of exactly the same form. We can take advantage of that fact in a grammar in which there is only one VP rule by making use of a regular expression notation for the right-hand side of the rule.

$$VP \rightarrow V\ \{\alpha_1 \cup \alpha_2 \cup \cdots \cup \beta_m\}^*$$
$$<V\ syncat> = \bigoplus_{i=1}^{n} [< COMP_i\ cat >]$$

where n is the number of constituents in the instantiation of the rule, and $COMP_i$ provides a way of accessing the constituents.

We can now begin to clarify just how such a free-wheeling subcategorization scheme can be implemented in strict PATR-II. First of all, the method of getting the behavior of a Kleene star in context-free grammars is to use a recursive category, i.e., for each possible complement

category α_i, we have a rule:

$$VP_1 \rightarrow VP_2\ \alpha_i$$
$$\langle VP_2\ syncat \rangle = \langle \alpha_i\ cat \rangle \oplus \langle VP_1\ syncat \rangle$$

We add a rule to start the recursion:

$$VP \rightarrow V$$
$$\langle VP\ syncat \rangle = \langle V\ syncat \rangle$$

We merely require that a "full-fledged" VP is one whose *syncat* is the empty list Λ. It can be easily proved that this grammar weakly generates the same language the previous one(s) did. The difference, of course, is that the structure is now nested, not flat.

Finally, the question remains as to how lists and the $\oplus$ operation can be encoded. Lists can be encoded recursively as either a special symbol denoting the empty list, Λ, or pairs containing a list element and a list. We shall call these two parts *first* and *rest*. The *syncat* arc of a verb will then have a value something like

```
[first: α1
 rest:  [first: α2
         rest: ···
               [first: αm
                rest:  Λ]···]]
```

The previous grammar can now be expressed as

$$VP_1 \rightarrow VP_2\ \alpha_i$$
$$\langle VP_2\ syncat\ first \rangle = \langle \alpha_i\ cat \rangle$$
$$\langle VP_1\ syncat \rangle = \langle VP_2\ syncat\ rest \rangle$$

$$VP \rightarrow V$$
$$\langle VP\ syncat \rangle = \langle V\ syncat \rangle$$

We have seen a smooth progression from flat to nested structure to deal with the same problem of subcategorization. The progression involved moving the information about constituency from phrase structure rules to subcategorization information in the lexicon. Indeed, any context-free grammar can undergo such a transformation to yield an equivalent PATR-II grammar that has only one nonunary rule and preserves the weak-generative character of the language. (See Appendix C.) In effect, we just move all the syntactic information into the lexical

entries, so that the same PATR-II grammar skeleton can be used to model any CF grammar. Because the construction is local, the two methods can be combined freely. It is this aspect that we take advantage of in the transformation of verb phrase rules.

4.2.4.2 Complex Subcategorization

By far the most important comparisons are the similarities rather than the differences between the flat and the nested methods of handling VP structure. These are embodied in the progression of grammars described above. The techniques encode the same information in ways that reflect the direct isomorphisms between them. However, the nested technique for subcategorization can be extended to allow verbs to subcategorize relative to any aspect of the DAG associated with a complement, not just the category. The grammar above can be rewritten as shown below to allow arbitrary information about complements to be subcategorized for by unifying the elements of the syncat list with the whole DAG associated with the complement, not just the *cat* subDAG.

$VP_1 \rightarrow VP_2\ \alpha_i$

$\langle VP_2\ syncat\ first \rangle = \langle \alpha_i \rangle$

$\langle VP_1\ syncat \rangle = \langle VP_2\ syncat\ rest \rangle$

$VP \rightarrow V$

$\langle VP\ syncat \rangle = \langle V\ syncat \rangle$

4.2.5 Coordinating Syntax and Semantics Revisited

We now return to our discussion of the coordination of syntactic complement structure and semantic argument structure. Our grammar so far has the complement structure of the verb recorded in the feature *syncat* and the semantic structure in the feature *trans*. Since all of the information for the mapping is thus available in the lexical entry, we can perform the mapping directly by unifying the translations of the various subcategorized elements with the various argument positions. For symmetry, we add the preverbal NP to the *syncat* list so that it too can be unified into the translation. Our grammar becomes

$S \rightarrow NP\ VP$

$$
\begin{aligned}
\langle S\ head \rangle &= \langle VP\ head \rangle \\
\langle NP\ head\ agr \rangle &= \langle S\ head\ agr \rangle \\
\langle VP\ syncat\ first \rangle &= \langle NP \rangle \\
\langle VP\ syncat\ rest \rangle &= \lambda
\end{aligned}
$$

$VP_1 \rightarrow VP_2\ NP$

$$
\begin{aligned}
\langle VP_1\ head \rangle &= \langle VP_2\ head \rangle \\
\langle VP_2\ syncat\ first \rangle &= \langle NP \rangle \\
\langle VP_1\ syncat \rangle &= \langle VP_2\ syncat\ rest \rangle
\end{aligned}
$$

$VP \rightarrow V$

$$
\begin{aligned}
\langle VP\ head \rangle &= \langle V\ head \rangle \\
\langle VP\ syncat \rangle &= \langle V\ syncat \rangle
\end{aligned}
$$

Uther:

$$
\begin{aligned}
\langle cat \rangle &= np \\
\langle head\ agr\ number \rangle &= singular \\
\langle head\ agr\ person \rangle &= third \\
\langle head\ trans\ ref \rangle &= uther'
\end{aligned}
$$

Arthur:

$$
\begin{aligned}
\langle cat \rangle &= np \\
\langle head\ agr\ number \rangle &= singular \\
\langle head\ agr\ person \rangle &= third \\
\langle head\ trans\ ref \rangle &= arthur'
\end{aligned}
$$

knights:

$$
\begin{aligned}
\langle cat \rangle &= v \\
\langle head\ agr\ number \rangle &= singular \\
\langle head\ agr\ person \rangle &= third \\
\langle head\ trans\ pred \rangle &= knight' \\
\langle head\ trans\ arg1 \rangle &= \langle syncat\ rest\ first\ head\ trans \rangle \\
\langle head\ trans\ arg2 \rangle &= \langle syncat\ first\ head\ trans \rangle \\
\langle syncat\ first\ cat \rangle &= np \\
\langle syncat\ rest\ first\ cat \rangle &= np \\
\langle syncat\ rest\ rest \rangle &= \Lambda
\end{aligned}
$$

4.2.6 Auxiliaries

Handling auxiliary verbs is a related question. It seems that here a nested structure (as

in GPSG or PSG) is relatively well agreed upon. Thus, a rule of the form

$$VP_1 \rightarrow V\ VP_2$$

$$\begin{aligned} \langle VP_1\ head\rangle &= \langle V\ head\rangle \\ \langle V\ head\ aux\rangle &= + \\ \langle V\ syncat\rangle &= \langle VP_2\rangle \\ \langle VP_1\ syncat\rangle &= \langle VP_2\ syncat\rangle \\ \langle VP_1\ syncat\ rest\rangle &= \Lambda \end{aligned}$$

would suffice to handle auxiliaries for the nested-structure grammar. Here the *syncat* of the V will require certain features to be obtained on the sibling VP (VP_2), say that its *form* feature be *nonfinite*. By making *form* a *head* feature, we guarantee that the form of a VP comes from its first auxiliary, since the auxiliary is the head of its VP ancestor (VP_1). Finally, all the complements of VP_2 must be attached before permitting auxiliaries, and the *syncat* feature—now possessing information only about the preverbal constituent—passes from lower to upper VP, that is, from VP_2 to VP_1.

Note that the verb is required to have a + value for the *aux* feature. The VP → V rule presented earlier must be augmented by the restriction that the *aux* feature be −.

4.2.7 Adverbial Modifiers and the Generalized Wasow Effect

Modifiers can be easily dealt with in the nested-structure framework by a single rule, e.g.,

$$VP_1 \rightarrow VP_2\ ADVP$$

$$\langle VP_1\ syncat\rangle = \langle VP_2\ syncat\rangle$$

This rule allows adverbials to occur freely among the complements of a verb, embodying the so-called *Generalized Wasow Effect*[4], which is evident in such sentences as

1) Uther gave Lancelot on Thursday a sword.

Questions about how semantics would be affected and what head features should be percolated are as yet unresolved.

[4]The phenomenon and its name were brought to our attention by Ivan Sag.

4.2.8 An Implementation Notation for Grammar Writing

The PATR-II formalism can be viewed as a formal language for defining natural-language grammars. Unfortunately, as with many formal languages, the notation we have described so far is somewhat clumsy and verbose. Furthermore, there is no way to capture certain generalizations about the lexicon that a user might want to encode. We shall now describe a specific implementation of a natural-language-processing system whose underlying formalism is PATR-II and whose users are able to tailor the notation to their intended use of the formalism. As before, the intention is not to impose any particular usage, but to allow users to design their own mode of operation. The utilization of the formalism that has been described in this section has benefited from the notation, but so would many other implementations based on different strategies. The current PATR-II implementation supports the notation. Without it, our lexicon would be much more redundant.

4.2.8.1 Templates

Lexical items often share a great deal of structure because of their intended application or similarities in the way they function. We would like to define *template* DAGs that can be combined to form the lexical items in such cases. For instance, many verbs in English will share certain subcategorization information, such as a single noun-phrase complement that comprises the second argument of the predicate/argument structure. We can define a template called *Transitive* to encode this information:

Let *Transitive* be

$$
\begin{aligned}
\langle \textit{syncat first cat} \rangle &= \textit{np} \\
\langle \textit{syncat rest first cat} \rangle &= \textit{np} \\
\langle \textit{syncat rest rest} \rangle &= \Lambda \\
\langle \textit{head trans arg1} \rangle &= \langle \textit{syncat rest first head trans} \rangle \\
\langle \textit{head trans arg2} \rangle &= \langle \textit{syncat first head trans} \rangle \\
\langle \textit{head aux} \rangle &= \textit{false}
\end{aligned}
$$

Templates for *V* and *3sing*, respectively, can encode the fact that the word is a verb and that it is in the third person singular form.

Let *V* be

$$<cat> = V$$

Let *3sing* be

$$<head\ agr\ number> = singular$$
$$<head\ agr\ person> = third$$

The lexical entry for *knights* then becomes

knights:

V Transitive 3sing
$$<head\ trans\ pred> = knight'$$

4.2.8.2 Path Abbreviations

Like DAGs, path specifications can be abbreviated by using the same syntax. For example, the path abbreviation

Let *Pred* be

$$<head\ trans\ pred>$$

allows the same lexical item, *knights*, to be encoded

knights:

V Transitive 3sing
$$Pred = knight'$$

In summary, the use of templates and path abbreviations to tailor an implementation of PATR-II to a particular intended usage allows the grammar writer to capture the generalizations pertinent to that usage, at the same time facilitating the task of grammar writing and debugging by partitioning the grammar writing process into modules. Lexical rules provide a similar tool for accomplishing these objectives.

4.2.8.3 Lexical Rules

To encode the relationships among various lexical items—for instance, between the passive and active forms of a verb—we need a notion of a *lexical rule*. A lexical rule takes

as input a single DAG and generates an output DAG by means of unifications. These DAGs are denoted by the metavariables *in* and *out*, respectively.

As an example, we first discuss the active-passive dichotomy. Rather than generate the active from the passive or *vice versa*, we generate both of them from a protoentry for the verb whose *syncat* feature is exactly like the *syncats* presented previously, except that the final node is not marked with a Λ and an arc *<syncat tail>* is added pointing to the final node in the *syncat* list. The *Transitive* template now looks like the following:

Let *Transitive* be

$$\begin{aligned} \textit{<syncat first cat>} &= \textit{np} \\ \textit{<syncat rest first cat>} &= \textit{np} \\ \textit{<syncat rest rest>} &= \textit{<syncat tail>} \\ \textit{<head trans arg1>} &= \textit{<syncat first head trans>} \\ \textit{<head trans arg2>} &= \textit{<syncat first rest head trans>} \\ \textit{<head aux>} &= \textit{false} \end{aligned}$$

A lexical rule *active* is now defined to take a protoentry whose *syncat* was generated in this form as input and to generate an entry whose *<syncat tail>* is Λ. *Passive*, on the other hand, takes the same protoentry and moves the first element of the *syncat* list to the end of the list (the tail), thus making it a postverbal complement and making the previous leftmost postverbal complement the subject. Formally, expressed, we have

Define *Active* as

$$\begin{aligned} \textit{<out cat>} &= \textit{<in cat>} \\ \textit{<out head>} &= \textit{<in head>} \\ \textit{<out head voice>} &= \textit{active} \\ \textit{<out syncat>} &= \textit{<in syncat>} \\ \textit{<out syncat tail>} &= \Lambda \end{aligned}$$

Define *AgentlessPassive* as

$$\begin{aligned} \textit{<out cat>} &= \textit{<in cat>} \\ \textit{<out head>} &= \textit{<in head>} \\ \textit{<out head voice>} &= \textit{passive} \\ \textit{<out syncat>} &= \textit{<in syncat rest>} \\ \textit{<out syncat tail>} &= \Lambda \end{aligned}$$

Define *AgentivePassive* as

$$\begin{aligned} \textit{<out cat>} &= \textit{<in cat>} \\ \textit{<out head>} &= \textit{<in head>} \end{aligned}$$

$$
\begin{aligned}
\langle out\ head\ voice \rangle &= passive \\
\langle out\ syncat \rangle &= \langle in\ syncat\ rest \rangle \\
\langle out\ syncat\ tail\ first\ cat \rangle &= pp \\
\langle out\ syncat\ tail\ first\ lex \rangle &= by \\
\langle out\ syncat\ tail\ first\ head\ trans \rangle &= \langle in\ syncat\ first\ head\ trans \rangle
\end{aligned}
$$

The operation of the three lexical rules on the protoentry for the verb ***knight*** is shown as an example. First the protoentry:

```
[cat: v
 head: [aux: false
        form: nonfinite
        trans: [pred: knight
                arg1: <f1134>
                      []
                arg2: <f1138>
                      []]]
 syncat: [first: [cat: np
                  head: [trans: <f1134>]]
          rest: [first: [cat: np
                         head: [trans: <f1138>]]
                 rest: <f1140>]
          tail: <f1140>]]
```

The active form is

```
[cat: v
 head: [aux: false
        form: nonfinite
        voice: active
        trans: [pred: knight
                arg1: <f1134>
                      []
                arg2: <f1138>
                      []]]
 syncat: [first: [cat: np
                  head: [trans: <f1134>]]
          rest: [first: [cat: np
                         head: [trans: <f1138>]]
                 rest: <f1140>
                       Λ]
          tail: <f1140>]]
```

The agentless passive form is

```
[cat: v
 head: [aux: false
```

```
            form: nonfinite
            voice: passive
            trans: [pred: knight
                    arg1: []
                    arg2: <f1138>
                          []]]
      syncat: [first: [cat: np
                       head: [trans: <f1138>]]
                       rest: <f1140>
                             Λ]
               tail: <f1140>]]
```

Finally, the agentive passive:

```
     [cat: v
      head: [aux: false
             form: nonfinite
             voice: passive
             trans: [pred: knight
                     arg1: <f1134>
                           []
                     arg2: <f1138>
                           []]]
      syncat: [first: [cat: np
                              head: [trans: <f1138>]]
               rest: [first: [cat: pp
                              lex: by
                              head: [trans: <f1134>]]
                      rest: <f1140>
                            Λ]
               tail: <f1140>]]
```

4.2.8.4 Advantages of the Notation

It has been mentioned already that the notation we have introduced, and which is used throughout the lexicon, allows convenient abbreviations. Let us exemplify this claim by presenting a lexical entry in both its full and abbreviated forms. Here is an entry for the verb *seem*:

seem V - TakesIntransSbar Monadic Extrapos
- TakesInf RaisingtoS;

This entry collapses two verb entries for *seem*. Both entries share the category, i.e., both forms are verbs. The dashes indicate the start of each subentry. The following two sentences provide examples of the different syntactic environments that distinguish the two forms:

It seems that Uther sleeps.
Uther seems to sleep.

What follows are the definitions for the templates that are contained in the complex entry.

Let *be* be

```
                <cat>  =  v
           <head aux>  =  false
     <head trans pred> =  <sense>
```

Let *TakesIntransSbar* be

```
     <syncat first cat> =  sbar
         <syncat tail>  =  <syncat rest>
```

Let *Monadic* be

```
     <head trans arg1>  =  <syncat first head trans>
```

Let *TakesInf* be

```
         <syncat first cat>  =  np
    <syncat rest first cat>  =  vp
<syncat rest first headform> =  infinitival
         <syncat rest rest>  =  <syncat tail>
```

Let *RaisingtoS* be

```
         <head trans arg1>  =  <syncat rest first head trans>
<syncat rest first syncat first>= <syncat first>
```

The first subentry also contains a name of a lexical rule, Extrapos. Here is the rule:

Define *Extrapos* as

```
              <out cat>  =  <in cat>
             <out head>  =  <in head>
         <out head aux>  =  false
      <out head agr per> =  p3
   <out syncat first cat> =  np
   <out syncat first lex> =  it
        <out syncat rest> =  <in syncat rest>
```

$$\begin{aligned} \langle out\ syncat\ tail \rangle &= \langle in\ syncat\ tail\ rest \rangle \\ \langle in\ syncat\ tail\ first \rangle &= \langle in\ syncat\ first \rangle \end{aligned}$$

Without the notational tools introduced in this section we would have to write the following two verb entries for the two forms of *seem*:

$seem_1$:

$$\begin{aligned} \langle cat \rangle &= v \\ \langle head\ aux \rangle &= false \\ \langle head\ trans\ pred \rangle &= \langle sense \rangle \\ \langle syncat\ first\ cat \rangle &= np \\ \langle syncat\ tail \rangle &= \langle syncat\ rest\ rest \rangle \\ \langle head\ trans\ arg1 \rangle &= \langle syncat\ rest\ first\ head\ trans \rangle \\ \langle syncat\ first\ lex \rangle &= it \\ \langle syncat\ rest\ first\ cat \rangle &= sbar \\ \langle head\ agr\ per \rangle &= p3 \end{aligned}$$

$seem_2$:

$$\begin{aligned} \langle cat \rangle &= v \\ \langle head\ aux \rangle &= false \\ \langle head\ trans\ pred \rangle &= \langle sense \rangle \\ \langle syncat\ first\ cat \rangle &= np \\ \langle syncat\ rest\ first\ cat \rangle &= vp \\ \langle syncat\ rest\ first\ head form \rangle &= infinitival \\ \langle syncat\ rest\ rest \rangle &= \langle syncat\ tail \rangle \\ \langle head\ trans\ arg1 \rangle &= \langle syncat\ rest\ first\ head\ trans \rangle \\ \langle syncat\ rest\ first\ syncat\ first \rangle &= \langle syncat\ first \rangle \end{aligned}$$

In fact, these entries are the structures that are built from the "short" lexical entry when the word *seem* is encountered in an input sentence.

But our notation does not only allow convenient abbreviations; it also plays an important role in the linguist's use of the formalism. The actual format of the rules and lexical entries written by the linguist can be detached from the formalism. The grammars look more like those to which he is accustomed. Moreover, and perhaps most importantly, grammar writers can use the notational tools to express generalizations they could not state in the "pure" unification notation of the formalism. The fact that the DAGs associated with a syntactically motivated verb class like raising-to-object share some structure can be expressed in a nonredundant way, even if the amount of structure held im common cannot be encoded in a single unification

statement. The linguistic observation that all English modals are finite is expressed by including the template Finite in the definition of the template Modal.

The definition of the notational tools can also be used by the grammar writer to induce constraints upon the form and power of the grammar. One could reserve lexical rules for certain types of regularities such as relation-changing rules. It is quite conceivable that, at some point, the rules and lexical entries of our grammars will contain nothing but justified abbreviations of the kind introduced above.

4.2.9 Future Research on Uses of the Formalism

Clearly, the coverage of the grammar needs further expansion. But there are also more basic questions that require closer attention than how to handle other grammatical phenomena. The linguistic status of templates and lexical rules needs to be determined. One could adopt a simple view and use lexical rules every time the power of pure unification with a template does not suffice, i.e., whenever changing to the graph structure of lexical entries requires more than the simple addition of arcs and nodes. It would be more gratifying, though, if one had a clearer correspondence between the use of notational tools, on the one hand, and classes of linguistic regularities, on the other.

Another set of problems arises with the planned integration of non-truth-conditional and pragmatic information. If the truth-conditional part of the semantics of a phrase is incorporated in the DAG, there is no obvious reason to exclude the presuppositional elements of its meaning. The details of such a solution as well as of its interaction with discourse representations need to be worked out in the course of further research.

4.3. The Current PATR-II Implementation

4.3.1 Overview

The development of the PATR-II implementation took place on the SRI-AI DEC 2060

time-sharing system operating under TOPS-20. The original implementation is written in INTERLISP-10. In order to integrate PATR-II with the other components of KLAUS, the prototype INTERLISP version needed to be transported to a LISP machine. This version now runs on a SYMBOLICS 3600 in ZETALISP. A third version of PATR-II was programmed in PROLOG. This implementation does not include all the components of the prototype. It served mainly as a testbed for a structure-sharing unification algorithm.

The prototype implementation has five major program components: a set of top-level functions; a component for building and handling the internal lexicon; the morphology component; a context-free parser; a set of functions for structure unification. The grammar consists of a set of syntactic rules, a lexicon for basic word forms, a set of affix lexicons, definitions of lexical rules, templates and path names, and a set of finite state automata representing the morphophonemic regularities of English.

4.3.2 Implementation of the Basic Formalism

4.3.2.1 Top Level and User Interface

The top-level component starts the program, initializes global variables, sets user privileges, and runs the user interface. The main function for the user interface is COMMANDS. It will prompt the user with "command or sentence to parse." At this level, the user can give commands that load and clear grammars, parse sentences, debug, trace, and edit the grammar and lexicon, save any desired versions of the grammar, or save the whole system.

If the user input is enclosed in parentheses, the expression will be evaluated as an INTERLISP S-expression. If a sentence is given instead of a command, PATR-II will attempt to parse it turning control over to the parser for this purpose. The parser activates lexical lookup, morphological analysis, phrase structure building, and graph unification. If parses are found, the corresponding semantic translations will be printed out.

4.3.2.2 The Lexicon Functions

PATR-II actually has several lexicons: a stem (or root) lexicon and several small affix lexicons. Lexicons written in the notation described in Section 4.2.8 are stored internally as letter trees. The lexical information of an entry in these trees is associated with its last letter. The trees are used as discrimination networks for lexical lookup. There are functions that add, delete, display, and change entries. Other functions build new internal lexicons from inputted lexicon files or write out letter trees in the linguistic format.

4.3.2.3 Morphological Analysis

The lexicon for a language-processing system should not have to list the full morphological paradigm for each entry when there are many indications of the productivity of morphological rules for such processes as plural formation, conjugation, and English genitive inflection. On the other hand, the regularities that govern these processes are quite different from those entailed in syntactic processes and, moreever, it is impossible to separate morphological from phonological rules. Therefore, one often speaks about the morphophonemic component of a grammar. The design of PATR-II takes the special status of morphophonemic processes into account by assigning them to a separate component of the system: the morphological analyzer.

Our morphological analyzer is based on a recent implementation of Kimmo Koskenniemi's "bi-level model" for morphological analysis and synthesis [64]. This implementation was developed in INTERLISP as a course project at the University of Texas under the direction of Lauri Karttunen [57].

Two-level rules do not describe transformations of segment sequences in the same way as do rules of generative phonology. They are simply descriptions of correspondences between lexical and surface forms. In this respect the model resembles old-fashioned structural phonology, although it also differs from the letter in several important ways. Just as in structural phonology, in the two-level model there are no rule interactions, no relationships such as the bleeding or feeding that result from the sequential application of rules, so that subsequent rules apply to the output of earlier ones.

Like the other parts of the PATR-II processor, the morphological component is language-independent.

Morphological rules are represented in the processor as automata--more specifically, as finite-state transducers. There is a one-to-one correspondence between the rules and the automata. The idea of compiling rules into finite-state machines comes originally from Martin Kay and Ronald Kaplan [55]. In addition to the functions that analyze morphological forms by running the finite-state automata there are functions that compile and merge these automata from sets of phonological rules.

4.3.2.4 The Parser

The parser of the INTERLISP prototype PATR-II is a context-free, bottom-up chart parser without lookahead. It was inspired by the Bear-Karttunen PSG parser , which in turn is based on Dan Chester's implementation of the Cocke, Kasami, Younger algorithm (refer to [6] for a description of the parser and algorithm).

Before a new constituent is added to the chart, the DAGs of parent and children nodes are selectively unified by the graph unification component according to the unifications listed in the body of the applied rule. The completed edges of the chart of the PATR-II parser include pointers to the DAGs associated with the nodes.

The treatment of long-distance gap-filler dependencies is based on the opinion that the phenomenon is so general that the processor and not the grammar should be responsible for introducing and percolating gaps. Consequently, no grammar rules have to be duplicated to account for gap production. The current solution resembles the one in the PSG parser: the parser simply "assumes" a trace between every two adjacent words in the input. These traces can stand for NPs or PPs. Their agreement features are carried up the tree and are unified in the end with the filler's agreement features.

We intend to replace the parser with a "smarter," more predictive one later, that will recognize potential gaps only at places where they can really occur. We also want to investigate

how a phrase-linking solution of the type proposed by Peters and Ritchie [100] could be implemented in the PATR-II formalism.

4.3.2.5 Graph Unification

As described before, a PATR-II grammar is a set of context-free rules annoted with DAG unifications. It is useful to approach the problem of constructing a parser for PATR-II by determining which extensions should be made of a context-free parser to enforce the constraints specified by unifications. However, some parsing strategies that are reasonable for context-free grammars are not applicable or are just too inefficient for the extended parser. This is especially the case with parsers that require the context-free grammar to be rewritten into some normal form, because, in general, an annoted grammar cannot be rewritten this way.

The prototype PATR-II parser is a pure bottom-up context-free parser that applies all unifications associated with a rule when the latter is used to build a new phrase (parent) from its subphrases (children). When unifications are applied, both the parent phrase and the children may become more specified (more "instantiated"). Because of local or global ambiguities in the grammar, a given phrase may appear as the child of more than one parent phrase by virtue of rule applications that instantiate the child in different ways. For the parser to work properly, these alternative instantiations of the DAG associated with a phrase must be segregated. The prototype achieves this segregation by copying all the child phrases and their DAGs before trying to apply a rule (even if the rule application may eventually fail because of contradictory unifications or values).

The copying method used in the prototype is easy to implement, which is the main reason it was chosen. However, the wholesale copying of DAGs with each rule application requires far more space (and time) than the "structure sharing" method we are now considering. A further problem with the prototype parser is that the pure bottom-up strategy has difficulty in dealing with missing constituents (gaps) in a general manner.

The structure-sharing method of DAG representation and unification is closely modeled on the technique of the same name used in automatic theorem provers. This connection between

PATR-II parsers and theorem provers is more than coincidental, as it derives from the very close inherent relationship between PATR-II grammars and first-order theories.

Using structure sharing, the DAG associated with a phrase is represented by a pair of items: its "skeleton"—a DAG derived by compile-time application of all the unifications of a single rule; its "index," a number identifying the particular rule application that created this DAG. The index of a DAG is used to tag records ("bindings") that describe additions made to the DAG through unification. Bindings are stored in "binding environments." Although each alternative partial analysis of the input has its own binding environment, most of these environments share information because they have been derived (through alternative rule applications) from earlier partial analyses.

The standard structure-sharing technique, invented by Boyer and Moore [11], requires an amount of searching for the bindings of a DAG that, at worst, can be proportional to the size of the preceding analysis. Instead of this scheme, we have in our experiments with structure sharing adopted a "logarithmic tree" representation of binding environments and a parsing strategy that make binding lookup at worst logarithmic with respect to the size of the analysis. The parsing strategy used, which is a variation of the Earley parsing technique, has the further advantage of allowing gap-introducing rules with full generality. However, to achieve the logarithmic time bound forces us to copy new complete phrases as they are created, although partial analyses are still fully structure-shared. Since the trade-offs between this method and the standard structure-sharing one are difficult to identify theoretically, we plan to implement another version of the structure sharing PATR-II parser, using the standard Boyer and Moore method.

The Prolog implementation of PATR-II is based on an experimental structure-sharing parser of the kind described above.

4.3.3 PROLOG Implementation

Besides the INTERLISP and the LISP machine implementations of PATR-II, there exists also a PROLOG implementation of the basic formalism on the DEC-2060. It is based on the

experimental structure-sharing PATR-II parser described in Section 4.3.2.5. The Prolog program has run successfully with various PATR-II grammars, with an efficiency similar to that of the copying parser. Prolog has been useful for rapid "throw away" testing of alternative parsing mechanisms. The advantages of a structure-sharing parser are expected to dominate performance for larger grammars than our current ones, at which point it will become worthwhile to reimplement the parser using more efficient low-level coding on the LISP machines.

4.3.4 LISP Machine Implementation

For the integration of PATR-II with the other components of KLAUS, the prototype PATR-II implementation that had been developed in INTERLISP on a DEC 2060 had to be transported to a Symbolics 3600 LISP machine.

The transfer of PATR to the 3600 was done in such a fashion that further development could be done on the 2060 and, at the same time, make it relatively easy to retransport to the 3600. An initial effort to translate the INTERLISP-10 code into ZETALISP code directly by using the INTERLISP TRANSOR translator revealed a number of problems.

An alternative method was tried. Symbolics offers an INTERLISP Compatibility Package (ILCP) consisting of a translator that runs under INTERLISP and a run time package that runs on the 3600. The translator on the INTERLISP end mainly checks for upper/lowercase problems, handles comments, and other syntactic features of INTERLISP. The run time package on the 3600 provides a simulated INTERLISP environment. That is, many of the INTERLISP functions are defined to work as they do in INTERLISP. For instance, MAP takes its arguments in the order used by INTERLISP rather than the opposite order used by ZETALISP.

The disadvantage of this method is that the ILCP is a rather large set of software that is still being developed. It was necessary to rewrite all the INTERLISP I/O functions, as the supplied definitions did not cover PATR's particular usage of those functions. At present, this software package must be loaded each time the PATR code is loaded. However, it was decided

that running with the ILCP was easier and more reliable than using a special translator for the PATR code.

Some functions required by PATR, e.g., ASKUSER, were not available. This function was coded directly in ZETALISP to use normal mouse selection when possible.

When the PATR code was run on the 3600 under the ILCP, several coding problems were discovered. Most of these were avoidable (i.e., taking the CAR of an atom) and a patch was made on both the INTERLISP and ZETALISP versions of PATR.

The only serious difference between these two versions is in the treatment of case distinction. ZETALISP is indiscriminate; it translates all normal input into upper-case. INTERLISP-10, on the other hand, leaves all input in its original case.

PATR made extensive use of the lower/upper-case distinction, but, fortunately, most of this selectivity was aesthetic rather than essential. We were able to modify the code so that, in almost all cases, the program works regardless of whether or not lower and upper case are merged. In one or two places, where the difference could not be compensated for by coding, the code had to be hand-patched when translated from INTERLISP-10 to ZETALISP.

The above describe the differences in the running code of the two systems. Other differences are found in the user interface. Some of the utilities provided by the INTERLISP-10 system are moot on the ZETALISP system, e.g., EXIT and SAVE. Others, such as EDIT, were not directly available on the ZETALISP system. EDIT was coded to be as much like the INTERLISP-10 system as possible. DRIBBLE was a feature that could not be provided without a substantial coding effort.

The section of code for the user interface needed to be redone. As a result, a menu of the available commands is now permanently displayed on the screen. To choose a command, the user simply points the mouse and clicks one of the buttons (usually the left) on the mouse. In a few cases, clicking the middle or right button provides for different options of the basic command. For instance, clicking right on FASTLOAD allows the user to specify the system name first.

As PATR outputs text to the screen, it specifies to the 3600 for certain items of text what type of item is being displayed. For instance, when it outputs the name of a DAG, it informs the 3600 that it is outputting a DAG. If PATR later asks for the name of a DAG, the user may either type in a DAG or point the mouse to one of the printed DAG names. When he points to this DAG, a box appears around it, indicating that it is a possible answer. If he then clicks a button on the mouse, that DAG name is inputted.

Almost all of these user interface changes are in the top-level routine that takes commands from the user. Therefore, it can be just loaded in place of the INTERLISP-10 command interface. Any new commands can be easily added. A few changes were necessary in the body of the system where output is done so that, for instance, the 3600 is told when a DAG name is printed. These changes were also added to the INTERLISP-10 version.

The utilization of the display and user interface features of the 3600 have created a superior working environment. The menu-driven top-level functions, together with the multi-window display improve grammar and program development. New debugging and editing facilities that utilize the available ZETALISP function packages are still being added to the system.

4.3.5 Future Research on the Implementation

Among the related projects we want to undertake next are the implementation of the structure-sharing unification algorithm on the LISP machine, the development of a phrase structure parser with more predictive power, and a phrase-linking solution to unbounded dependencies [100].

4.4. Conclusion

Major parts of our implementation are a grammar formalism and an implementation notation designed to serve as a "programming language for linguists." That is, it is a powerful grammar-writing system that allows the encoding of many analyses of linguistic phenomena.

In the sense that the formalism does not attempt to characterize all and only the grammars of natural languages (though a more constrained theory might use the formalism as its "semantics" so to speak), it does not embody a linguistic theory. Instead, it is a tool linguists can use to express linguistic analyses formally; its implementation is a tool for testing such expressions.

The formalism and the notation for grammar writing proved to be adequate and convenient devices for writing grammars that cover the grammatical phenomena we have dealt with so far. The notation has also shown itself to be useful as a conceptual aid in the formulation of linguistic-research problems.

Our modular implementation, consisting of a top level, a parser, a unification component, and a morphological analyzer, makes it easy to replace any individual component.

The current implementation was designed as a research tool. This means that the advantages modularity and the convenience of modifying grammars as well as implementation had priority over efficiency. Nevertheless, the process of parsing and translating sentences of different degrees of complexity is performed at reasonable speed.

Appendix A. A Formal Definition of the Formalism

Definition 4.1. DAG:

A DAG defined over a finite set of labels A is either

- an atomic label $l \in A$, or
- a possibly empty set s of pairs $< l, v >$ where $l \in A$ and v is a DAG and s does not *cover* s. (Covering is defined recursively as follows: for all $< l, v > \in s$, s covers v and s covers anything covered by v. The atomic label l covers only itself.)

l is called the *attribute* or *feature* and v the *value* of the attribute.

Definition 4.2. Path:

A *path* is a sequence $< n, l_1, \ldots, l_m >$ (hereafter notated without commas so as to avoid confusion with other sequences) where n is a DAG and the l_i are atomic labels. Such a path *denotes* the node n_m where $< l_i, n_i > \in n_{i-1}$.

Definition 4.3. Grammar:

A PATR-II *grammar* is a sextuple $< N, T, A, R, L, S >$ where

- N is a finite nonempty set of nonterminals,
- T is a nonempty set of terminals,
- A is a finite set of labels, (usually a superset of $N \cup T$),
- R is a finite set of grammar rules (see below),
- L is a relation in $T \times D$, where D is the set of DAGs definable over A, and
- $S \in D$ is the start DAG.

Definition 4.4. Grammar rule:

A grammar rule has two parts:

- a context-free phrase structure rule with uniquely identified nonterminals, notated, e.g., $VP \rightarrow V\ NP_1\ NP_2$
- a set of unifications, notated as $m = n$ where m and n are DAGs or path specifications with nonterminal labels instead of DAGs as the first elements, e.g., $< VP\ l_1\ l_2 > = < V\ l_3 >$.

Definition 4.5. Admissibility:

A rule $l_0 \rightarrow l_1 \cdots l_m$ with unifications U *admits* a sequence of DAGs $< n_0, n_1, \ldots, n_m >$ iff

- if $l_i \in T$, then $< l_i, n_i > \in L$, and
- if $l_i \in N$, then the path $< n_i \; cat >$ denotes l_i (minus any subscripts), and
- for all $p_1 = p_2$ in U the node denoted by p_1 is the same as that denoted by p_2. A path $< l_i \; k_1 \cdots k_q >$ denotes a node n if and only if the path $< n_i \; k_1 \cdots k_q >$ denotes n.

Definition 4.6. Derivation:

A DAG n_0 *derives* a sequence of DAGs $< n_1, \ldots, n_m >$ if there is a rule $r \in R$ such that r admits $< n_0, n_1, \ldots, n_m >$. This is notated $n_0 \Rightarrow n_1 \cdots n_m$. The symmetric transitive closure of $\Rightarrow$ is notated $\Rightarrow^*$.

Definition 4.7. Language:

The *language* of a PATR-II grammar $G =< N, T, R, L, S >$ is the set $\{w \in T^* \mid S \Rightarrow^* w\}$

Definition 4.8. Unification:

The *unification* of two DAGs n_1 and n_2 is a DAG n where

- if $n_1 = n_2$, then $n = n_1$,
- if n_1 is atomic and $n_2 = \{\}$, then $n = n_1$, and similarly with n_1 and n_2 interchanged,
- if neither n_1 nor n_2 is atomic, then for all l such that $< l, v_1 > \in n_1$, and $< l, v_2 > \in n_2$, $< l, unify(v_1, v_2) > \in n$ and for all l such that $< l, v > \in (n_1 \cup n_2) - (n_1 \cap n_2)$. $< l, v > \in n$.

Appendix B. Some Grammars for Hard Languages

The following grammar accepts the non-context-free language $a^n b^n c^n$:

$S \rightarrow As\ Bs\ Cs$

$$\begin{aligned} \langle As \rangle &= \langle Bs \rangle \\ \langle Bs \rangle &= \langle Cs \rangle \end{aligned}$$

$As_1 \rightarrow As_2\ A$

$$\langle As_1\ succ \rangle = \langle As_2 \rangle$$

$As \rightarrow \epsilon$

$$\langle num \rangle = 0$$

$Bs_1 \rightarrow Bs_2\ B$

$$\langle Bs_1\ succ \rangle = \langle Bs_2 \rangle$$

$Bs \rightarrow \epsilon$

$$\langle Bs \rangle = 0$$

$Cs_1 \rightarrow Cs_2\ C$

$$\langle Cs_1\ succ \rangle = \langle Cs_2 \rangle$$

$Cs \rightarrow \epsilon$

$$\langle Cs \rangle = 0$$

a:

$$\langle cat \rangle = a$$

b:

$$\langle cat \rangle = b$$

c:

$$\langle cat \rangle = c$$

The following grammar accepts the non-indexed language $a^{2^{2^n}}$:

$S \rightarrow A$

$$\begin{aligned} \langle S\ m \rangle &= \langle A\ m \rangle \\ \langle A\ n \rangle &= 0 \\ \langle A\ p \rangle &= \langle A\ q \rangle \end{aligned}$$

$A \rightarrow B$

$$\begin{aligned} \langle A\ m\rangle &= 0 \\ \langle A\ n\rangle &= \langle A\ p\ succ\rangle \\ \langle A\ q\rangle &= \langle B\ m\rangle \end{aligned}$$

$A_1 \rightarrow A_2\ A_3$

$$\begin{aligned} \langle A_1\ m\ succ\rangle &= \langle A_2\ m\rangle \\ \langle A_1\ n\rangle &= \langle A_2\ n\rangle \\ \langle A_1\ q\ succ\rangle &= \langle A_2\ q\rangle \\ \langle A_1\ m\ succ\rangle &= \langle A_3\ m\rangle \\ \langle A_1\ p\rangle &= \langle A_3\ p\rangle \\ \langle A_1\ q\ succ\rangle &= \langle A_3\ q\rangle \\ \langle A_2\ p\rangle &= \langle A_3\ p\rangle \end{aligned}$$

a:

$$\begin{aligned} \langle cat\rangle &= B \\ \langle m\rangle &= 0 \end{aligned}$$

$B_1 \rightarrow B_2\ B_3$

$$\begin{aligned} \langle B_1\ m\ succ\rangle &= \langle B_2\ m\rangle \\ \langle B_1\ m\ succ\rangle &= \langle B_3\ m\rangle \end{aligned}$$

Appendix C. Conversion to Normal-Form PATR-II Grammars

In Section 4.2.4.1 we state that any context-free grammar can be converted to a PATR-II grammar with only one nonunary rule. The construction is as follows: Given a context-free grammar $< N, T, R, S >$, we construct a PATR-II grammar with the following rules:

$S' \rightarrow S'$

$$<S' \textit{ syncat}> = \lambda$$

$S' \rightarrow S_1\ S_2$

$$<S' \textit{ syncat}> = <S_1 \textit{ syncat rest}>$$
$$<S \textit{ syncat first}> = <S \textit{ cat}>$$

For every rule $\alpha \rightarrow \beta_1 \cdots \beta_n \in R$, add the rule

$S \rightarrow \alpha$

$$<S \textit{ syncat first}> = \beta_1$$
$$<S \textit{ syncat rest first}> = \beta_2$$

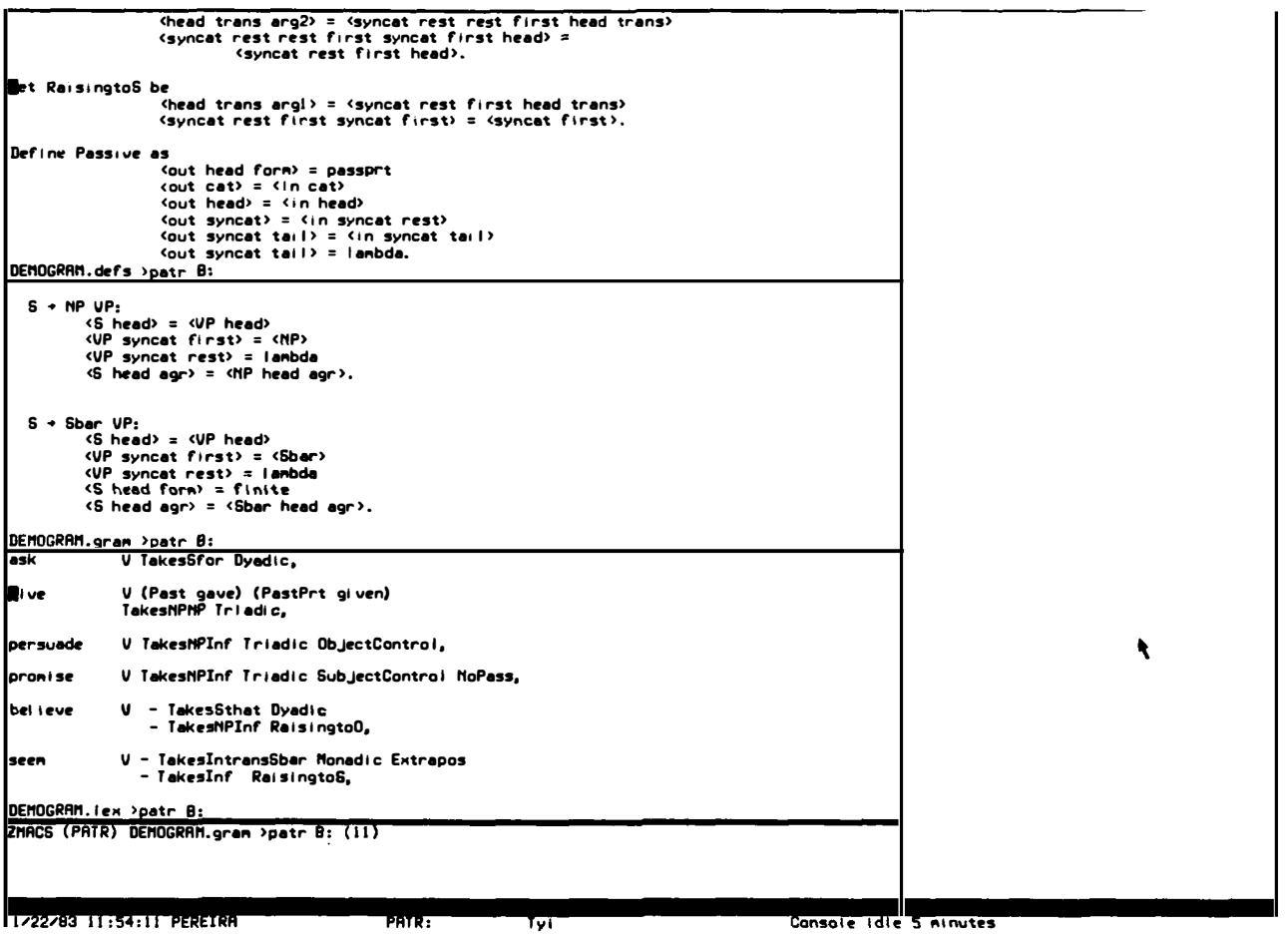
etc.,

and, for every β_i, add the rule

$S \rightarrow \beta_i$

Appendix D. Sample Rules of the PATR-II Grammar

The LISP machine screen below displays three editing windows with samples of definitions (one template and one lexical rule) in the upper window, of syntactic rules in the middle window, and of lexical entries in the lower window.



Appendix E. A Sample Dialog with PATR-II

The right-hand-side window (interaction window) of the following LISP machine freeze frame contains a three-sentence sample dialog with KLAUS. The left-hand-side window can be used for displaying the chart, DAGs, words, and rules that were built or used during the parsing of a sentence. There is a mouse-operated menu window on top of this display window. In the freeze frame, the DAG associated by the parser with the last sentence is displayed in the display window. Display window and interaction window are seperated by another menu window which represents the user's options at the top level of KLAUS.

For each of the sentences in the short dialog, the parser found exactly one parse but multiple scopings of quantifiers and tense operators. The selection of the desired scoping was performed in a temporary menu window using the mouse. Assertions as the first two sentences are accepted by KLAUS if they do not contradict with already known propositions. Possible responses to alternative questions are "YES", "NO", or "I DON'T KNOW". The answer to the third input sentence, which is an alternative question, is based on the knowledge the system gained through the first two input sentences.