TWINCLE: The WINdowing Computational Linguistics Environment

D. Terence Langendoen, Dept of Linguistics, The University of Arizona
Clinton L. Jeffery, Dept of Computer Science, The University of Arizona
Jon Lipp, Dept of Computer Science, The University of Arizona

Introduction

The TWINCLE project is an effort to develop a friendly computational environment for linguists to write and test grammars for natural languages, by enabling them to specify the components of those grammars using windowing tools and high-resolution graphics.[1] It has long been recognized that the main hindrance to the development of grammatical models for natural languages of any reasonable degree of complexity and detail is the inability of linguists and programmers to understand the complex interrelationships among the various parts of the model. Providing graphical representations of those interrelationships, and the ability to specify grammatical models in terms of those representations will enable linguists, even with little or no help from programmers, to build complex grammatical models to their own specifications.

In this phase of the TWINCLE project, we are designing an environment for supporting the development of *augmented phrase-structure grammars* (*attribute grammars* or *feature-structure grammars*) [Alle87], because this formalism is widely used in computational linguistics, is general enough to provide models for nearly every currently popular grammatical theory, and is relatively easy to visualize. Such grammars generally provide a set of *terminal elements*, which for our purposes we can identify with *lexical entries*, each consisting of a *label* (typically a string representing the conventional appearance of the entry in text), a *lexical class* and a set of attributes and values. They also generally provide a set of *nonterminal elements* consisting of a *grammatical class* and a set of attributes and values, and a set of *productions* or *rules*. for combining terminal and nonterminal elements together into representations (often, but not necessarily representable as trees, but always as sets of directed, connected graphs) of linguistic structures of potentially great length and complexity.

---

[1] This work was supported in part by the Advanced Telecommunications Research Program of the Department of Electrical and Computing Engineering and by the Cognitive Science Program of The University of Arizona.

Choice of Programming Language

Programming Language Alternatives

Certain very-high-level languages such as PROLOG and LISP are traditionally advocated for work in natural language processing and computational linguistics; such languages are characterized by their list processing capabilities, automatic memory management, and built-in data structures. These features make them appropriate for exploratory programming and rapid prototyping. Other languages, such as C, C++, Smalltalk, Actor, and Visual BASIC are more appropriate for writing programs with extensive graphical user interface components such as TWINCLE. The TWINCLE project faced strong requirements for both exploratory programming support and extensive graphical user interface facilities. It also had strong constraints: our limited financial resources forced us to consider only alternatives that would run with acceptable interactive speed on an Intel 486 processor and we had a very limited budget for software.

Neither PROLOG nor LISP was selected because we were aware of no implementation available that would meet our performance, budget, and user-interface requirements. It is possible that such a system exists, but most high-performance implementations of these languages are expensive and run only on workstation hardware.

The Intel platform does offer several affordable languages with which to write user interface software. C and C++ were rejected because they did not offer a high-enough level of interface to support exploratory programming. Actor and Visual BASIC suffer from various limitations and are not portable across window and operating systems. Among commercially available languages SmallTalk comes the closest to meeting our constraints. But although SmallTalk is highly suitable for rapid prototyping in the hands of expert SmallTalk programmers, it has a fairly steep learning curve and implementations of SmallTalk on Intel systems are neither inexpensive nor exceptionally fast.

The Icon programming language has extensive string and list processing capabilities that make it a natural choice for text-oriented applications [Gris90]. With the release of Version 8.6, Icon includes a high-level interface to the mouse and graphics capabilities of modern personal computer systems [Jeff92]. Both local expertise with Icon and its public domain status encouraged its selection for use in TWINCLE. The project started out using UNIX and X Window System software on the 486; when Icon's mouse and graphics capabil-

ities were ported successfully to IBM OS/2 2.0, porting TWINCLE to that environment required almost no changes to the source code.

Selected Features of Icon and Idol

Several features of Icon make it particularly appropriate to the domain of computational linguistics. These features include:

* Sophisticated string scanning makes it easier to do more sophisticated lexical and morphological analyses not possible with stock tools based on regular expressions.
* Direct support for a character-set data type as well as support for more general heterogeneous data structures with polymorphic operations.
* Sophisticated control structures that support goal-directed evaluation and control backtracking simplify the implementation of many algorithms.

In addition to these features, the object-oriented extension to Icon called Idol has features we considered important in a larger-scale effort:

* Classes allow extension of Icon's built-in repertoire of data types in a way that minimizes functional dependencies between different code modules.
* Inheritance allows new types to be specified in terms of existing types, encouraging code re-use. In the computational linguistics domain many components such as lexical items fall into natural hierarchies such as hyponymy that can be expressed directly with classes and inheritance.
* Idol also extends Icon with some basic non object-oriented features that are extremely useful in larger programs that are written in several files, such as constant declarations and source file inclusion.

Taken together, Icon's symbolic processing capabilities, X-Icon's user interface support, and Idol's object-oriented model make this environment very appropriate for computational linguistics applications.

Lessons learned about Icon and Idol in TWINCLE

Over the course of the project two valuable lessons were learned about Icon and Idol. The first and most significant result is that Icon's built-in operations would benefit from greater polymorphism. Polymorphism is the ability to use

one notation or set of operations on more than one type of data. Icon has a large number of polymorphic operations, and while these are useful we found that many operations are not as polymorphic as they could be.

For example, during the course of the implementation, we encountered a need to mix records and tables in our parse trees: records are compact and are used to efficiently represent lexical items, while tables are more flexible since arbitrary keys can be inserted into them. Icon's list, table, and record types are all heterogeneous, that is, structure elements may be any type and in particular such structures could contain some elements that are tables and some that are records.

In addition to this minimal requirement that Icon meets well, mixing records and tables requires that the parse engine be able to process nodes polymorphically; it applies the same operations irrespective of whether a given node is represented by a record or a table. In either case, linguistic features of the node are accessed by string name during parsing. The TWINCLE parse engine processes nodes irrespective of whether they are leafs represented as records or internal nodes represented as tables. But record fields are accessed using the . operator and tables are subscripted using the [ ] operator.

What was needed in this application was a way to access records by field name using a string instead of the . operator – a subscript operation analogous to table subscripting. The following Icon procedure implements element access by string name independent of whether its first argument is a table or a record.

```
procedure access(x, s)
  if type(x) == "table" then return x[s]
  # else x is a record
  every i := 1 to *x do {
    im := image(x[i])
    if im[*im - *s - 1:0] == "."||s then return x
  }
end
```

The problem with this procedure is that it is a slow way of doing a natural thing. The use of the image() function to obtain the records' field names is clumsy. A better solution is to make Icon's subscript operator more polymorphic to support record field access by string name. Our familiarity with the Icon implementation allowed us to make this tiny addition. By submitting it back to the Icon Project, the extension was made available to the rest of the Icon community as part of Version 8.7. It is likely that increasing the polymor-

phic behavior of other operations would result in similar improvements in the expressive power of Icon.

Making additions to a language is not something to be taken lightly however. Icon's subscript operator was already defined for records when the supplied index was an integer, e.g. r[2] refers to the second field of the record independent of its field name. Since Icon automatically converts strings to integers when necessary, the extension had to make sure it did not conflict with existing semantics: it can not blindly assume that a record subscripted by a string is a field name, but must first check to see if the string can be converted to an integer and applied as an integer index. Since the set of strings that can be converted to integers is disjoint with the set of strings that make valid record field names, the addition poses no semantic conflict.
The second major thing we learned about the implementation language used concerns the object-oriented preprocessor Idol. Idol object instances are very similar to records–they consist of named fields that may normally be accessed only by operations defined in the instance's class. In order to access an object's fields the object must support field-access methods.

We wished to use objects for the primary lexical structures in our system in order to take advantage of the inheritance mechanism. But lexical elements have a hierarchical feature structure that may be several levels deep. This sort of structure is very naturally expressed using records. Implementing such a structure using objects would entail that many field-access procedures be written, and the end result would be slower than record field accesses.

Since Idol objects are implemented using Icon records, it was a simple matter to optionally lift the field-access restriction and allow object fields to be accessed by normal Icon record field access. This change to Idol compromises its strict object-oriented nature in favor of increased utility. It does not reduce Idol's expressive power or object-oriented functionality, but it reduces objects to more of an organizational tool for Icon programs instead of an all encompassing execution paradigm. This might be viewed as a Simula-style object orientation instead of a SmallTalk-style object-orientation. The more restrictive model is still available via an option to the Idol translator.

The Computational Environment

We have integrated working prototypes of five environments for the development of attribute grammars. These environments may be used for specifying the following parts of grammars: lexical attributes, lexical classes, lexical

entries and syntactic rules. The fifth environment is a parser for testing the grammar developed in the other four environments. We refer to this environment as TWINCLE0. TWINCLE0 is written entirely in Icon; we have decided to defer use of Idol until we have tested the environments on a variety of small-scale grammars. TWINCLE0 currently runs under the OS/2 operating system.

When TWINCLE0 is invoked, a menu appears which allows the user to select an environment to use, as shown in Figure 1.

Figure 1. Main Menu

After completing work in any environment, one is returned to the main menu, from which one may also exit the program.

The Lexical Attribute Environment

The lexical attribute environment is the place where one specifies attributes and their possible values for lexical items generally. The environment prompts the user for a **label** for each lexical attribute (its name), and the **type** of its values. Five types of attribute values are permitted: binary (boolean), numeric, symbolic, and structured. One selects the type from a pop-down menu, as shown in Figure 2, where ***binary*** has been chosen for the attribute PROPER. If ***binary*** is chosen as the value type for a particular lexical attribute, then the possible values *true* and *false* are supplied for that attribute. As Figure 2 also shows, a scrolling window lets the user see the lexical attributes that have already been defined.

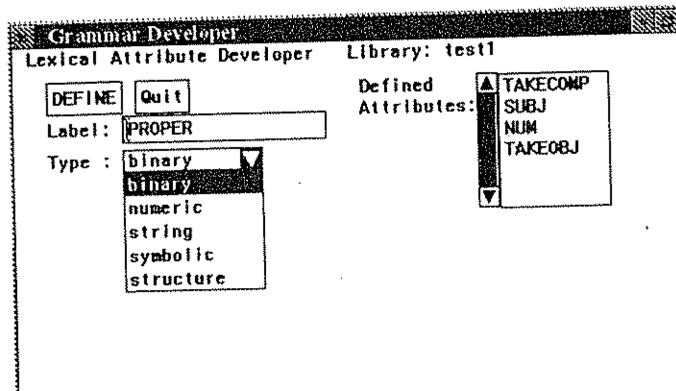

Figure 2. Lexical attribute developer showing selection of binary-valued attribute

If one selects the ***symbolic*** type, then a window appears for editing the possible symbolic values that the particular attribute may have. In Figure 3, we show the point at which the user has specified the possible values *sing* and *plural* for the attribute NUM.

If one selects the ***structured*** type, then one is prompted for the attributes that may appear as the values of the structured attribute. Only previously defined lexical attributes may be selected, as shown in Figure 4, in which the structured attribute SUBJ takes on as its possible values the attribute NUM and its associated values. The ***numeric*** and ***string*** attribute types have not yet been implemented.

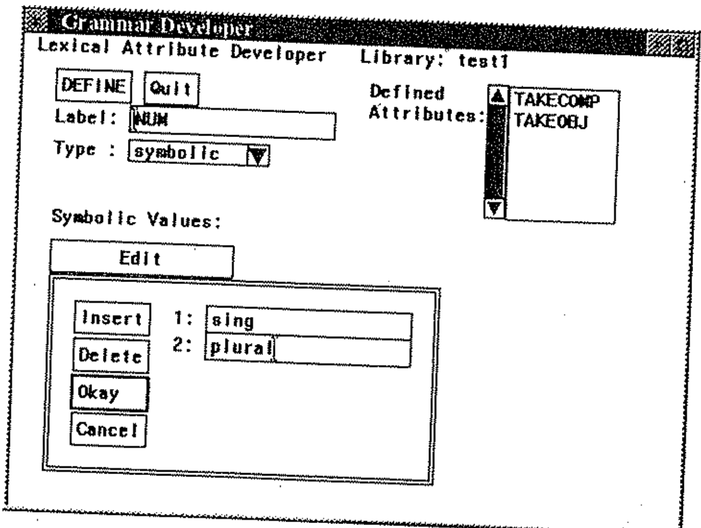

Figure 3. Lexical attribute developer showing selection of symbolic-valued attribute

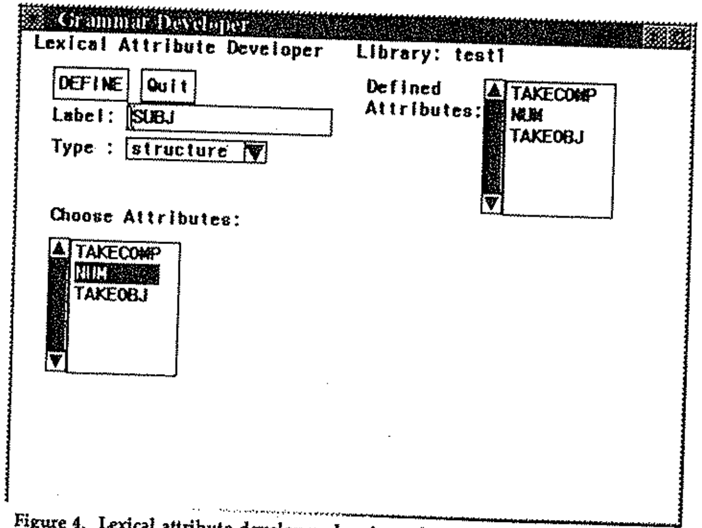

Figure 4. Lexical attribute developer showing selection of structured-value attribute

The Lexical Class Developer

The lexical class developer is the environment in which one specifies lexical classes and their associated attributes. As Figure 5 shows, one is prompted for the label of the lexical class, and a scrolling window shows the previously defined attributes, from which one selects the attributes that are desired for the particular lexical class. The already defined classes are shown in a scrolling window off to the right. In this illustration, the label of the class is *N*, and the attributes that are about to be selected for it are *TAKEMOD*, *PROPER*, and *NUM* (the first two of which have been previously defined to be of *binary* type, and the third to be of *symbolic* type.

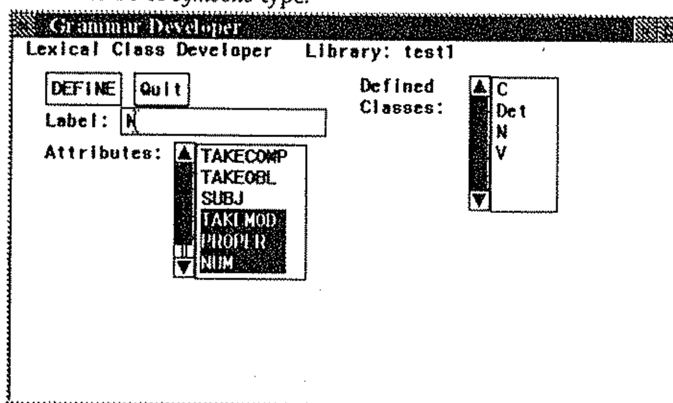

Figure 5. Lexical class developer showing a specification for the lexical class *N*

The Lexical Entry Developer

The lexical entry developer environment permits the user to specify lexical entries as belonging to particular lexical classes with particular lexical attributes and values. In this environment, the user must provide a label for the lexical entry, and must select a class from one of the classes defined in the lexical class developer; these are displayed in a pop-down window. Once the class is selected, the labels for all of the attributes defined for that class are displayed, and one may specify values for each of them. The previously defined possible values are displayed in pop-down windows, as illustrated in Figure 6, in which the pop-down window for the values of the NUM attribute

is shown. In a scrolling window off to the right, the already defined lexical entries are displayed.

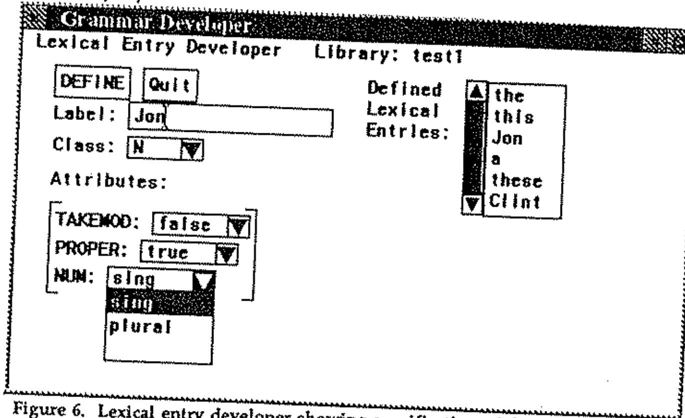

Figure 6. Lexical entry developer showing specification of the entry *Jon*

We have not yet implemented a lexical entry editor, so that at the moment the only way to redefine a lexical entry is to reenter it from scratch. If one does so, TWINCLE0 asks whether you want to replace the previously defined entry with the new one, as shown in Figure 7.

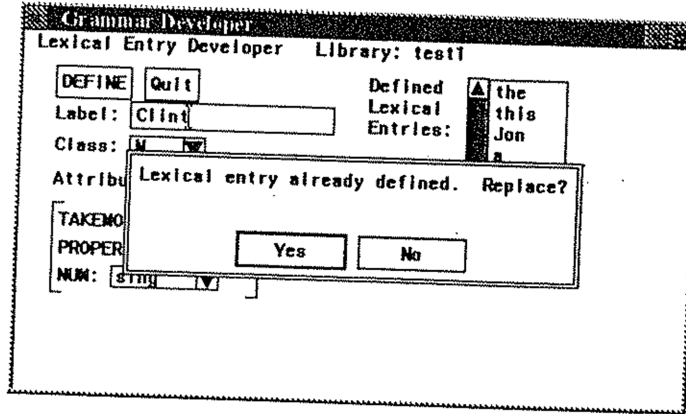

Figure 7. Lexical entry developer showing prompt to replace previously defined entry

The Rule Developer

Given a set of lexical entries, TWINCLE0 provides a syntactic rule developer environment for specifying rules for combining them into larger expressions and for combining these results further, until the largest linguistically definable entities are created. At the outset, the only rules that one can write are those that combine lexically-defined classes, so that one has to start the process of rule writing from bottom up.

The rule developer environment is in two parts. In the first part, called the ***rule specification*** environment, the user specifies the **arity** of the rule (the default arity is 2), and the labels of its **children** (the elements that would appear on the right-hand side of the rule if it is written as a phrase-structure rule). The labels that TWINCLE0 knows about are displayed in pop-down windows associated with each child. In Figure 8, the user has selected the labels *Det* and *N*. As rules are added to the grammar, and new labels for syntactic categories are defined, the new labels appear in these windows.

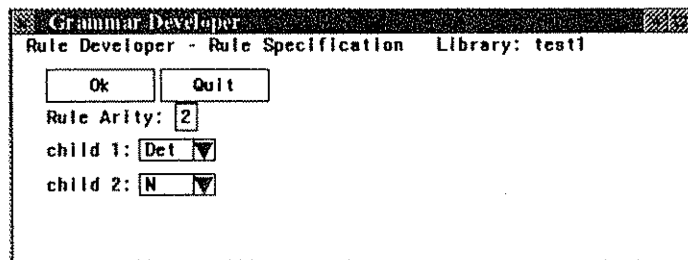

Figure 8. Specifying the children of a syntactic rule

The second part of the rule developer is called the ***test formulation*** environment; it enables the user to supply the label for the **parent** of the rule (the element that would appear on the left-hand side of the rule if it is written as a phrase-structure rule; this properly belongs to the rule specification environment, and we will be moving it soon to that window). In this environment also, the user indicates from which child the parent inherits attributes (i.e., which child is the head of the construction), and may specify a pointer from the parent to a non-head child. Moreover, the user may specify **tests** of two sorts, by selecting the ***specify tests*** option. One type of test is a **simple value** test, in which a particular attribute in one of the children is required to have a certain value. The other is an **equality** test, in which two attributes (typically, but not necessarily, in different children) are required to be the same. In

Figure 9, we show the development of the rule *NP* --> *Det N*, in which *N* is identified as the head, *Det* is identified as the value of the SPEC attribute of the *NP*, and the NUM attributes of the *Det* and the *N* are required to be equal.

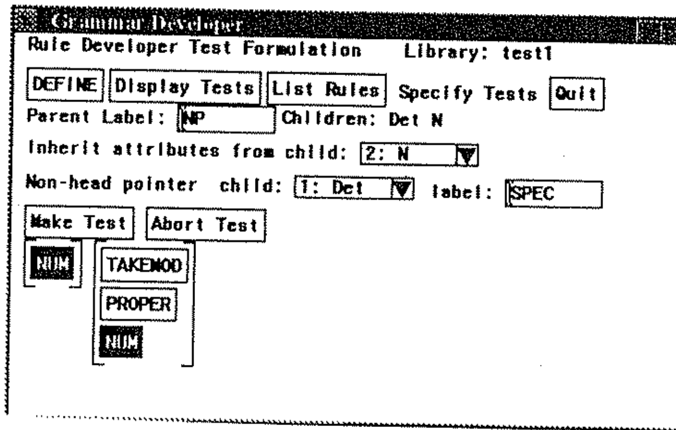

Figure 9. Rule developer showing specification of the rule *NP* --> *Det N*

The test formulation environment also enables the user to display the tests that have already been defined for the rule so far (Figure 10), and the rules that have already been defined at the point the request is made (Figure 11).

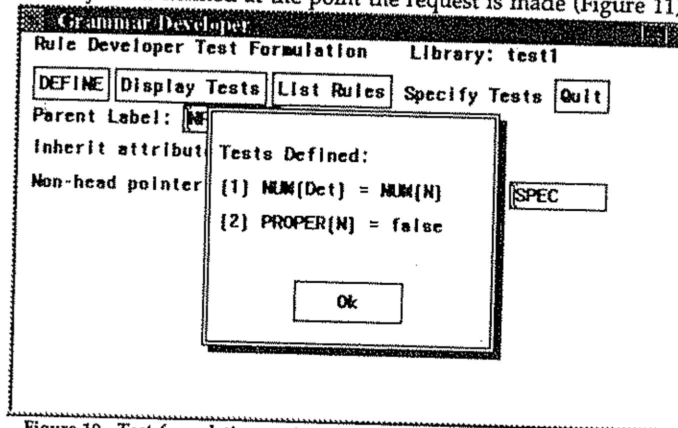

Figure 10. Test formulation environment, showing tests already defined

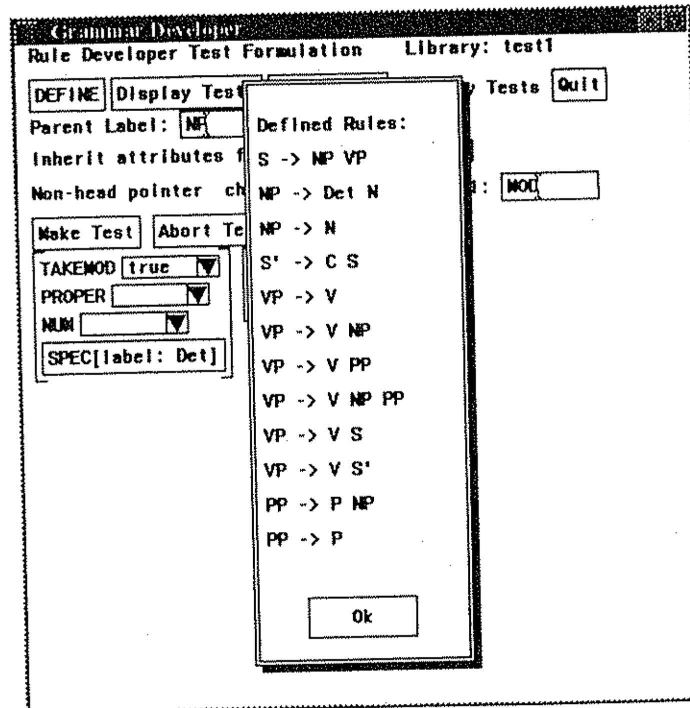

Figure 11. Test formulation environment, showing rules already defined

The Parser

TWINCLE0 incorporates a simple chart parser for augmented phrase-structure grammars [Kay86]. In the parsing environment, one must select a goal class, and enter a string. If the parser can analyze the string as an instance of the class, then the parse is displayed as a tree, with the nodes labeled by the labels of its constituents, as shown in Figure 12, where the goal is specified as *S*, and the string to be parsed is *Jon sleeps*. If one clicks on any node in this tree, then the attributes associated with that node are displayed in a superposed window, as shown in Figure 13. A more elaborate example is shown in Figure 14.

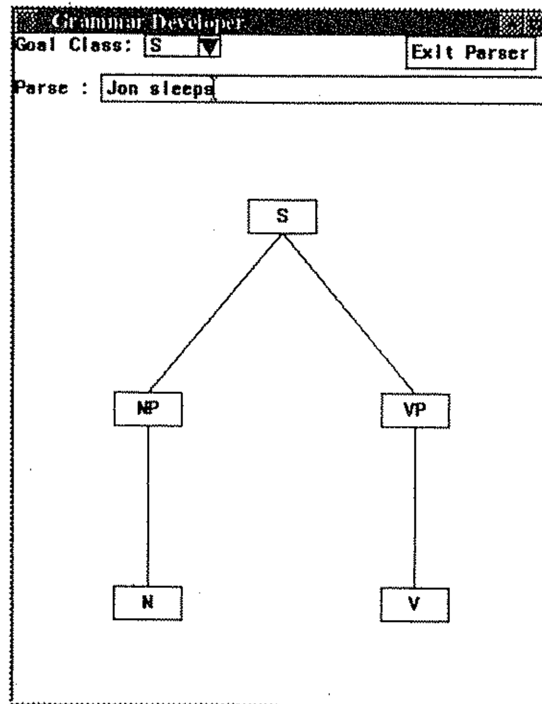

Figure 12. Parse of *Jon sleeps* as an instance of the class *S*

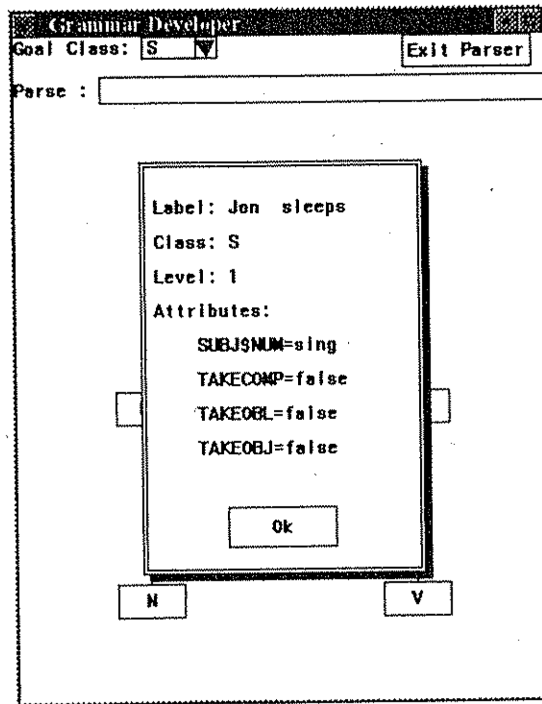

Figure 13. Display of attributes of root node of tree in Figure 12

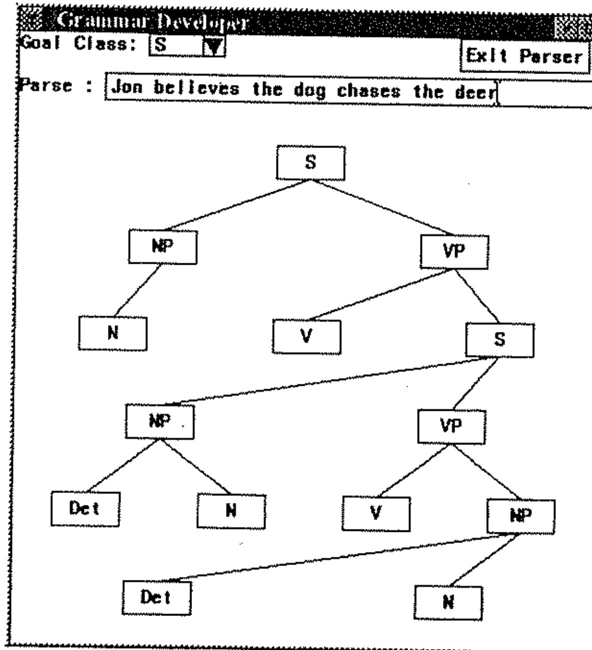

Figure 14. Illustration of a more elaborate parse tree

References

[Alle87] Allen, J. *Natural Language Understanding*. Menlo Park, CA: Benjamin/Cummings, 1987.

[Gris90] Griswold, R.E. and Griswold, M.T. *The Icon Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[Jeff92] Jeffery, C.L. *X-Icon: An Icon Window Interface*. Technical Report 91-1c, Department of Computer Science, University of Arizona, February 1992.

[Kay86] Kay, M. "Algorithm schemata and data structures in syntactic processing", in Grosz, B.J., Sparck-Jones, K. and Webber, B.N., eds., *Readings in Natural Language Processing*, pp. 35-70. Los Altos, CA: Morgan-Kaufman, 1986.