Report Number: WUCS-88-16

1988-02-01

Automatic Interface Generations From Grammar Specifications

Steve B. Cousins

This paper presents a method for automatically generating user interfaces to programs. All possible legal strings of input to a moderately interactive program, taken together, specify the input language of that program. A grammar for such a language is fundamentally knowledge about the language, and that knowledge can be used to assist the program's user in constructing legal program input. The set of words which can appear next in an input sentence, the 'Next set', is defined and a technique for calculating it with a modified version of Prologs's Definite Clause Grammar parser is given. One type of interface... **Read complete abstract on page 2.**

Automatic Interface Generations From Grammar Specifications

Steve B. Cousins

Complete Abstract:

This paper presents a method for automatically generating user interfaces to programs. All possible legal strings of input to a moderately interactive program, taken together, specify the input language of that program. A grammar for such a language is fundamentally knowledge about the language, and that knowledge can be used to assist the program's user in constructing legal program input. The set of words which can appear next in an input sentence, the 'Next set', is defined and a technique for calculating it with a modified version of Prologs's Definite Clause Grammar parser is given. One type of interface this method can generate is a menu-based front-end. The concept of menus is used very generally to include any method that allows a user to make a choice from among several options. The main difficulty with this technique is that menus may become very large (or may on occasion be infinite). This problem is overcome by the introduction of 'pre-terminals'-- classes of language terminals defined by predicates. When a preterminal is chosen from a menu, the user is prompted to type a value, which is then verified against the predicate associated with the preterminal.

AUTOMATIC INTERFACE GENERATION FROM GRAMMAR SPECIFICATIONS

Steve B. Cousins

WUCS-88-16

February 1988

Center for Intelligent Computer Systems
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Presented at the User-System Interface Conference, Austin, Texas.

This work was supported by the Center for Computer Aided Process Engineering and by the Center for Intelligent Computer Systems.

Abstract

This paper presents a method for automatically generating user interfaces to programs. All possible legal strings of input to a moderately interactive program, taken together, specify the input language of that program. A grammar for such a language is fundamentally knowledge about the language, and that knowledge can be used to assist the program's user in constructing legal program input.

The set of words which can appear next in an input sentence, the 'Next set', is defined and a technique for calculating it with a modified version of Prolog's Definite Clause Grammar parser is given. One type of interface this method can generate is a menu-based front-end. The concept of menus is used very generally to include any method that allows a user to make a choice from among several options.

The main difficulty with this techinque is that menus may become very large (or may on occasion be infinite). This problem is overcome by the introduction of 'pre-terminals' — classes of language terminals defined by predicates. When a preterminal is chosen from a menu, the user is prompted to type a value, which is then verified against the predicate associated with the preterminal.

1 Introduction

This paper presents a method of automatically generating user interfaces to programs. The key to the technique is the Next set,* the set of tokens that can come next at some point in the process of constructing a command to a program. Using the Next set, menu and command-line interfaces can be automatically generated.

Programs that are moderately interactive work best with this technique. The term 'moderately interactive' is used to distinguish the class of programs that are more interactive than batch programs, but are not highly interactive (such as Computer-Aided Design programs). For example, database management programs are moderately interactive, since the user normally gives them a line of input at a time, and then waits for a response.

A language for interfacing with a moderately interactive computer program may be simple like a command language or complex like a subset of a natural language. In either case, a grammar for this language captures in a concise way all of the sentences in the language, and is fundamentally *knowledge* about that language. Compiler writers are well aware of the value of a grammar in interpreting input to a computer. This same information about the language can be used to assist the user in creating valid sentences in the program's input language.

One advantage of automatically generating interfaces is that it relieves an application programmer of the need to generate them. In the paradigm of this work, the programmer specifies a grammar for the input language of a program, and an intelligent interface is generated for the program automatically. Existing programs can be 'retrofitted' with this type of interface by writing grammars for their input languages.

Another advantage of having a single mechanism generate user interfaces is that interface uniformity across programs is enforced. Each program automatically works the same way as other programs designed using the same technique, which reduces the need to retrain users. Apple Computer Company[1] has successfully persuaded software developers to adhere to a single interface standard on its Macintosh computer line, and has had great success in keeping that system easy to use.

A third important advantage of this work has to do with the reusability of program modules. Since the input to an application program is well-defined, it is much easier to use the application program as a back-end to other programs. Connectibility is very important because there is a large base of established software currently in use. Users of existing systems do not want to pay for or be involved in the redesign of systems that already work for them. However, these same users would like integrated systems which combine the functionality of the existing programs with modern user interface technology.

The method of developing user interfaces proposed here advocates that the user interface actually be a separate program which acts as a filter between the user and the application program. A user interacts with the user interface program, which in turn sends sentences reflecting the user's intentions to the application program. The

*In this paper, the convention of capitalizing the 'N' when referring to the set formally defined in this work is used to keep the distinction between other uses of the phrase, e.g., "the next set to be looked at".

prototype interface generators are written in Prolog, using Prolog's standard language parsing formalism, Definite Clause Grammars (DCGs)[2]. These Prolog prototypes construct sentences of a language, so that these sentences could be sent to application programs using a communication module. This work does not address the issue of inter-program communication directly.

A MENU INTERFACE EXAMPLE In this example, a menu interface to a robot control program is presented, in order to make concrete the ideas being presented. A menu can be considered as a set of items from which one item is to be selected. User interfaces containing menus have become popular because users of menu-driven programs have their options enumerated in the menus. This example will show how one such interface is generated from a description of a program's input language.

Consider a simple program for controlling a robot, which can take as legal input the commands:

```
Move the base <x> steps
Move the shoulder <x> steps
Move the elbow <x> steps
Move the wrist <x> steps
Open the hand
Close the hand
```

A menu interface generated by a system such as the one proposed in this paper from a grammar for this language would begin by displaying a menu to the user and waiting for a response. The initial menu would contain the items:

```
Move  Open  Close
```

For this example, assume the user wants to move the elbow motor 25 steps. The user would choose the `Move` option from the first menu and the system would now generate a second menu. Since the only word that can follow `Move` in this simple language is `the`, the system makes this choice for the user and proceeds to calculate what words can follow `Move the` in this language. The second menu the user will see will contain the items:

```
base  elbow  wrist shoulder
```

The user can now indicate his intention to move the elbow by choosing the `elbow` option from this menu. The system now determines that there is only one possible continuation of the prefix `Move the elbow: <x>`. For now, assume that the system 'knows' that angle brackets around a letter indicate that it is a variable. The system uses this knowledge to ask the user for a value to replace the `<x>` with. The user will type '25' in response to the system's query.

Finally, the system again calculates that only `steps` can follow a prefix of `Move the elbow 25` in the robot program's input language, so that word is appended to the input sentence being constructed, giving as the final request:

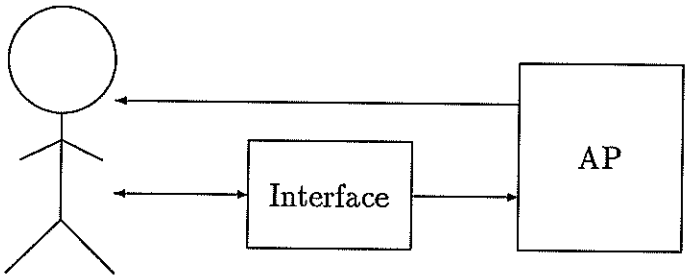

Figure 1: User—Application Program Interaction

```
Move the elbow 25 steps
```

In the course of constructing the query, the user saw two menus and had to enter one number. The only mistakes the user could have made were choosing the wrong menu item or typing the wrong number. With traditional interface technology, the user would have had to make 23 keystrokes. Any error made in the course of the keystrokes would have caused a traditional system to reject the user's command.

2 Interface Architecture

This section proposes a new way of programming user interfaces. First, the functional location of the user interface with respect to the rest of the application program is discussed. Then, a problem with generating sentences from grammars, called the preterminal problem, is introduced, and is shown to be a result of the system architecture. Traditionally, user interfaces were built into applications programs in the form of read and write statements. With the advent of software engineering and structured programming, the read and write statements were moved into separate modules, but the design and coding of the interface was still a part of the application program development. Recently, more and more libraries of interface utilities have become available, such as form-generation packages and the user-interface toolkits.[3,4,5,6]

In the work discussed here, the user interface is removed even farther from the application program. The input to the application program is specified fully by a grammar, and the interface is conceived of as something which assists the user in composing input to the application program. Figure 1 shows the relation between the user and the application program. In this model, the output from the application program is not interpreted, but is given to the user 'raw'. (Alternatives to this are possible, but are not a part of this work.) The system implemented in the course of this work, called the Visual Interface Utility (VIU) takes a description of the input language for an application program and produces an interface for that program. A grammar G for the input language of the application program is taken as knowledge about the input language and is used to generate the interface. VIU is the program that translates the grammar G into the interface. Through a series of user interactions, a proper sentence of L is constructed which reflects the intentions of the

user.

Two implementations of this architecture have been developed in the course of this work, one in LISP and the other in Prolog. In the LISP implementation, VIU was instantiated by a LISP function which interacted with the user through calls to a menu predicate, had its grammar passed as a parameter, and returned a list representing a sentence of L as its function value. In the Prolog implementation, the DCG parser (implicitly defined when the grammar was loaded) was called by a driver which interacted with the user and returned a sentence of L.

HANDLING PRETERMINALS In the simple example of the first section, the language contained the symbol <x> and the program was assumed to have the knowledge that that symbol denotes a variable. This symbol which represents a variable is actually a preterminal. We now consider preterminals in detail, and define the problem that they cause in generating sentences, called "the preterminal problem". A solution to the preterminal problem is derived here by defining a new grammar type. We begin with some definitions.

Definition 1 *A* preterminal *is a non-terminal which can only be replaced in a derivation by values which are constrained by a predicate, and which are neither terminals nor non-terminals.*

An example of a preterminal is the class of integers. While it is possible to write a grammar which recognizes and generates integers, doing so in the context of VIU is not desirable for a couple of reasons. First of all, users of VIU deal with logical atoms such as 25, not with sub-atomic parts such as 2 and 5. But more importantly, it may not always be possible or convenient to write a sub-grammar for the class a preterminal represents.

Definition 2 *The* preterminal problem *is that in many languages there are preterminal categories which are not easily or conveniently described with a traditional grammar.*

A high-level solution to this problem is to define a new type of grammar which includes preterminals.

Definition 3 *A* preterminal grammar (PTG) *G is a six-tuple:*

$$G \equiv < T, N, P, PP, S, R >$$

where T, N, and P are disjoint, $S \in N$, and R is a set of rules $N \rightarrow (T + N + P)^*$. PP is a set of preterminal predicates associated with preterminals:

$$PP : P \rightarrow \{f : \alpha \rightarrow 2\}$$

Preterminal grammars can produce two levels of languages, with and without preterminals.

Definition 4 *A pre-language PL(G) for a PTG G is*

$$PL(G) \equiv \{\alpha : S \Rightarrow^* \alpha \wedge \alpha = \alpha_1 \ldots \alpha_n \wedge (\forall i)(1 \leq i \leq n \rightarrow (\alpha_i \in T \vee \alpha_i \in P))\}$$

Definition 5 *A Language L(G) for a PTG G has all of the preterminals replaced by terminals satisfying their preterminal predicates:*

$$L(G) \equiv \{\alpha : \alpha \in PL(G) \wedge (\forall i : 1 \leq i \leq n : \alpha_i \in T \vee (\exists p : p \in P : PP(p)(\alpha_i)))\}$$

Thompson[7] discusses a similar problem in his dissertation on NLMenus. His problem, which he calls the 'value recognition' problem, refers to "recognizing database values when they appear in queries or commands." The preterminal problem is more general, because it deals with specifying values from classes, even when they have never been seen before. Thompson solves his problem in NLMenus with 'interaction experts.' The solution that will be proposed here is similar, but will use predicates which can interact, but may also perform simpler computations when appropriate.

The preterminal problem needs to be solved in order for VIU to be generally useful. At a high abstraction level, the solution is to recognize preterminals as a unique part of the grammar and have the implementation handle them in some way. For example, the implementation might display a special input window and ask the user for the value of the terminal at that point using the name of the preterminal as a prompt. One solution for the Prolog implementation is given in section 3.

3 The Next Set

This section defines the Next set, and describes one way of calculating it. The reader is assumed to have some knowledge of Prolog, or is advised to refer to a standard language reference (such as [8]). Simply stated, the Next set is the set of words that can come after a given prefix of a sentence. When considered this way, it is clear that such a set is useful to calculate in order to know what possibilities are legal at any given point in the construction of a sentence. Obtaining this knowledge is a very important step toward the goal of helping someone to construct a legal sentence, which must be the most important goal for an intelligent interface to a language-based system.

To build a menu interface to a language, it is necessary to know what words are permitted next at any point in a sentence. This set of 'next words' is called the Next set. A menu interface could simply display the Next set at each point in a parse and allow the user to choose one member of the set.

Definition 6 *A* Next set *N for grammar G and prefix* Π *is defined by*

$$Next(G, \Pi) \equiv \{\alpha \in (T + P) : \Pi\alpha\beta \in PL(G)\}$$

for some $\beta \in (T + P)^*$.

```
s --> dog_name, dog_action.
s --> boy_name, boy_action.
boy_name --> [john].
dog_name --> [rover].
boy_action --> [yells].
boy_action --> action.
dog_action --> [barks].
dog_action --> action.
action --> [runs].
action --> [hides].
```

Figure 2: A simple grammar

Initially, the Next set is the set of words which may occur as the first word of any sentence in PL(G), Next(G,ϵ). When one of these is chosen to be the first word of the sentence, the Next set for that one-word prefix is the set of second words in the set of sentences beginning with the chosen first word. In general, after a valid prefix of n words has been chosen, the set of all $(n+1)$st words that, when appended to the first n words forms a prefix of a valid sentence, is called the Next set for that prefix. The Next set can then be used, e.g., by placing it in a menu.

Consider the simple example in figure 2. This grammar describes a language containing 6 sentences:

john yells john runs john hides
rover barks rover runs rover hides

The initial Next set for this language (the Next set for the empty prefix) is {john, rover}. For one of these prefixes, 'john', the corresponding Next set is {yells, runs, hides}. This example is so simple because it has no preterminals.

Many different parsing algorithms can be modified to calculate the Next set. Tomita[9] points out that what is needed is a left-to-right on-line parser. In the course of this work, two parsing algorithms have been modified to calculate the Next set: Earley's algorithm and the Prolog implementation of Definite Clause Grammars. The Prolog implementation is discussed in the next section. A novel discovery made in the course of this work is the relative ease with which the Next set can be calculated in Prolog. The heart of the Prolog implementation of VIU lies in adding a single line to the normal Definite Clause Grammar translation mechanism. The rest of this section discusses the results of this work.

BASIC MODIFICATIONS TO DCGs The Definite Clause Grammar mechanism of Prolog is a translator from a grammar syntax to Prolog predicates. The parser that results after the translation uses Prolog's backtracking mechanism thus making it similar to an Augmented Transition Network parser[10]. The Next set is calculated in this mechanism by analyzing the cases in which the parser fails on the input and must backtrack.

DCGs handle terminals by translating them into calls to a special predicate called c ('connects'). The c predicate is ultimately called each time a word from the input sentence is considered. By carefully redefining the c operator it is possible to calculate the Next sets. The connects operator, c, is defined by the DCG-to-Prolog translator simply as:

```
c([W|S],W,S).
```

This definition reads: 'if the head of the first argument matches the second argument, succeed, and return the tail of the first argument as the third argument.' If the match fails, the predicate fails. In the context of parsing a sentence, the first argument is the input sentence and the first word of this sentence is being compared with a terminal symbol.

Based on Prolog's backtracking, we can redefine c to incrementally calculate the Next set. Normally, c either succeeds or fails based on whether or not the terminal symbol W is the head of the incoming list. This decision to succeed or to fail is based on the assumption that the entire input sentence is passed as the first argument. If the first argument is only a prefix of a sentence however, in parsing that string Prolog will at some point fail because legal words in the language are attempted to be matched against empty input. Whenever this happens, W is a word that should be a member of the Next set, because if the input were not empty, but contained W, there would be at least one place in the grammar that would accept W, namely, the current place. W should be added to the Next set, but c should fail the test so that other words in the set 'farther down the grammar' (since we are depending on Prolog's particular parsing mechanism for DCG's) can be found.

Assuming the predicate save(W) saves W by adding it to the Next set which is being incrementally calculated, the new definition of the c operator is as follows:

```
c([],W,[]) :- save(W),fail.
c([W|S],W,S).
```

Notice that the second line of the definition is just the original definition of c, and that the first line never succeeds. The new line of the definition is only active when its head matches, which is in exactly those cases described above.

The definition of save can be a simple assert, since facts asserted are not retracted during backtracking. Save is defined as:

```
save(W) :- assert( next(W) ).
```

If the relation next is empty before attempting to parse a sentence with a DCG (i.e. next(X) would fail), next will contain all of the words in the Next set when the parse has completed (and failed). A simple recursive program implements the interface described above. The predicate menu(L,W) is assumed to take a list of words to be in the menu L and return the word chosen, W. The predicate get_sent(Sent) returns a sentence in some grammar through Sent. We assume the grammar starts with the non-terminal s. Recall that in trying to prove s, the DCG mechanism will automatically make calls to c as terminal symbols are reached. The program in figure 3 implements get_sent.

```
get_sent(Sent) :- get_sent([],Sent).
get_sent(Prefix,Prefix) :-
        no_nexts,
        s(Prefix,[]).
get_sent(Prefix,Sent) :-
        setof(X,next(X),Menu),
        menu(Menu,Word),
        append(Prefix,[Word],NewPrefix),
        get_sent(NewPrefix,Sent).

no_nexts :- retract(next(X)),fail.
no_nexts.
```

Figure 3: Prolog program 'get_sent'

A PROLOG SOLUTION TO THE PRETERMINAL PROBLEM The preterminal problem occurs when the grammar refers to a class of terminals that cannot be conveniently be described in the grammar. DCGs and PTGs have been defined, and are combined to solve the preterminal problem in Prolog.

Definition 7 *A* Preterminal Definite Clause Grammar (PTDCG) *G is a seven-tuple*

$$G \equiv < T, N, P, PP, S, R, A >$$

where T, N, P, PP, S, and R are defined as in PTGs and A is a set of Prolog predicates (actions) as defined in DCGs.

Clearly, what PTGs contribute to PTDCGs are preterminals and preterminal predicates. DCG notation already has a distinction between non-terminals and terminals. To distinguish preterminals in Prolog, the names of all preterminals are stored in the relation `preterminal`. Preterminal predicates are implemented by associating a Prolog predicate with each preterminal. Before completing the explanation of the solution to the preterminal problem in Prolog, the mechanism that the DCG parser uses to solve the preterminal problem is examined.

A form of the preterminal problem occurs in Prolog's definite clause grammars even when they are not augmented to calculate the Next set. This is because there is no lexical analyzer generator working along with DCGs. Prolog has a solution for the problem, which is to allow DCGs to operate on lists of objects instead of lists of characters, thereby taking advantage of Prolog's built-in lexical analyzer which separates the input stream into objects. Statements like the following are a common occurrence in DCG grammars:

```
number --> [ X ], {number(X)}.
```

The above statement would be translated into Prolog, just as any other DCG rule would be, but number is effectively a preterminal because it matches a whole class of terminals: the set of objects which satisfy the number predicate. Note that the number predicate is just the preterminal predicate for the number preterminal.

Definite clause grammars provide a method of specifying preterminals, but the method causes a problem with the new definition of the c predicate used to calculate the Next set. The problem is that the variable X is inserted into the Next set. Since it does not make sense to have a variable in the set of next legal words, some solution to this problem must be found. The straightforward way to keep the variables out of the Next set is to put a guard on the rule which adds things to the Next set. The following rule replaces the first definition of c given above:

```
c([],W,[]) :- not( var(W) ), save(W), fail.
```

Unfortunately, this definition merely prevents the preterminals from adding anything at all to the Next set. The solution to this problem is to add the preterminals to the Next set manually, and the technique for doing this is the same as the technique that was used to add the additional rule to the definition of c. When the preterminal is 'called' with an empty input string, the preterminal should be added to the Next set. Taking the example of the number preterminal, the following rule is added to the original one:

```
number([],_) :- save( 'number*' ), fail.
```

The asterisk is appended to the name of the preterminal in this case, so that preterminals in the Next set are distinguishable from terminals. Finally, a third statement is added so that routines using the Next set will be able to distinguish terminals from preterminals. The final preterminal declaration for the number predicate is:

```
number --> [ X ], {number(X)}.
number([],_) :- save( 'number*' ), fail.
preterminal('number*').
```

Adding the lines similar to the ones above for each preterminal solves the preterminal problem in the Prolog implementation. If VIU gets beyond the prototype stage, some 'syntactic sugar' should probably be added to hide the implementation of this solution. The critical information about a preterminal is its name, its Next set name, and its associated predicate. A statement such as:

```
preterminal(number,'number*',number).
```

might be translated into the above three statements when the grammar is loaded into Prolog, just as DCGs are translated into Prolog representations.

4 Automatic Menu Generation

Once the Next set can be calculated, the most straightforward thing to do is to generate a menu interface, as has been indicated in previous examples. A generalized

menu predicate can be instantiated for a whole range of display hardware and menu systems. A general menu predicate is viewed as a module or subsystem that interacts with the user.

Given the set of next legal words, the task of the menu-generating subsystem is to collect the set and pass it on to a menu predicate. There are two problems in doing this. The lesser of the two problems is one of dealing with the error conditions of the menu predicate. For example, the user may request to abort from the entire program (which may or may not be a standard choice on every menu, depending on the menu system used) by signaling that in some way while he is faced with a menu. The more important problem is the pre-terminal problem, which was introduced in section 2.

The idea behind a general menu predicate is that the essence of a menu is a list of items and a choice among the members of that list. Whether a mouse is used to point at the item of choice, or whether the name of the chosen item must be completely typed in is a measure of menu convenience, not of 'menu-ness'. Therefore, the important features of the menu are encapsulated in the function

$$\texttt{menu} \ : \ \texttt{Next} \rightarrow (\iota x)(x \in \texttt{Next})$$

where (ιx) reads 'an x'. In the notation of Prolog, this is written as:

```
menu(+Set,-Choice)
```

The best menu system is one which the user already knows how to operate. In the prolog implementation of VIU on the Sun Workstations, the best menus are those used by the operating system, which are part of the SunView window system. The SunView window system has been integrated with Quintus Prolog through a product called ProWindows.[11] Using this system, a menu predicate has been defined in the manner of the general menu predicate above. This system will also allow experimentation with more general graphical interfaces, such as allowing integer pre-terminals to be entered with a slider instead of with the keyboard.

5 Conclusions

Automatic interface generation from grammars describing the input languages of programs provides a convenient way of separating the user interface of a program from its functionality. The application designer specifies a grammar for the input language of his program and a menu or command-line interface can be generated automatically.

The concept of a Next set, the set of words that can follow a prefix of a sentence in order that the new prefix is still a valid prefix in the language, was defined. One method of calculating the Next set using Prolog was presented, and the value of the Next set in automatically generating interfaces was shown.

Parsing programs use lexical analyzers or very specialized grammars to handle preterminals. The range of values a preterminal can take is not necessarily finite, however, and so the preterminal itself and not its values must be put into the Next set. This, in turn, requires special handling of Next set values.

Not every program is well-suited for a menu or command line interface. Database applications seem particularly well suited to these types of interfaces, while highly interactive or graphics-oriented programs do not. Work remains in the area of automatically generating more complex interfaces to programs.

References

[1] *Inside Macintosh.* Apple Computer Company, Cupertino, CA, 1985.

[2] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis–a survey of the formalism and a comparison with augmented transition networks. In Barbara J. Grosz, Karen Sparck Jones, and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, pages 101–124, Morgan Kaufmann Publishers, Inc., 1986.

[3] Luca Cardelli. *Building User Interfaces by Direct Manipulation.* Technical Report 22, Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301, October 1987.

[4] G. Pfaff, editor. *User Interface Management Systems.* Springer-Verlag, New York, 1985.

[5] Robin Faichney and David Barnes. The interconnection of highly interactive software modules. In *Tools of a Profession—10th International Conference on Software Engineering*, September 1987. Submitted to the conference.

[6] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[7] Craig Warren Thompson. *Using Menu-based Natural Language Understanding to Avoid Problems Associated with Traditional Natural Language Interfaces to Databases.* PhD thesis, The University of Texas at Austin, 1984.

[8] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, 1981.

[9] Masaru Tomita. *An Efficient Context-free Parsing Algorithm for Natural Languages and Its Applications.* Technical Report CMU-CS-85-134, Carnegie-Mellon University, May 1985.

[10] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 3(10):591–606, 1970.

[11] *ProWINDOWS 1.0β Reference Manual.* Quintus Computer Systems, Mountain View, CA, October 1987.