A Natural Language Query System for Hubble Space Telescope Proposal Selection

Thomas Hornick [1]

William Cohen[1]

and

Glenn Miller[1]

Astronomy Programs, Computer Sciences Corporation

Space Telescope Science Institute [2]

3700 San Martin Drive

Baltimore, MD 21218

Abstract

The proposal selection process for the Hubble Space Telescope is assisted by a robust and easy to use query program (TACOS). The system parses an "English subset" language sentence regardless of the order of the keyword phrases, allowing the user a greater flexibility than a standard command query language. Capabilities for macro and procedure definition are also integrated. The system was designed for flexibility in both use and maintenance. In addition, TACOS can be applied to any knowledge domain that can be expressed in terms of a single relation. The system was implemented mostly in Common LISP. The TACOS design is described in detail in this paper, with particular attention given to the implementation methods of sentence processing.

[1]Staff Member of the Space Telescope Science Institute

[2]Operated by the Association of Universities for Research in Astronomy for the National Aeronautics and Space Administration

1 Introduction

The Hubble Space Telescope (HST) will be launched aboard a space shuttle into low Earth orbit where it will function as a remotely controlled observatory for the next 15 years. Astronomers wishing to use HST submit observing proposals to the Space Telescope Science Institute (STScI). Due to the unprecedented capabilities of HST, a high oversubscription rate is anticipated; 1000-2000 proposals will be submitted each year, while only about 200-300 can be granted observing time. The evaluation of proposals will be accomplished via a peer-review process: An external scientific review committee (the Time Allocation Committee or TAC) advises the Director of the STScI in the selection of the HST observing program [1]. Although the scientific merit of a proposal is of prime importance, the selection process must take into account various limited resources such as unocculted viewing time, power, and communications links.

Due to the complexity and size of the proposal selection process, it was clear that software support was essential to assist in tracking proposal evaluation criteria. The Time Allocation Committee Operations Support (TACOS) system was developed to meet the following requirements:

- Flexibility: Although the basic procedures of the selection process are fixed, several detailed aspects are either uncertain or subject to change. Additionally, it was recognized that the first round of proposal selection would undoubtedly lead to several adjustment to the selection procedures. Therefore a prime consideration in the design of TACOS was to create a very flexible system in terms of the input data, query language and presentation of the output.
- Natural language interface: As the TAC members can devote little time for training in the use of the system, an easy to learn interface that allowed queries to be expressed in an English-like form was desirable. Tolerance to input errors and on-line help was also important. Available database query languages had complex and rigid syntax which require a significant amount of formal user training.

Several features of TACOS are worthy of note:

- Use of a bottom-up, shift reduce parser, rather than the more popular Augmented Transition Network (ATN). In the adopted approach, the grammar is maintained as data to the parser, whereas the grammar is embodied in the code of an ATN.
- Flexibility was not only incorporated into reporting, but also into the maintenance of the the system itself. Easy access to the initialization files allow the following to be modified if needed:
  - The TACOS database values and fields may be modified slightly or replaced completely, as long as the database has been generated in the proper input format. In fact, the database can be replaced by a new data and fields related to a completely different subject, which allows TACOS to be utilized in an unlimited number of domains.

- Changes to any of the field names or the security level of a field made be made within one initialization file.
- Keywords may be added, deleted, or renamed by making the proper changes to the lexicon initialization file.
- Grammar rules may be added, deleted, or modified by making the proper changes to the grammar initialization file.
- All keywords and phrases may be customized via macro definitions to accomodate the user's needs by either a initialization file or interactively by using the "define" phrase. The customization feature includes defining procedures, in which a series of commands may be executed with one predefined macro.

- The system was implemented in VAXLISP, a version of the popular Common LISP. Source code of the system may easily be ported to other hardware with little modification.

The first section of this paper provides an overview to the TACOS system. Sections 3-6 contain the details on the system design, featuring an in-depth look at the major components. The last section gives the conclusions of the paper.

2 Overview

The TACOS database consists of the all proposals being considered for selection. For each proposal, the database contains information relevant to selection such as Proposal ID, Principal Investigator information, total exposure time, dark time, etc. Each field can have three possible values:

- **original** - value input to TACOS; the original value is maintained to allow comparison to the limited value.
- **corrected** - In the event that an original value is in error, a TACOS user can fill the corrected slot.
- **limited** - The selection process may allow a proposal more or less of some quantity than was requested. This is kept in the limited slot.

The TACOS system will run interactively during the selection meetings (several sessions should be meeting simultaneously). Committee members have copies of each proposal and summary reports on criteria essential for selection, e.g. viewing time, number of targets, etc. During the deliberation process, the committee produces a ranked list of proposals and may also set limitations on resources used by a proposal. The TACOS system keeps track of these resources and produces a variety of repors from this information. A few examples will illustrate several features of TACOS.

```
define procedure "adjust limits" as
```

```
    define "scale" as
        "limited primary time / original primary time"

    define "resources" as
        "parallel time to scale*parallel time,
         parallel time used to scale*parallel time used,
         parallel time made to scale*parallel time made,
         time on target to scale*time on target,
         data volume to scale*data volume,
         number of uplinks to scale*number of uplinks,
         scheduling difficulty to scale*scheduling difficulty"

    limit resources

end procedure
```

Simply inputting the phrase "adjust limits" implements the procedure and sets the new limits.

To provide immediate feedback to the members of the meetings, requests can be made to display or print relevant data on the proposals. For example:

```
Tacos> display resources with primary time greater than 20
con't>
```

```
Tacos> output resources with primary time greater than 20 using "printer"
con't>
```

The ability to sort all proposals on field values is especially usefull when reviewers require a limit to be placed on a particular resource.

```
Tacos> display resources by largest primary time for all proposals
con't>
```

Due to the limited availability of HST resources, the committee may recommend the allocation of resources by a panel grade. If resources are allocated on this basis, committee would need to view all proposals in order of highest to lowest recommendation. They would also require a cumulative total of resource allocation to be shown according to the rank order, to allow them to determine when all resources have been allocated.

```
Tacos> display cumulative primary time by panel rank
con't>
```

TACOS is written mostly in VAX LISP, with some support routines written in VAX/VMS Digital Command Language (DCL), and editing and display capabilities provided using VAX/VMS Text Processing Utility (TPU).

3 Design

This section describes the high level design of the TACOS system. Figure 1 shows the architecture of the system.

The underlying structure of the TACOS database is rather complex, featuring a **objset**, a set of objects. Each **objset** contains a sorted list of the elements it contains and a hash table mapping the name of an element to the element itself. Each **element** is a hash table mapping field names to **fields**. These **fields** are actually property lists that maintain three values: original, corrected and limited. A special sort of the **objset** is maintained and is known as the **master set**.

Each sentence is processed in the following way. First, **read-input** reads the sentence from the terminal (or some other source.) **Read-input** reads until it reaches a blank line; it then concatenates the lines it has read into a string and hands this to **scan**. **Scan** then converts this string into a list of tokens: each token at this point is either a LISP atom, a number, a string, or a LISP list composed of numbers and/or strings. This *token list* is then checked for macros by **expand**. If any macros are present, then they are expanded in line. Finally, each token in the token list is classified by the **classify** module. **Classify** replaces each token with a simple parse tree.

Next, the expanded and classified token list is passed to the parser, where the list of trees is parsed into a parse tree. This tree is then in turn converted into a function which uses a restricted subset of LISP by the **codegen** module.

This function is then passed to the **backend** module. The first thing the **backend** does is to apply the function, via the **codeeval** module, to the **context** data structure, changing one or more fields of the **context** structure. The **context** structure is used for communication between frontend of TACOS and the backend. After various fields in the context structure have been set, the **backend** module examines the structure and uses the information in it to do what the user requested with his original command.

The **context** structure contains many fields; below is a list of the more important ones:

- The *verb indicator* which indicates what verb was used.
- The *consider set*, a set of proposals.
- The *change set*, a set of *change values*. A *change value* describes how some field of a proposal should be changed, as determined by the proposal selection committee.
- The *display set*, a set of values to display. Display values are structures that describe a value to display; they have these fields:
  - a *tag* to be used as a column heading;
  - a column *width*;
  - a *display function* which indicates what value to display, (This function is a LISP data structure which is actually a function of a proposal);
  - a *statistical function*, which indicates how to display the value - (e.g., "total" or "average").

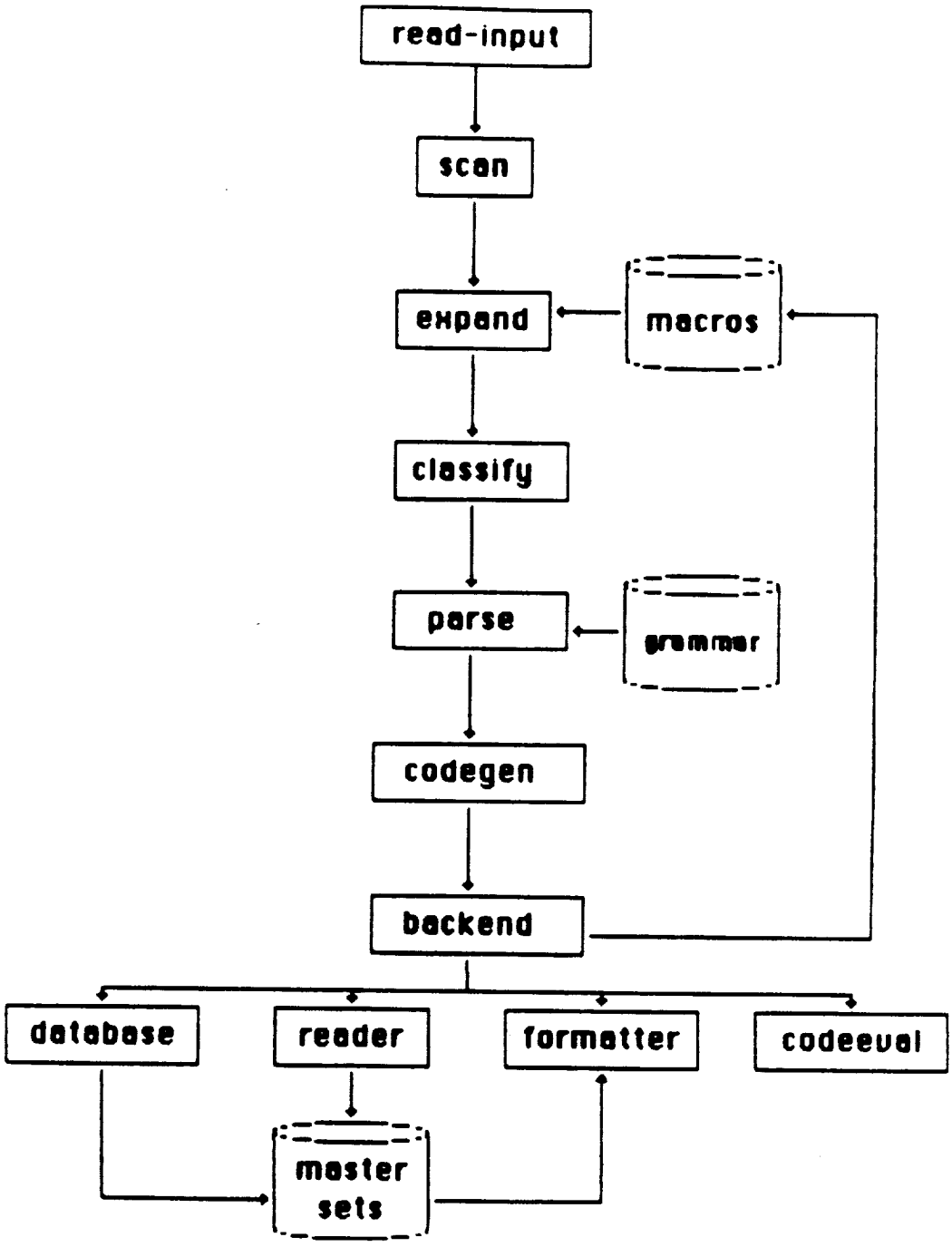

Figure 1: TACOS System

- The *order set*, which indicates how to sort the list of proposals. Each order value is a structure with these fields:
  - an *order function* which is just like the display function above, but is used differently; and
  - an *indicator* which indicates ascending or descending order, with default as descending.
- The *define-set*, which always contains a single pair of strings or is empty.

Other slots in the **context** structure are used internally, or for communicating specialized pieces of information to the **backend** module: for instance, there is a special slot for telling the **backend** module what procedure to call when the verb is **call-procedure**.

Most slots in the **context** structure are static; if **codeeval**ing a sentence doesn't change a slot, then the old value remains. TACOS will use these slot values as default values in the event that any of the set fields previously described are not specified in a single input sentence.

After the **readinput-...-codeeval** cycle is complete, most of the hard work is over. The set of proposals to operate on and other sets (such as sets of fields to display) have all been determined at this point. **Backend** now executes the sentence; this is nothing more than running the verb through a big **case** statement (actually a **cond** clause) and executing the appropriate procedure. Most of these procedures are, if not trivial, at least straightforward. **Backend** recognizes the following verbs:

- display - create a tabular display.
- edit - write the values to be modified into a table, invoke an editor so they can be changed, and then read the table back in.
- change, limit, correct - give a new set of values to some fields. The new values should be specified in the sentence, *e.g.*, "change requested primary time to 3 hours". **Change** changes the original value of a field; **limit** changes the limited value; and **correct** changes the corrected value of a field.
- define - define a macro.
- create partition - write a table representing a set of proposals (more generally, a set of objects) into a file.
- load partition - reads in a set of proposals from a given file. Appropriate error messages will be displayed if the set is disjointed from the current set of proposals, to prevent the database structure from being corrupted.
- define procedure, end procedure - these commands delimit a procedure definition.
- call procedure - call a predefined procedure
- help - call a help utility.
- hint - give a hint about the last error message.

- output, output using format - these are used to output scalar values (like "total dark-time"). The "using format" qualifier is meant to be used only in predefined procedures and is for formatted output.
- exit - quit the system.

4 Sentence Understanding - the TACOS Frontend

This section describes in detail the modules that comprise the TACOS frontend - *i.e.*, the part of TACOS dedicated to mapping the sentence typed in by the user into a LISP function that modifies the **context** structure. The cycle is as follows:

1. the **read-input** function reads a stream of characters from the terminal and produces a string;
2. the **scan** function turns that string into a list of tokens, terminated with a special end-of-sentence marker;
3. the **expand** function expands macros in that list;
4. the **classify** function maps these tokens to a list of parse trees;
5. the **parse** function parses this input, producing one parse tree for the non-terminal symbol "sentence";
6. the **codegen** function transforms this parse tree into a LISP function

4.1 Read-input

Read-input is responsible for reading a sentence from some input stream. Any leading blank lines are ignored, and a trailing blank line indicates the end of sentence input. The module operates in two modes, interactive and non-interactive. In interactive mode, a prompt is printed before each line is read: a primary prompt for the first line, and a secondary prompt for each line thereafter. In the example input sentence that was demonstrated in Section 2, the default prompts were shown, where "Tacos>" was the primary prompt, with "con't>" being displayed for each line of sentence thereafter.

A blank line was selected over the more commonly used period for the purpose of terminating input due mostly to our own experiences with natural language processors - it took quite a while for many people to get used to ending each sentence with a period.

4.2 Scan

The **scan** module performs lexical analysis of a string, returning a list of tokens. The tokens are either symbols, strings (corresponding to quoted strings in the input), numbers of some standard type, or lists of numbers and/or strings. **Scan** issues warnings about ill-formed

input (non-terminated quoted strings, non-terminated lists, illegal characters) but will never abort with an error.

Lexical analysis was done using a pseudo-readtable implemented with a hash table. This method was selected over a customized LISP readtable for two reasons; a pseudo-readtable has demonstrated a faster execution time and a customized LISP readtable is prone to changing output in generally unpredictable ways when modified.

All of the macro functions used by the **reader** are implemented at the user level. This was necessary for a variety of specific reasons, all of which are a manifestation of the problem that the standard LISP **read** function is *not* robust, and it was easy to draw **read** into a LISP error. As an example, if a control character is encountered when reading a symbol name, then LISP will signal an error. This is one of the reasons that we needed to map all alphabetic characters to a symbol-parsing routine.

4.3 Expand

Expand performs macro expansion on a list of tokens, in which any macros are immediately expanded in line. Implementation of this module is a different problem from parsing for one big reason: a bottom-up parser takes the shortest production it can, and the macro expander takes the longest. Cycles are avoided in this way: when defining X as Y, actually define X as *expand(Y)*. (Then when you perform the macro expansion by replacing X with *expand(Y)*, there is no need to also expand the replacement string, *expand(Y)*, since it has already been expanded. In other words, macros are always defined relative to previous macro definitions.)

The expand module uses a list of macro structures to do its work. Each macro contains two parts:

- The target. This is a list of tokens - when this list is encountered in the token list, then it can be superseded by the replacement.
- The replacement. This is another list of tokens, possibly empty.

Expand starts at the beginning of the token list, and begins matching targets to it, starting with the longest ones. When it finds a match, then the part of the sentence corresponding to the target is stripped off and superseded with the replacement, which is a parse tree, and **expand** continues expanding the remainder of the token list. If no match is found, then expand continues by trying to match macros that start at the second token, then the third, etc. and so on.

Expand works in time O(n)O(m) where n is the length of the sentence and m is the number of macros. There are more efficient methods to do this expansion, but they call for more complex ways of programming. The method implemented is more than sufficient for the macros we were concerned with expanding, since the problem is similar to regular expression matching.

4.4 Classify

Classify manipulates a list of tokens by mapping them to a list of simple parse trees. **Classify** issues a warning for each unknown symbol. If there are any unknown symbols in the sentence, then it aborts with an error message after the entire sentence is classified.

Normally, this classification process is carried out by a tokenizer, which also performs lexical analysis duties. The reasons for splitting **classify** off into a different module are:

- It makes the **scan** function more generally useful. It is used, for instance, in the table reader.
- It makes the **expand** function both generally useful (although not, as yet, generally *used*) and easier to write.

For implementation reasons, it's nice for the parser to deal with nothing except parse trees. Thus, **classify** outputs a list of "primitive" parse trees - trees without any subtrees. The "type" field of the generated tree indicates the part-of-speech/token-type of each token, and the "munger" field is a function that, when evaluated by **codegen**, will generate LISP code for the tree (see Section 4.6 for details.) These are obtained via lookup in the lexicon.

The lexicon is a hash table that maps symbols to the part-of-speech of that symbol, and parts-of-speech to the function needed by **codegen**.

4.5 Parse

Parse takes as input a list of "primitive parse trees" and outputs a single parse tree. The parser is data-driven, controlled by a set of grammar rules read in from a file.

The parser module is a bottom-up, shift-reduce parser; the end product of this method of parsing is a stack of parse trees. Each tree represents some recognized grammatical constituent (perhaps only a token.) The parse is deemed succesful if and only if the stack, after parsing is complete, contains a single parse tree representing a sentence constituent (sentence is thus hard-wired in as the distinguished symbol of the grammar.) If the parse is not successful, an error message is generated which is basically a dump of the parse stack; for example,

```
Error: I don't understand the sentence: if <comparison-list> then <sentence>
```

The heuristics for generating hints are described below.

The most usual way of parsing natural language is with an ATN parser - *i.e.*, top-down with backup. For several reasons, have departed from this conventional wisdom and used a bottom-up parser without backup.

First, there is a maintainability issue. ATN grammars are programs, not data, and as programs they are less readable than most. Also, the ATN representation for a grammar doesn't look anything like the BNF for it, which is what most competent grammar-designers would think in. Writing ATN grammars requires an intimate understanding of how ATNs

work, which is not a common piece of knowledge to have. In contrast, a shift-reduce parser runs off reasonably comprehensible data.

Second, there is an efficiency issue. An ATN interpreter is essentially a branch-and-bound search program, looking for possible parses. This can get expensive when you have certain types of conjunctions. It gives one some extra power and flexibility, but at a cost.

Finally, top-down parsing with backup gives, at least in our experience, no useful information when a parse fails about why it failed or how to fix the problem. In our opinion, this is a grievous shortcoming. The stack resulting from a bottom-up parse contains a great deal of information about the sentence and why it didn't parse. Although not all of the information provided is used, this fact did influence the decision on which technique to implement.

Bottom-up parsing is a poor technique to integrate with backup and search. To compensate for the lack of backup, two techniques are used. First, any grammar rule can look ahead an arbitrary distance into the input stream. Second, the code generated for a parse tree can depend on the context in which it appears. For instance, the parse tree for a field may generate the name of the field in one context, and a function to access the field in another context.

This brings us to another decision: the close integration of the parser with the codegen module. Every grammar rule contains an associated function that will produce LISP code that corresponds to the parse tree that was input for that rule. Attaching this semantic information to the grammar has obvious advantage; it eliminates the necessity for a file parallel to the grammar file that would be used by the code generator.

The following heuristics are used to provide hints after an unsuccessful parse:

- First, the parser looks at the first token in the token list. If this is a verb or the keyword that marks a major clause, then a summary of the syntax for that clause is given.
- If this fails, then a list of allowable clauses, with a brief description of each, is given.

A special free list is kept of parse tree structures, rather than relying on the LISP garbage collection routines, thus increasing the speed of the parser by a large factor. The code for the parser itself is perhaps a little opaque; the subroutine structure was carefully chosen for efficiency rather than for simplicity.

Some future enhancements to make the parser module more effective would include:

- Some consistency checks on the grammar - at least enough to catch spelling mistakes, etc.
- Although this parser handles short sentences effectively, it probably would be much less effective on a program-sized parsing problem. To increase performance might require some sort of indexing scheme; for instance, using a modified digital search tree at a single level might increase parsing speed dramatically.
- Eventually replace the parser module with a LR(k) (right parser works deterministically, if allowed to look k input symbols ahead at each step) parser-generator like YACC. These types of parsers operate in strictly linear speed.

4.6 Codegen

Codegen takes as input a single parse tree, and generates a LISP structure that corresponds to that tree.

The munger functions used by **codegen** are modeled after YACC actions. The same special variables **$1, $2,** ... are used to represent the code generated by the first, second, ... subtrees of the tree being generated. In addition, these variables have been added:

- ***1, *2,** ... represent the first, second, ... subtrees themselves.
- ***tree** is the tree being **codegen**ned.
- **$subtrees** is a list containing **$1, $2,** ...
- ***subtrees** is a list containing ***1, *2,** ...
- ***env** is the environment (see below).
- ***len** is the number of subtrees.

All of these are used somewhere in the TACOS grammar. The YACC pseudo-variable "**$$**" isn't needed - LISP conventions for returned values are used instead.

The code generation process may be best explained with an example. Consider this grammar rule:

```
(expr --> expr '+ addend
   YIELDING '(+ ,$1 ,$2))
```

When this production is used, a parse tree T is produced; attached to T will be a pointer to a function somewhat like this:

```
(lambda (*subtrees *env *tree)
   (let (($1 <code generated for first subtree>)
         ($2 ...))
      '(+ ,$1 ,$2)))
```

When (**codegen** T) is called, this function will be called with the appropriate parameters. To find values for **$1** and **$2**, as required by the **let** special form, **codegen** is essentially called recursively (actually, a lower-level routine is called.) The body of the **let** statement is then executed, producing LISP code to evaluate the **expr**.

Code generation is thus done top-down. This is necessary for the following important reason. Often, the meaning of a non-terminal symbol (e.g., the code that must be produced for it) depends on the context on which the symbol appears. To take an example from the TACOS grammar,

```
<set-list> and <set-list>
```

is normally interpreted as a union of two sets. However, in this sentence

```
all proposals in <set-list> and <set-list>
```

the conjunction must be interpreted as an *intersection*.

The parser usually has no way of knowing, when it parses

```
<set-list> and <set-list>
```

what context the conjunction will eventually appear in. Sensitivity to context is then left up to the **codegen** module. Codegen routines can determine what context they are used in by checking the environment for signals passed by some higher-level function.

The environment is implemented as an association list. A routine can pass a signal to the n-th subtree with the **send** macro; instead of using $\$n$, it uses the macro call **(send *n 'signal)**. The signal can be received by the called function with the macro **(receive)**, which always returns the most recent signal left for that function.

5 The Context Structure

The **context** structure is a special structure that contains the "content" of a sentence. The **context** structure can be thought of as a frame representing a sentence. The slots in the frame are populated by evaluating the function that is created by the **codegen** module.

Part of what the context structure contains are a set of default values for what set of objects to operate on, what functions to display, and so on. These values are permanent; they are not changed unless they are explicitly overwritten by a command. Permanent context values include:

- The *consider set*, the last of proposals (more generally, objects) specified.
- The *display set*, which describes the columns to display in a table.
- The *change set*, which describes a list of fields to modify.
- The *order set*, which describes a list of functions to order by.
- The *last repeatable verb* used in a sentence. A TACOS command need not contain a verb; it may instead specify a new set of proposals, for instance. In this case, the last repeatable verb is used to complete the ellipsis. Repeatable verbs are **display**, **edit**, and **output**.

Other slots of the context structure hold temporary values. The values are like permanent values, but the grammar requires that a command populate them before they are used. They are in context not to ensure that they are preserved from one command to another, but as a convenience; the **context** structure is the only mechanism for communication with the backend module.

These **communication** slots are:

- The *partition* slot, which holds the file name of a partition to **load** or **create**.
- The *called-procedure* slot, which holds the number of the procedure to **call**.
- The *define set*, a set of values to define, or to retrieve the definitions of. Currently, the grammar only allows you to put a single value in this set, but software does support processing a list of define values.

Another type of slot is the temporary slot. Temporary slots need not be populated by the grammar, but are not permanent because the last value specified is not likely to be re-used. Temporary slots are reset to a default value before the function produced by **codegen** is evaluated. Temporary slots include:

- The *verb indicator*, which defaults to the last repeatable verb.
- The *display command*, a VMS command to display a table. This defaults to a command that invokes the TACOS examiner, which allows a user to view fields they have specified in a full screen format.
- The *field selector*, which defaults to **nil** (indicating to prompt the user for a value.) This determines what selector value of fields will be changed in an edit command.
- The *format string*, which indicates the format of an output statement. Its default value is **nil**, which means to use a default output reporting format.

A final type of slot in the context structure is the internal slot. Internal slots are used mostly for procedure definition and conditional execution.

Each of these types of slot is, of course, treated differently by the software. Together, the slots completely describe a sentence, and provide a clean and well-organized interface between the TACOS frontend and the TACOS backend.

6 Command Execution - the TACOS Backend

This section describes the **backend** of the TACOS on-line system. The backend takes as input a function produced by codegen, and then uses that function to modify the context structure. It then looks up, from the verb indicator in the context structure, what low-level routine to use to execute the user's command. Finally, that routine is executed, and **backend** returns.

The value that is returned by backend is one of:

- **'t**, to indicate normal execution
- **'exit** to indicate that an **exit** verb was processed and that the session should terminate.

Backend does a considerable amount of consistency checking of the context structure. If a check fails, then **backend** throws an appropriate error message.

Backend uses a number of other modules to do its work. Some of these modules are quite complex in their own right. These modules are:

- **Codeeval**, which evaluates the code produced by codegen and thus makes the necessary changes in the context structure.
- The **database** module, which handles access of the TACOS database.
- The **reader** and **formatter**, which handle input of tables and output of tables, respectively.
- The **procedures** module, which handles storage and retrieval of predefined procedures.
- The **logging** module, which maintains a log of changes to the database.
- The **field_prot** module, which provides some security to the database in the sense that it restricts write access to parts of certain fields.
- The **system** module, which makes calls on the operating system.
- The TACOS examiner (a full screen viewing facility) and the TACOS editor (provides full screen editing capabilities).

7 Conclusion

The TACOS system has been throughly exercised in a series of mock TAC meetings and the response from the users has been very favorable. The power and flexibility of the queries was demonstrated during these trials. Another feature that drew favorable comments from the users was the ability to customize the dialect via macros and predefined definitions. Up to six simultaneous TACOS systems were run during the trials on a VAX 8600, along with other STScI users. Performance was acceptable. Increased speed for initializing the system was obtained by increasing the priority of the TACOS process, yet this did not significantly degrade performance for non-TACOS users. We are planning to move the initialization procedures into a suspended version of the TACOS system, which will insure even greater time savings.

We have described how the TACOS system supports the HST proposal selection process in several ways:

- A flexible, easy-to-use, English-like command language
- A reliable on-line help facility
- A flexible grammar with free formatting of sentence input
- A quick and helpful diagnosis of erroneous input

The design of TACOS includes several innovative technologies which are useful in the creation of a natural command language, including:

- Using a bottom-up, shift reduce parser for sentence understanding.
- Modular development of functions to allow maintenance of all grammar rules, macro definitions, database field security, and keyword phrases to be made with requiring recompilation.
- Implementation of the system in a highly portable language.

The result of our efforts is a easy-to use, extensible command language system that is applicable to many other problems.

References

[1] *SO-05 Proposal Solicitation and Review, Volume 1*, STScI 85051A, Final, August 1985

[2] Aho, A. V. and Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley Publishing Company, 1977

[3] Winograd, T., *Language is a Cognitive Process*, Addison-Wesley Publishing Company, 1983