A MACRO SYSTEM FOR ENGLISH-LIKE COMMANDS

by

Lawrence Alan Crowl

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

Timothy E. Lindquist, Chairman

John A. N. Lee

Donald W. McKeon

May, 1983
Blacksburg, Virginia

ACKNOWLEDGEMENTS

I would like to thank my chairman, Dr. Lindquist, for his valuable help. I would also like to thank Dr. Lee and Dr. McKeon for their service on my committee.

TABLE OF CONTENTS

Appendix

Chapter I

INTRODUCTION

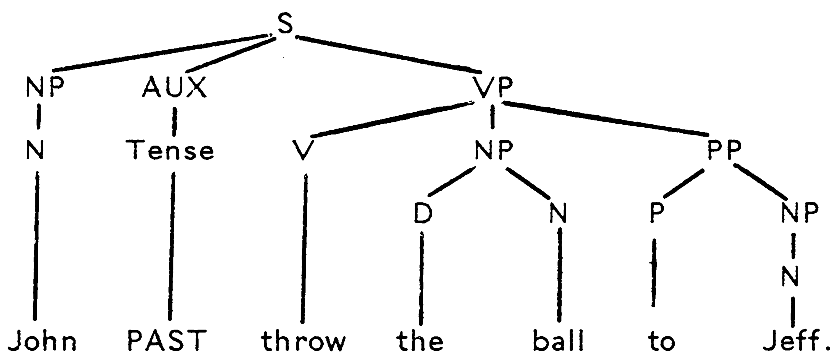
The current explosion in the use of computers has generated a need for user-friendly interfaces between humans and computers. These interfaces have a variety of requirements. For those interfaces which require selection from a wide variety of actions and which must reference items such as files, the general form of interaction is parametric, and it is commonly called the command language.

1.1 ENGLISH

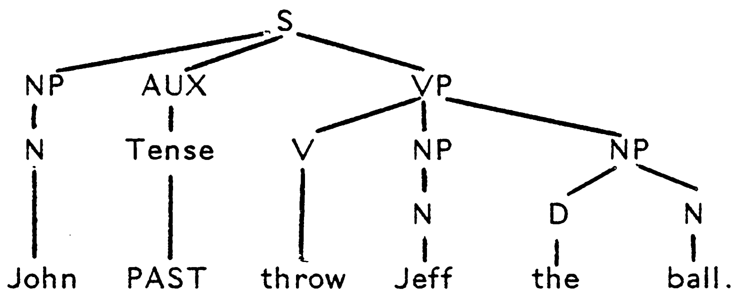
Command language designers are faced with the problem of making the language natural to the users of the language. A language is natural to its users when they do not conciously think of the syntax of the language when they create a message. As an example, people do not consciously think of the syntax of their native language when they speak. In linguistics, a person's subconcious knowledge of sentences in a language is called linguistic competence. Competence includes a knowledge of the semantics, syntax, and phonology of the language. A person's production and interpretation of actual sentences in a language is called the linguistic performance.

English is a native language to the users of many computer systems. However, as a natural language, English has some problems as a computer command system. A discussion of these problems requires the introduction of a model of linguistic competence. One such model is known as the Generative-Transformational model, as described in [AKMAA79]. (This is the standard version of Noam Chomsky in [CHOMN65]). The model is composed of three components: a syntactic component, a semantic component, and a phonological component, the latter two being interpretive. The syntactic component consists of a base subcomponent and a transformational subcomponent.

The base is composed of a set phrase structure rules for generating <u>deep structure</u> (inverted tree diagrams depicting the hierarchy of phrases), and a lexicon (a list of minimal units with meaning - morphemes - together with word formation rules). The constituents may be either another phrase, a lexical catagory (noun, verb, etc.), or a marker for such grammatical concepts as verb tense (a grammatical morhpeme). Deep structure phrase markers serve as input to the semantic component in determining the linguistic meaning of a sentence. An example deep structure phrase marker is the following:



The transformational part is composed of a number of partially ordered rules which map one phrase marker into another, from deep to surface structure. In theory, meaning is preserved through the transformations. For example, the Indirect Object Movement transformation would change the above phrase marker to the following derived phrase marker:



Another transformational rule is the Affix Attachment rule, which would take the derived phrase marker and map it into another with the terminal string as follows:

John threw Jeff the ball.

The resulting surface structure phrase markers serve as input to the phonological component to determine the phonetic form of sentences.

English has three forms of ambiguity, from [AKMAA79, page 232]:

1. lexical - the use of an individual word is ambiguous. For example, "He found a bat".
2. surface - the use of a group of words is ambiguous. For example, "He visited a little girls' school". This form of ambiguity arises from two different phrase structures which have the same terminal string of words.
3. underlying - a sentence has an underlying ambiguity if its ambiguity is not one of the two forms above. For example, "The chicken is ready to eat".

In each of the three cases above, the ambiguity may be resolved by referring to the context of the sentence.

Pronouns refer to a fully specified noun phrase in a very complex way, Involving linguistic as well as extra-linguistic constraints. For instance, in "Jim could not meet the Senator because he could not get into the Senate building" and "Jim could not meet the Senator because he was in a Senate committee meeting, the pronoun he is linguistically co-referential with both Jim and the Senator. However, based on certain non-linguistic knowledge, the pronoun "he" will likely be interpreted as refering to Jim in the first sentence and "the Senator" in the second.

I. D. Hill [HILLI72] writes, "ordinary English is really far too woolly and ambiguous" to specify concise instructions. This is a subjective statement, but he justifies this statement by identifying some problems English (or any native language) has with specifying instructions. According to Hill, English has no scoping rules, that is, there are no rules regarding how many of the preceding or following instructions are affected by the current instruction. For instance, in a set of four instructions where the last instruction is "repeat", the last instruction could refer to the first three instructions or the third instruction alone. Obviously, Hill is concerned with an economy of expression, since a referential phrase may be expanded to narrow the scope; e.g., repeat all of the above or repeat the last one. Implicit merging and forking of instruction sequences cause ambiguity, and require a high degree of intelligence to make the logical connections. The punctuation of English is ambiguous, especially in lists and functions. Using technical terms as nontechnical, and vice versa, can result in serious loss of meaning.

The basic problem is that many English statements can have several meanings, and the computer may choose one the user does not intend. In normal language usage, the context will mark one reading as preferred; i.e., an ambiguous sentence in isolation may be disambiguated by the various inter-sentential connections in the discourse. However, the context often becomes very large, and may include a knowledge of such things as the manner in which people live, the intent of the creator of the sentence, and an understanding of the roles people play in their interactions. The full meaning of an utterance depends on both linguistic knowledge as well as knowledge of the world, intentions of the speaker, etc. Moreover, the shared knowledge of speaker and listener (reader and writer) is crucial, and it may happen that the speaker makes a wrong assumption concerning what the listener knows. This means a sentence can be misunderstood, even though both sender and receiver believe the sentence is clear and unambiguous. Normal language use therefore entails more than the knowledge of the rules of sentence formation (competence). As such, a natural language like English cannot be easily used as a command language for current computer systems.

1.2 ENGLISH-LIKE

However, Ledgard et al [LEDGH80] show that user performance improves significantly when English-like commands are used for a text editor. Their experiments used a single interactive text editor which had two command interfaces, the first of which was English-like and the second was more notational. They state:

> The results demonstrate that redesigning the surface syntax of a command editor so that the commands more closely resemble English phrases resulted in far better performance. On all measures, performance using the English editor was superior to performance using the notational editor. This was true regardless of the experience of the users.

One possible reason may be that English-like commands allow the users to draw on their competence in English and apply it to the commands. This heavy dependence on linguistic knowledge should be successful because users spend less time concentrating on the syntax of the command and more time on the task to be performed. This use of competence also includes the transfer of meanings of ordinary words. When the users type the verb "replace", they expect something to be removed and something else put in its place.

Having established that users should perform better with English-like commands, a command language implementer's task is to define a set of phrases, or commands, that are close enough to English to be comfortable to the users, unambiguous in their syntax, and readily analyzable.

The usual approach for designing English-like commands is to form the commands such that they look exactly like English sentences. The command's string of words matches the corresponding English sentence's string of words. The words are organized in a linear, one level manner. Each word is related to the adjacent words only by its order, i.e. one before the other. For those systems, in the phrase "turn on the light", "turn" is related to "on" in the same way that "on" is related to "light".

As discussed in the previous section, English has a deeper structure in its sentences. It is composed of a nesting of phrases. In English, "turn" is more closely related to "on" than "on" is to "light" because turn on is a two word verb - a verb with a particle. Thus, "turn on" is a phrase and "the light" is a phrase, (as is turn on the light a larger (verb) phrase). A word in a phrase is more closely related to another word in the phrase than a word which is not in the phrase; that is, rules of semantic interpretation based on the phrase structure (in deep structure). English allows many of these phrases to be reordered in surface structure. Because the commands look so much like English, users may attempt to apply the same phrase interpretation to commands. When they reorder, or omit, the perceived phrases, they violate the simple word order of the command. The reordered command will not be understood even though it is a clear English sentence. Matching English word order without also attempting to match English phrase structure is misleading.

Many English-like command systems allow the use of noise words. Wherever noise words appear in a command, they are ignored. Noise words supply no information to the command processor, and are present only to make the user feel more comfortable. Determiners are often treated as noise words. In this case, "turn on the light" would mean exactly the same as "turn on a light" even though their English meanings are different.

There are, then, certain differences in the semantic interpretation of this type of English-like commands from that of corresponding sentences in the English language. This type of English-like command system has only a very shallow relationship to English. And as such, can only borrow very little of the competence of an English speaker.

1.3 PROJECT SCOPE

Many English-like command systems have been implemented. When such a system is implemented, the command analysis is a significant task. Each command system must implement the analyzer, which is another example of "reinventing the wheel". The quality of the interface is often limited because of the resources which must be spent on developing the analyzer instead of on improving the commands. The intent of this thesis is to develop a system which allows the designer of a command interface to quickly implement an English-like set of commands.

The approach of this thesis is to mimic the structure of English. The commands are composed of phrases. Words in a phrase may be manipulated together. This allows the users to borrow more heavily on their competence in English. Because the phrase structure of English is extremely complex, the phrase structure of commands will be much simpler. Only a few types of phrases will be included, and in less complex forms.

This thesis will also attempt to make every word in the command meaningful, no noise words will be allowed. Users are prevented from attaching meaning to meaningless words by removing meaningless words. This helps to prevent the loss of meaning as exemplified by "the light" and "a light".

David Beech [BEECD79] provides two definitions that are worth summarizing here: (The actual definitions are quite extensive and involved.)

1. A <u>computer language</u> is a form for communicating with computing systems that has a syntax and a semantics comprehensible to an appropriate recipient.

2. A <u>command language</u> is a computer language whose primary concern is to invoke system functions.

This thesis concentrates on one aspect of a command language, the syntax of invoking system functions, and will not address the semantics of the commands at all. For the remainder of this thesis, most terms will refer to the syntactic meanings of those terms. No semantic interpretation of those terms is to be applied.

Since the system aids in the definition of syntax and does not aid in the definition of semantics, the system does not aid in the definition of a command language. The command designer must address the semantic consistency and completeness of the set of commands defined. The command designer must also choose English words appropriate to the meaning of the command. The system will not aid in any such work. The semantics and primitive operations are provided by a base language. Good syntax will be useless without the proper semantics to back it up.

In addition, this system only concentrates on the flow of information from the user to the computer. It does not address such topics as error messages, help facilities, and command clarification. Concentration on single commands allows discussing a command without regard to the context of the command. However, two way communication requires a dialog, and therefore requires that a context be maintained and considered.

Chapter II

COMMAND SYNTAX

This chapter develops the syntax of the commands analyzed by the system. In developing this syntax both the command designers and command users are presented with a framework of the manner in which commands are structured, that is, the manner in which components of the command are related to each other. The user will generally extrapolate the framework to include an expectation of the effect of certain syntactic constructs such as modifiers. Because the syntax of commands is based on English, some syntactic constructs may be ambiguous. Therefore, this chapter provides some disambiguating rules for interpreting or rewriting these syntactic constructs.

2.1 REQUIREMENTS

There are many requirements on a language, and Schneider, Wexelblat and Jende [SCHNM79] have suggested some features that they feel a language should have. These features refer to both the syntactic and semantic parts of the language. This thesis considers only the syntactic part.

1. Users need <u>consistency</u> except where inconsistency prevents dangerous mistakes.

2. The language should be <u>natural</u> to both the user and the application.

3. The language should be <u>reasonable</u> in the syntax required of the user.

4. The language should be <u>complete</u>. Everything that needs to be done can be done.

5. The language should be <u>sound</u>. That is, all correct syntax should have valid semantics.

The general form of the commands should be flexible enough to allow the components of the command to appear in any reasonable order. Because commands may be entered interactively, user's English performance must be considered. English performance differs from competence in that it may contain corrections and repetitions of items. For example, "The neighbor sold his blue, no red car" is not what most people would consider a well formed sentence. However, it clearly means "The neighbor sold his red car". This is because the speaker is replacing the mispoken word "blue" with "red" in mid sentence. Because of this, any unnecessary requirements on the ordering of command components will only restrict and annoy the user. There are many variations of order, from completely unordered to completely ordered. Because many components have identical forms, the former would be difficult to analyze, and the latter would restrict the user too much. A medium that allows high variability, but still has enough order or structure to be readily analyzed must be found.

Because specifying all information needed in a command all of the time can become tedious at best, the format must allow default information for that which is left unspecified. The presence of default information implies that there must exist a method for overriding the default. When a default is overridden, the new information may apply only to a part of the remaining command. For example, suppose a command exists to print the files whose names are provided in a list, and that files are preceded by a header page by default. Altering the default (suppressing the header page) for a single file in the list instead of the entire list is an example of a limited override of a default. In order to specify the scope of the default and explicit information, command and parameter modifying capabilities must be present. These capabilites are usually represented in command substructures.

2.2 REFERENCING

Command languages are generally used when a computer user needs to reference items. A discussion of what and how items are referenced is in order. Akmajian et al [AKMAA79, page 235] present a useful view on reference:

> Linguistic-reference is simply the reference of the expression as a part of the language. Here it is customary to distinguish singular expressions from general expressions. Singular referring expressions include such proper names as Julius Caesar, The Eiffel Tower, and Paris; pronouns such as she, he, and they; phrases such as The present president of the United States, and the dents on the fender, which are used to refer to some definite single thing or single collection of things. On the other hand, general referring expressions such as verbs, adjectives, and common nouns, apply correctly to many individuals or combinations of individuals. Red applies to any red thing, table applies to any table, and so forth. The difference in referential function between the sentences below reflects a difference in the referential structure of the language;
> a. The first table had one leg. (singular expression)
> b. Each table has at least one leg. (general expression)

Computer are systems generally capable of dealing only with specific known objects, that is, they are capable of singular referring expressions only. This thesis will not address those experimental computing systems which may have the capability for general referring expressions.

2.2.1 Entities

An entity is something that exists physically or as data. Some entities might be files, numbers, terminals and printers, or collections of such things. Computers reference entities by an identifier. Each fully specified identifier references a unique entity. Most operating system command languages restrict the identifiers of entities to simple, word-like names. For simplicity, this thesis will also restrict itself to such names. This restriction allows a much simpler recognition of entity identifiers.

Since computers reference entities by name, the determiners, or articles, are not needed, so they will not be introduced into the command system. For example, the user would not tell a computer to "Run a program." The user would tell it to run a specific program, such as "Run the payroll program." But the payroll program has a name, say Payroll. The program is run by referring to it by its name - "Run Payroll". If the names of the entities are considered proper names, then this restriction follows naturally.

By removing determiners, the case where "the" and "a" are considered identical is prevented. This helps to prevent the user from expressing a thought in English that is radically different from its meaning as a command. This also prevents the command system from easily dealing with applications that require the use of general referring expressions.

Later in this thesis, command examples will occasionally be presented that use English common nouns. These words are meant to be names for a collection of objects known to the computer, and not as general reference.

Because many types of entities may be inconsistent with the intended use of a command, the command designer should be able to restrict the type of entity that is referenced at a point in a command. For example, a command to add numbers should not be given the name of a file.

2.2.2 Values

In this thesis, the term value means the syntactic unit consisting of the name of an entity together with its modifiers. For example, if "apple" is the name of an entity then "red apple" is a value. An entity name which is not followed by any modifiers is also a value. For instance, "chair" is both the name of an entity and a value.

2.2.3 Lists

Commands often contain lists of values as well as single values. A single value can contain only one entity name, but a list of values can contain many entity names. For example, the user compiles only one program (value), but the concatenation of files requires a list of files (values). Thus, command systems should be capable of accepting lists.

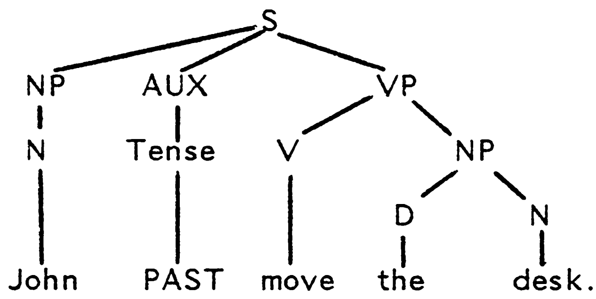
English provides lists of values with the form "pear, green apple, orange, and grape" or "wall, high ceiling, floor or door". The comma may or may not exist before the conjunction. The overriding feature is the separation of elements by commas (or semi-colons). The last element is preceded by a conjunction. If a separator is used to indicate the next value, then the conjunction can be removed. If no separator follows an element, then there are no more elements in the list. This method of constructing lists is very common, simple, and easy to understand, so the system uses it.

2.2.4 Objects

One important syntactic catagory is the direct object of the verb. In English, [AKMAA79, page 149]

> The object of a main verb can be structurally defined in English as the noun phrase that immediately follows the verb structure under the verb phrase structure.

For example, in the phrase structure:



the object is "the desk" because the noun phrase marker from which it descends immediately follows the verb marker under the verb phrase marker.

There are also objects of prepositions. These objects are the noun phrases which follow the preposition in a preposition phrase marker. (Certain of these objects - those in preposition phrases that are not also adverb phrases - will be analyzed as indirect objects of the verb; see section 2.3.2 below.)

In the command system an object is composed of a value or list of values. These objects contained in the definition of other parts of a command, in the same way that objects in English are contained in the definition of other phrases. In "move chair, desk, lamp", the list of values "chair, desk, lamp" is referred to by "move" and together constitute an object.

2.3 RESTRICTING ENGLISH

This section reduces the complexity of an English sentence to a form that is readily analyzable. The approach used is to take a commonly used construct in English and then reduce it to a simpler form. Those English constructs that are not mentioned in the following discussion, such as relative clauses, do not not have a corresponding construct in the commands.

2.3.1 Imperative Sentences

The model for the command syntax is imperative English sentences. Imperative sentences are used to command someone to perform an action and have an implicit second person subject (you). The simpler forms of these sentences have their verbs as the first word in the sentence. (Example: (You) Mow the lawn.) The simpler, verb first form of the imperative sentence is the form used to model commands. The computer is assumed to be the implicit subject, that is, the computer will perform the work. The restriction of commands to imperative sentences is not as unreasonable as it might first appear. People usually question other people to obtain data; however, people can command computers to supply data.

The verb at the front of an imperative sentence allows the recognition of the corresponding command before any processing or retention of data is necessary. The command verb is an example of a key. A <u>key</u> is a word used to recognize the presence of a command or certain portions of a command. The word must be known to the command processor before the commands are analyzed.

2.3.2 Direct and Indirect Objects

Imperative sentences often have a direct object and sometimes have an indirect object. The direct object is that which is acted upon or affected by the verb, and the indirect object is the benefactor of the action or recipient of the direct object. For example, in the sentence "Give the ball to the child", the direct object is "the ball" and the indirect object is "the child". Notice that the indirect object may be moved (by a transformation) to the position following the verb, without the preposition: "Give the child the ball." These English objects have a direct analog as objects in commands. The command objects follow and are associated with the command verb. The command objects are called parameters.

Often an object of a verb does not appear because it can be assumed or is not cognitively necessary. For example, in telling someone to "move", the object "yourself" can be inferred, so it need not be explicitly stated. This implies that the parameters may or may not occur, that is, they are optional. Optional parameters allow supplying information to override defaults or request additional actions.

Some objects must be present for the sentence to have meaning. For instance, in telling a computer to print, we must also tell it what to print. This implies that we must be able to require parameters in a command. If the parameter is required, it must appear in every instance of the command. Required parameters are used when the information they provide is necessary to execute the command.

In English sentences indirect objects are often located in preposition phrases following the direct object: "Give the ball to the child". The preposition "to" can be represented as a key preceding the parameter. Parameters which have a key followed by an object are called keyed parameters. Because the key uniquely specifies which parameter an object is, keyed parameters may appear in any order. This allows the most flexibility in the input order.

Although keys allow parameters to appear in any order, they should not be required for two reasons: first, the direct object of a verb often does not have a leading preposition, and second, if the parameters can be interpreted on a positional basis, the prepositions are unnecessary information. (Users do not like typing unnecessary information.) Parameters without keys must be interpreted on a positional basis. Because of this they are called positional parameters. Parameters will be interpreted based on the order in which they were defined.

A further restriction can be removed. The restriction that a parameter be either keyed or positional can be removed allowing parameters to be both keyed and positional, that is, the key is optional.

2.3.3 Modifiers

Both the action of the command and the entities referenced by the command will occasionally need to be modified. In English this is done by expanding phrases constituents as specified by the phrase structure rules; e.g. by expanding a verb phrase with adverbial phrases (e.g. a prepositional phrase or an adverb), or by expanding a noun phrase with adjectivial phrases (e.g. a prepositional phrase or an adjective). A prepositional phrase follows what it modifies and is composed of a preposition followed by an object. The straight forward way to model this is as a key followed by an object. The syntax is "(item modified) key object". The key serves to indicate that the modifying phrase is present in much the same way that the verb indicates that the command is present. The object then supplies the modifying information.

An adverb is a single word which precedes or follows the modified verb. Because verb first imperative sentences are being used, the adverb must follow the verb. The same model presented above for the prepositional phrase can be used for adverbs by defining a modifier which is composed of only a key and which does not contain an object. The key indicates the presence of the modifier, where the presence of the modifier is sufficient information for the modification to take place. Unfortunately, under this model all adverbs must be known and specified in advance. This is because the keys must be specified in the definition of the command.

In English sentences, adjectives precede the items they modify. In interpreting adjectives, each of the adjectives must be stored, and then applied to the item modified after it is encountered. This is a more expensive process to humans, as well as machines, than applying modifications to an item as they occur. In many native languages (Spanish and French, for example), the adjectives follow the items they modify. So it is at least as natural for an adjective to follow an item as it is to precede it, and post-nominal adjectives are easier to analyze. Because of this, adjectives will be specified after the element they modify. The adjectives can now be treated exactly as adverbs. This is the single largest deviation from the English language, but it allows a much simpler analysis of the command. (This idea is not without precedent in English. The military is fond of describing things in just this way. For example -- meal, individual, combat.)

Because modifiers may also contain an object, that object must also be modifiable. Additional modifiers can be used, which allows an unlimited nesting of modifiers. This construct is common in English in such phrases as "the dog under the table in the barn". The prepositional phrase "in the barn" modifies the object of the preposition "under".

Because a modified object is also an object, it too can be modified. This is the same situation that occurs in English with "big red apple". The modifiers are applied in series - "big" modifies "red apple" and not "red" alone. This capability is represented as an occurence of another modifier following the first modifier.

There is no reason why the key for a modifier cannot also be modified. This would allow phrases such as "boy running to home" to appear in the command. Here, "to home" modifies "running". This is also represented in the English phrase "deep red apple". Here, "deep" modifies "red" and not "apple".

2.3.4 Terminators

Because many of the parameters and modifiers may be optional, we might have trouble determining the end of a command. So, we require that a terminator, or end of command marker, be supplied with every command.

2.3.5 Summary

The command syntax may be summarized by considering only five types of syntactic constructions: commands, parameters, value lists, values, and modifiers. Commands are composed of a key, followed by any number of modifiers, followed by any number of parameters. A parameter is composed of a single value or a value list, possibly preceded by a key. A value list is composed of a sequence of values separated by a character, such as a comma. A value is composed of the name of an entity followed by any number of modifiers. A modifier is composed of a key, followed by any number of modifiers, possibly followed by a value or a value list. (The formal grammar for this command syntax is given in appendix A section 1.1.)

2.4 PROBLEMS AND SOLUTIONS

Some problems exist with this grammar as thus far described. When the choice of a solution is unclear from a human standpoint, a decision which lends itself to simpler implementation is made.

2.4.1 Keys and Names

Both keys and names are instances of words. Because they both have the same syntax the keys will exclude their words from being used for names of files, queues and other entities. Most keys will be prepositions or functional words which are not often used in the naming of objects. Therefore, the system will not check for conflicts. The user is responsible for insuring that names do not confilct with keys. Prepositions and other words likely to be used as parameter or modifier keys should not be used as names of entities.

2.4.2 Multiple Uses of the Same Key

Using the same key in separate parts of a command could cause ambiguity in the command. This thesis will not attempt to address this problem so command designers should avoid multiple uses of the same key.

2.4.3 Nested Lists

Nested lists are ambiguous if the same separator and value types are used in the lists. For example:

```
print A, B with C, D, E
```

Is B printed with C, or C and D, or C and D and E? The solution chosen is that a following list element will be included in the innermost list in which it can be included. This allows the decision to include the next element in a list to be based on a minimum amount of data. So, in this example, B is printed with C, D, and E. This problem can be avoided if different list element separators are used for nested lists, such as commas and semicolons. This implies that there must be a mechanism for defining the separator. Such a capability is provided, although it does not insure that different separators are used when nested lists can occur.

Value lists cannot be modified. So in the example above, A will not printed with C, D, or E.

2.4.4 Parameters with Optional Keys

Parameters with optional keys may come out of order which would at best mean a delay in the interpretation of the parameter. For instance, a command defined to have the format:

```
copy from X to Y
```

with the from and to keys optional, given the call:

```
copy A from B
```

can not have A evaluated, or acted upon, without first encountering the `from` keyword because A could also have been a `from` parameter. In order to allow execution of a command as it is typed, the command processor must be able to determine which portion of a command a word is without waiting for later words. This requires a single left to right analysis with no look ahead. To allow this immediate interpretation of parameters, an unkeyed parameter will be matched with the first possible parameter in the definition. That is, parameters will be bound, or interpreted, at the earliest possible time. Under this interpretation, the second call will be invalid because the `from` parameter occurs twice and the to parameter does not occur at all.

In this example, if either of the keys were required, the command could be easily analyzed. If the to key were required, then if a parameter does not have a key, it must be the `from` parameter. This would allow the command:

```
copy A to B
```

This rule does not allow a command with the two forms

```
give ball to child
```

and

```
give child ball
```

which is the English indirect object movement transformation discussed in the first chapter. The choice between immediate execution of a command, and more freedom in placing the parameters will require psycological testing. Such testing is beyond the scope of this thesis, so the solution presenting a simpler implementation was chosen.

2.4.5 Modifier Scope

Modifiers associated with the verb mean that the interpretation of the entire command is modified. Modifiers associated with a value or another modifier mean that only that element is to be modified. Lists can not be modified as a whole, although individual elements of a list may be modified.

2.4.6 Multiple Instances of a Modifier

Two instances of the same modifier may be modifying the same element in a given instance of a command. This gives rise to two, or more, sets of values for a given modifier. For example,

```
apple red red
ship at sea at dock
Washington via Baltimore via Philadelphia, Boston
```

each have two instances of a modifier modifying the same element. In order to avoid the inevitable confusion that may arise because of different interpretations made by the command designers, the following interpretations are defined:

The repetition of a modifier containing no object will mean the same as a single occurence of the modifier. The first example would be interpreted as

```
apple red
```

The repetition of a modifier containing an object with a single value only, will cause the modifier to be revalued. Only one value of the modifier will exist, and it will be the last value. The effect will be as though the first instance did not exist. In the second example above, a ship can only be in one place at a time, so the interpretation would be

```
ship at dock
```

The repetition of a modifier containing an object with a value list, will cause the values to be added in addition to the values in the first appearance of the modifier. The effect is as though there were one occurrence of the modifier with the value list containing all values mentioned in the multiple modifier instances. Since travel to Washington may be through Baltimore, Philadelphia, and Boston, the interpretation of the third example is

```
Washington via Baltimore, Philadelphia, Boston
```

The command designers will know which interpretation to use because they choose whether the modifier may have no value, a single value, or a value list. The command users should be able to tell which form a modifier uses by consulting the documentation for a command.

Chapter III

COMMAND DEFINITION

This chapter describes the development of the definition of commands with the syntax presented in the previous chapter. The definition will require as little specification of the command as possible while not constraining the flexibility of the command syntax. That is, the definition will not have to account explicitly for all of the possible element orders that may appear in a command. In addition, the command processor will be capable of working with a variety of language systems.

This chapter discusses two approaches to defining English-like commands, the procedure call approach and the macro approach. The chapter then describes the approach actually used, which is a modified form of the macro approach.

3.1 PROCEDURE CALL APPROACH

Most of the previous general systems for English-like commands are based on invoking procedures. These systems are oriented towards use in a programming language and not towards interactive use. Each procedure has a single sample sentence which is used to call the procedure. So each different form of a command must have a different procedure to perform the different action. Default actions would be implemented as one procedure with the default explicitly mentioned and another with no default. For example, one procedure would be invoked by the command "print (file)" and a different procedure would be invoked by the command "print (file) on (printer)". The appropriate procedure is called based upon the form of the command used.

These methods can represent the word order of English sentences well, but cannot represent the structure of sentences at all. This is because each command is treated as a linear sequence of items, not as a structured collection of items. In these procedure call phrases, each item has the same level of association with all other items. However, in English, the word "them" in "give them the book" is more closely associated with "give" than with "the", and "the" is more closely associated with "book" than with "them".

3.1.1 SNAP (A Programming Language)

The use of English sentences, with defined semantics, is the base of a programming language called SNAP [BARNM69]. The language consists of a number of imperative sentences which act as instructions. The instructions have some flexibility, although the flexibility is fairly limited. Users may define instructions by defining a procedure to perform the action. The user defines the sample sentence, and indicates which parts of the sentence are the arguments. Only the arguments may vary from the sample sentence in an actual instruction.

The user can define adjectives using a similar English-like definition of a function. However, the user defined functions can only be used in system defined commands and can not be used in the call to a user defined procedure. This prevents a useful compounding of phrases.

As an example of SNAP, the definitions:

```
PROCEDURE TO SEND A BILL TO A PERSON AT AN ADDRESS FOR
  AN AMOUNT FOR THE FIRST TIME.  THE ARGUMENTS ARE A
  PERSON, AN ADDRESS, AND AN AMOUNT.    ...    RETURN.
  END OF PROCEDURE.
PROCEDURE TO SEND A BILL TO A PERSON AT AN ADDRESS FOR
  AN AMOUNT FOR THE SECOND TIME.  THE ARGUMENTS ARE A
  PERSON, AN ADDRESS, AND AN AMOUNT.    ...    RETURN.
  END OF PROCEDURE.
PROCEDURE TO SEND A CHECK TO A PERSON AT AN ADDRESS
  FOR AN AMOUNT.  THE ARGUMENTS ARE A PERSON, AN
  ADDRESS, AND AN AMOUNT.    ...    RETURN.  END OF
  PROCEDURE.
PROCEDURE TO EXTRACT THE SURNAME OF A PERSON.  THE
  ARGUMENT IS A PERSON.    ...    CALL THE LAST NAME THE
  SURNAME. RETURN. END OF PROCEDURE.
```

would allow instructions of the form:

```
SEND A BILL TO "JOHN SMITH" AT "12 WILLOW ROAD" FOR
  "$12" FOR THE FIRST TIME.
SEND A BILL TO "JANE DOE" AT "46 MAPLE STREET" FOR
  "$37" FOR THE SECOND TIME.
SEND A CHECK TO "SAM JONES" AT "4 EAST ELM ROAD" FOR
  "$42".
PRINT THE SURNAME OF "SALLY WILLIAMS".
```

but would not allow instructions of the form:

```
SEND A CHECK TO THE SURNAME OF "JILL MILLER" AT
  "123 MOUNTAIN ROAD" FOR "$16".
SEND "AL CARTER" A CHECK FOR "$31" AT "15 OAK ROAD".
```

This system has some other features that impact on its use as an English-like command system. First, it can only manipulate named lists, it can not manipulate a literal list. Second, it allows determiners although they appear to be ignored. Third, the system also allows multiple word names.

3.1.2 Groves' System

Lindsay Groves [GROVL82] has proposed a system for using simple English sentences to call procedures in a base language. Each procedure call is described in terms of a template starting with a word and followed by delimiters (words or punctuation marks) and typed parameters. (The types of parameters depend on the base language and may include expressions.) Each template must be distinct in at least one delimiter or parameter. Different sentence forms are provided by supplying a different template which often implies a different procedure.

For example, the definitions:

```
PROCEDURE merge <table: IN stringtable> into <afile: IN OUT
  stringfile>; BEGIN ... END;

PROCEDURE merge <table: IN stringtable> with <file1: IN
  stringfile> giving <file2: OUT stringfile>; BEGIN ... END;
```

would allow the procedure calls:

```
merge word_table into old_word_file ;

merge word_table with old_word_file giving new_word_file ;
```

3.2 MACRO APPROACH

The ability to deal with a variety of systems is best satisfied by defining the commands with some sort of macro. If the command grammar discussed above is taken to be the calling format of a macro which produces code in a standard programming language, several advantages become apparent:

1. Only the macro command processor and one language processor need be used, versus the two or more language processors needed on most current computer systems. Currently most systems have a system command language for file manipulation and program control in addition to a different programming language for normal applications development. For example, the VAX/VMS Digital Command Language and FORTRAN.
2. Commands may be used inside the text of applications programs. This could allow system level commands to be invoked from inside programs with the same syntax as they are invoked interactively. Most current systems have a different interface for requesting system services from inside a program.
3. Only one complete language needs to be learned, although the specification of commands using that language will need to be learned.

However, there are some disadvantages:

1. There is an overhead cost of expanding the macros. Standard commands are interpreted, whereas macro commands must be expanded and then interpreted.
2. Because the code generated by the macro must not depend on the ordering of the command's parts, the expanded code is not totally efficient.

As computing power becomes less expensive, these disadvantages will become less important.

3.3 PREVIOUS MACROS

In order to gain an idea of the features needed in an implementation we examine some previous macro systems. The types of macros examined are assembler macros, text macros, and syntax macros.

3.3.1 Assembler Macros

The first type of macro we shall consider is the assembler macro. An assembler macro is composed of a name that is used to reference it and a list of parameters. These parameters are bound to arguments in a strict one to one, positional manner. The argument values then replace the parameter symbols in the body of the macro. For example, the definition

```
macro  addition, left, right, result
load   left
add    right
store  result
mend
```

given the call

```
addition  alpha, beta+4, gamma-3
```

would generate

```
load   alpha
add    beta+4
store  gamma-3
```

This method of expansion is extremely inflexible. Two improvements on the flexibility have been made by most advanced macro assemblers. They are keyed parameters and default values for omitted parameters. The keyed parameters are preceded by a key which allows them to be specified out of order, provided the key is present. The defaults specify values for parameters which are omitted from the call.

Still more advanced macro assemblers allow conditional and repetitive expansion. The programmer may test which parameters are present and adjust the expansion of the macro to fit the call.

The two most important problems in matching these assembler macros to the grammar discussed in the previous chapter are the very inflexible nature of the calling format and the lack of any modification substructures.

3.3.1.1 XPOP

XPOP [HALPM64] may well be the most flexible assembler macro processor introduced to date. It has the capability of defining the separators, line continuation, and terminator for a call. It can define noise words that are to be ignored in the interpretation of the call. Parameters may be keyed, which allows the parameters to appear in an order other than the one in which they were defined. The command verb may be positioned anywhere among the parameters, provided of course that none of the parameters may be interpreted as a command verb. XPOP depends on a line orientation to know when one command begins and another ends.

The result is that an arbitrary number of English sentences can be written that will have the desired effect. The drawback is that English sentences can be written that will not have the expected effect. People interpret words and structures that only appear to be used in the calling format. XPOP provides only superficial similarities to English. It interprets a linear sequence of words only, and not a structure of phrases. Like the more primitive macro assemblers, all information is contained in a linear sequence of parameters.

For example (adapted from [HALPM64]) the macro

```
STORE  MACRO   $INTO$A,B,C
       CAL     B
       ORA     C
       SLW     A
       END
       CHPUNC
       CHPUNC  3S1 2 '2' 1T2'.
       CHPUNC  1S2.. 1T1.
       NOISE   4INTO 4CELL 3THE 6LOGICA 6FORMED 2BY 6OR'ING
       NOISE   6BOOLEA 6VARIAB 3AND 3SUM 2OF 2AS
```

Would allow calls like:

```
STORE INTO CELL 'GAMMA' THE LOGICAL SUM FORMED BY 'OR'ING...
       THE BOOLEAN VARIABLES 'ALPHA' AND 'BETA'.
STORE AS GAMMA THE LOGICAL SUM OF ALPHA AND BETA.
INTO GAMMA STORE THE SUM OF ALPHA AND BETA.
SUM ALPHA AND BETA AND STORE INTO GAMMA.
```

which are correctly understood, and

```
SUM GAMMA AND ALPHA STORE BETA.
STORE ALPHA INTO GAMMA OR BETA.
```

which mean the same in XPOP, but mean something else in English.

3.3.2 Text Macros

Text macros expand the concept of the assembler macro into high level languages. These macro processors have been steadily improving in their flexibility and ease of use over the years.

3.3.2.1 ML/I

Among the first of these full scale macro processors was ML/I [BROWP67]. The syntax description only supplies the delimeters, or separators, between parameters, and does not describe the parameters themselves. The description of lists in ML/I requires the use of a reference to another part of the syntax description. This is analagous to the "goto" program control structure. In addition, it is difficult to control the expansion in anything other than a straight, nearly literal, expansion.

For example, from [BROWP67], an IF statement with the forms:

```
IF arg1 = arg2 THEN arg3 END
IF arg1 = arg2 THEN arg3 ELSE arg4 END
```

has the definition:

```
MCDEF IF = THEN OPT ELSE END OR END ALL AS
<     LAC :A1.  /Load first argument.
      SAD :A2.  /Skip if it differs from the second argument.
      SKP      /Equal case: skip one instruction.
      JMP XX:T2. /Unequal case: jump over inserted code.
      :A3.     /Inserted code.
MCGO L1 UNLESS :D3. = ELSE; /Macro-time jump if else clause
                            /is absent.
      JMP YY:T2. /Jump over else clause.
XX:T2., :A4.   /Insert else clause.
YY:T2., MCGO L2; /Macro-time jump to end of replacement text.
:L1.XX:T2., :L2. >; /Placing of macro-time labels.
```

given the calls:

```
IF A = B THEN JMS SUB
END
IF PIG = DOG THEN LAC C
                DAC D
ELSE LAC Y
    DAC Z
END
```

would generate the code:

```
     LAC A   /First call.
     SAD B
     SKP
     JMP XX1
     JMS SUB
XX1, LAC PIG   /Second call.
     SAD DOG
     SKP
     JMP XX2
     LAC C
     DAC D
     JMP YY2
XX2, LAC Y
     DAC Z
YY2,
```

3.3.2.2 PM

Masataka Sassa has introduced a very powerful macro processor, which he has called PM [SASSM78]. The macro processor provides a regular expression form for the calling syntax, which is very easy to use. The recognition of a macro is done by "balanced" pattern matching. That is, parenthetical items, such as "()", "[]", and "begin end", are balanced. The expansion is controlled by an Algol-like procedure that references parts of the regular expression definition in the calling syntax. This processor is very well suited to use with high level languages.

For example, the definition (taken from [SASSM78]):

```
# INITIALIZE GLOBAL MACRO VARIABLES #
.MACRO SCOPE BEGIN ;
   .MACRO '.MACINTLZ' =
   .BEGIN
         .GLOBAL INT SNUM ;   SNUM := 9000 ;
         .GLOBAL INT VNUM ;   VNUM := 0 ;
   .END ;
   .MACINTLZ
.MACRO SCOPE END ;

# MACRO DEFINITION OF FOR STATEMENT #
.MACRO '.FOR' ID '.FROM' F ('.BY' B|/) '.TO' T
       '.DO' BODY:NONEG '.OD' =
.BEGIN
<%ID=%F ;
%SNUM IF (%ID.GT.%T) GO TO %(SNUM+1)
%BODY
%ID=%ID+>;
      .IF B /= '' .THEN <(%B)> .ELSE <1> .FI ;
<%/GO TO %SNUM
%(SNUM+1) CONTINUE%/> ;
      SNUM := SNUM + 2
.END ;
```

with the call

```
.FOR I .FROM 0 .TO N-1
   .DO
      S := S + A(I)
   .OD
```

with some other formatting macros yields

```
      I = 0
9000  IF ( I .GT. N-1 ) GO TO 9001
      S = S + A(1)
      I = I +1
      GO TO 9000
9001  CONTINUE
```

3.3.3 Syntax Macros

Syntax macros provide one important feature not provided in the earlier macros, that is type checking. The arguments to the macro will be checked to insure that they are of the appropriate type. This can be a very valuable feature in insuring that the command operates on correct objects before execution begins.

3.3.3.1 Leavenworth's System

Leavenworth [LEAVB66] describes a system where macros are defined in two parts, a macro call syntax (structure) and the resulting text (definition). The structure is defined with a limited form of regular expressions. This syntax definition has a serious drawback: it does not allow lists of objects. The text is expanded literally with some sections expanded based on the presence or absence of portions of the macro call.

In the definition, items enclosed by brackets [ ] are optional, and items enclosed by braces { } and separated by a bar | are alternatives. A clause is a section of the call that may or may not occur. They are numbered from left to right at the occurence of a marking symbol on their right. The marking symbols are: |, } and]. Output text enclosed by {n and } is output if clause n is present. Output text enclosed by {¬n and } is output if clause n is not present. When $n appears in the output text, it is replaced by the value of the nth variable in the call specification.

For example, the definition (from [LEAVB66]):

```
SMACRO FOR variable := express
{ WHILE express | [ BY express ] TO express }
DO statement

DEFINE BEGIN L1 : $1 := $2;
            L2 : IF {1 $3 THEN BEGIN $6; GO TO L1}
                 {3 $1 <= $5 THEN BEGIN $6;
                 $1 := $1 + {2 $4}{¬2 1}; GO TO L2}
                 END
      END
ENDMACRO
```

given the call

```
FOR I := J WHILE A < B DO A := A + I ;
```

would generate

```
BEGIN L1: I := J;
     L2: IF A < B THEN BEGIN A := A + I; GO TO L1
         END
END
```

3.3.3.2 MCOBOL

Triance and Yow [TRIAJ80] describe a system designed to generate macros for the COBOL language. The programmer describes the calling syntax in terms of COBOL record definitions and interprets the actual call with a COBOL program. MCOBOL was built with the intention of enlarging the set of statements a programmer may use. The system requires an extremely large amount of detail and has little chance of being successfully applied outside of the COBOL language.

Their example of an inline PERFORM statement has the definition:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. IN-LINE-PERFORM-MACRO
            CLASS IS "STATEMENT".
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. CDC-7600.
 OBJECT-COMPUTER. CDC-7600.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT GEN-TEXT-FILE ASSIGN GTFILE.
 DATA DIVISION.
 FILE SECTION.
 FD  GET-TEXT-FILE LABEL RECORDS OMITTED.
 01      GEN-TEXT-RECORD.
   03   GT-TESTN-CODE         PIC X(2).
   03   GT-INDIC-AREA         PIC X.
   03   GT-ACTUAL-TEXT.
     05 FILLER               PIC X(22) VALUE "PERFORM RE
-                                     "PEATED-CODE"
     05 GT-COUNTER            USAGE IS ARGUMENT.
     05 FILLER               PIC X(49) VALUE   "TIMES
-                                     "GO TO END-TEXT.
-                                  "REPEATED-CODE.".
     05 GT-STMNT              USAGE IS ARGUMENT.
     05 FILLER               PIC X(28) VALUE     ".
-                                 "END-TEXT.".
 MACRO-LINKAGE SECTION.
 01       IN-LINE-PERFORM
   03  PERFORM-VERB            PIC X(8)
                             VALUE "PERFORM ".
   03  INT                    USAGE IS ARGUMENT
                             CLASS IS "INTEGER".
   03  TIMES-WORD              PIC X(6)
                             VALUE "TIMES ".
   03  STMNT                   USAGE IS ARGUMENT
                             CLASS IS "STATEMENT".
   03  END-PERF                PIC X(12)
                             VALUE "END-PERFORM ".
 PROCEDURE DIVISION USING IN-LINE-PERFORM.
 MACRO-START.
     OPEN OUTPUT GEN-TEXT-FILE
     MOVE "CP" TO GT-DESTN-CODE
     MOVE SPACE TO GT-INDIC-AREA
     MOVE INT TO GT-COUNTER
     MOVE STMNT TO GT-STMNT
     WRITE GEN-TEXT-RECORD
     CLOSE GEN-TEXT-FILE.
END-MACRO.
     EXIT PROGRAM.
```

given the call:

```
    PERFORM 7 TIMES
    WRITE TEST-LINE AFTER ADVANCING 1
    END-PERFORM
```

would generate the code:

```
    PERFORM REPEATED-CODE 7 TIMES
    GO TO END-TEXT.
REPEATED-CODE.
    WRITE TEXT-LINE AFTER ADVANCING 1.
END-TEXT.
```

3.3.4 Problems and Solutions

Commands refer to named entities so a mechanism must exist to transfer that name from the call to the expanded code. The solution is the use of substitution variables where the name referenced in the call is assigned to a variable, and then when the variable appears in the expanded text, the name is substituted for the variable.

All of the more capable systems discussed above have the same problem - they are tools which are too general for the task of quickly defining commands. Each of the systems discussed above relies on a description of the syntax of macro calls. Thus descriptions must include all possible variations of the call. For instance, if a command has three parameters which may appear in any order, then the syntax description must explicitly specify that each of the six possible orderings are acceptable. The solution is to define commands not in terms of their syntax, but in terms of their components and structure. The flexibility of the calling syntax is derived from matching the actual call to the structure of the command.

Another problem is that the expansion is either too literal or too procedural. There is either not enough flexibility or the expansion must be programmed. Ideally, the expansion should be structural in nature. That is, it should be a byproduct of the recognition of the command. Because commands are interpreted based on their structure, the expansion of commands should also be based on their structure. This can be done by attaching to each component of the command, a section of code to be expanded. Some code may need to be expanded before the code specified by subelements as well as after the code specified by subelements. For example, procedure header code needs to be expanded before, and procedure trailer code after, statements expanded by subelements. Therefore, there should be code that is expanded on recognition of the element and on completion of the element.

In assembler and text macros, the types of the values were not checked for consistency with the intended use. Syntax macros do such checking and this concept is important enough to be carried over into the proposed system. This checking would help to prevent the case where the command processor accepted a command which later failed to execute because of typing errors in the base language.

3.4 DEFINITION LANGUAGE

The command definition is structural in nature. Elements of the command are described in terms of their heirarchical relationship with each other, not in terms of their syntax. The elements of the command are defined along with sections of code to be expanded at various points during the recognition of the element. The objective is to have each element generate portions of a complete procedure in the base language during the expansion. Since base language procedures may have header and trailer information that must be specified regardless of the order in which subelements appear, the definition must have the ability to attach base language code to both the start and end of elements' definitions.

Each definition has two types of information, command recognition information and code for expansion. These types of information are distinguished by placing them on separate types of lines within the definition of a command. Control lines begin with a flag character, and code lines do not. In this thesis, the flag character is always the % character, although it may be changed. Control lines indicate the structure of the command and give the information needed to recognize the elements of the command. Code lines specify, in the base language, the action to be taken when the corresponding element is recognized.

Code lines are treated literally with the exception of the replacement of substitution variables. Substitution variables are preceded with the flag character, in both control and text lines, and are active from the point of definition to the end of the macro.

Each element definition must specify the information necessary to recognize that element, and its subelements. In addition to recognition information, each element may have information indicating whether it is required or optional.

With the above constraints in mind, the language for defining commands was made as simple to implement as possible. While the commands themselves are English-like, little attempt was made to make the definition of commands English-like. Each element definition is composed of a header line describing the element, subelement definitions, and a trailer line indicating the end of the element definition. Lower case letters are treated as though they were upper case. (The formal grammar for the definition of commands is in appendix A section 1.2.)

3.4.1 Commands

A command is composed of seven parts specified in the following order:

1. the command header control line,
2. the code to be expanded when the command is recognized (optional),
3. the command modifier definitions (optional),

4. the code to be expanded after all of the modifiers have been expanded and before any of the parameters are expanded (optional),
5. the parameter definitions (optional),
6. code to be expanded after all of the parameters have been expanded (optional), and
7. the command trailer control line which contains the key CEND.

The command header control line contains the following items in order:

1. the key COMMAND,
2. the key KEY, and
3. a key that will be used to recognize the command.

For example,

```
% COMMAND KEY alpha
code
(modifier definitions)
more code
(parameter definitions)
last code
% CEND
```

3.4.2 Parameters

A parameter definition is composed of five parts specified in the following order:

1. the parameter header control line,
2. the code to be expanded when the parameter is recognized (optional),

3. one or more alternative value definitions,
4. the code to be expanded after all of the values have been expanded (optional), and
5. the parameter trailer control line which contains the key PEND.

The parameter header control line contains the following items in order:

1. the key PARAMETER,
2. one of the keys REQUIRED or OPTIONAL indicating whether the parameter is required or not,
3. a comma
4. a description of the nature of the parameter which is either
   a) the key VALUE indicating a single valued parameter or
   b) a list specification composed of
      i) the key LIST
      ii) the key BY and
      iii) a separator character in single quotes, and
5. optionally a parameter key specification which is composed of
   a) a comma,
   b) the key KEY,
   c) a key to be used in the recognition of the parameter,
   d) and one of the keys REQUIRED or OPTIONAL indicating whether the parameter key is required or not.

For example,

```
% PARAMETER REQUIRED, LIST BY ';', KEY beta OPTIONAL
code
(value definitions)
last code
% PEND
```

3.4.3 Values

A value definition is composed of five parts specified in the following order:

1. the value header control line,
2. the code to be expanded when the value is recognized (optional),
3. the value modifier definitions (optional),
4. the code to be expanded after all of the modifiers have been expanded (optional), and
5. the value trailer control line which contains the key VEND.

The value header control line contains the following items in order:

1. the key VALUE,
2. and a value specification which is composed of either
   a) a key specification which is composed of
      i) the key KEY, and
      ii) the literal key which is the value to be matched, or
   b) a variable specification which contains
      i) one of the types FILENAME, QUEUENAME, or NUMBER, and

ii) a variable which is to be equated to the actual value in the parameter. This variable is active from the point of definition through the end of the command call.

For example,

```
% VALUE NUMBER %gamma
code using variable %gamma
(modifier definitions)
last code also using %gamma
% VEND
```

3.4.4 Modifiers

A modifier definition is composed of seven parts specified in the following order:

1. the modifier header control line,
2. the code to be expanded when the modifier is recognized (optional),
3. the submodifier definitions (optional),
4. the code to be expanded after all of the submodifiers have been expanded (optional),
5. the value definitions (optional),
6. code to be expanded after all of the values are expanded (optional), and
7. the modifier trailer control line which contains the key MEND.

The modifier header control line contains the following items in order:

1. the key MODIFIER,
2. one of the keys REQUIRED or OPTIONAL indicating whether the modifier is required or not,
3. a comma
4. the key KEY,
5. a key to be used in the recognition of the modifier, and
6. a description of the nature of the modifier which is either
   a) null indicating that the modifier has no associated value,
   b) a single value indication consisting of
      i) a comma, and
      ii) the key VALUE, or
   c) a list specification composed of
      i) a comma,
      ii) the key LIST
      iii) the key BY and
      iv) a separator character in single quotes.

For example,

```
% MODIFIER OPTIONAL, KEY delta, VALUE
code
(modifier definitions)
more code
(value definitions)
last code
% MEND
```

Chapter IV

USING THE SYSTEM

This chapter describes a methodology for converting a set of meaningful actions, with an associated set of English sentences, into a set of commands. These principles are supported by an extended example.

4.1 METHODOLOGY

As discussed earlier, these commands will model imperative English sentences with the computer as the implied subject and the verb as the first word of the command.

In addition to the verb, a sentence may have a direct object and an indirect object. The objects should be modeled as parameters to the command. The keys for the parameters should be set to a meaningful preposition. Generally, the command will not be meaningful without some of its objects specified, so the corresponding parameters should be required.

The verb may also be modified by adverbs or adverbial phrases. The adverbs can be modeled as verb modifiers that have no associated value. They can also be modeled as parameters with no key, and with the adverbs as the parameter's values. The adverbial phrases can also be modeled in two ways. First, they may be modeled as a modifier to the verb with an associated value. The modifier will therefore appear before any parameters, which allows contextual information to be given before the objects of the verb are interpreted. Second, adverbial phrases may be modeled as a parameter with a required key. The "adverbial" parameter may not appear until the very end of the command, so it should not contain information necessary to interpret other parameters.

Frequently the entities referenced in a command must be qualified. In English this is done with adjectives, adjective phrases, and clauses. Clauses have not been introduced into commands. Adjectives and adjective phrases should be modeled as modifiers attached to the values to be qualified. Adjectives will be modifiers without values, and prepositional phrases can be modeled as modifiers with the preposition as the key and the value providing the modifying object. Modifiers, and their values, may also be qualified by modifiers. This allows the programmer of the command to define an arbitrary depth of modification.

4.2 EXAMPLE

Suppose someone wishes to move items on a table. The items can be moved from their established position to another position on the table or to the side of the table. Each item has a color of either red or blue.

4.2.1 Sample Sentences

Because imperative sentences are used, some of the sentences which may be used to command the movement are:

```
Move the blue pyramid to the side.
Move the red cylinder at (1,2) to (3,6).
Move the blue cube at (3,2).
```

4.2.2 Sample Commands

Using the methods described earlier, the sentences might be modified as follows. First, remove the determiners they are redundant information in this context. Second, place the adjectives after the items they modify. These relatively simple transformations yield the following commands.

```
move pyramid blue to side
move cylinder red at (1,2) to (3,6)
move cube blue at (3,2)
```

4.2.3 Defining the Command

The command structure could be defined as follows: (The % is the flag character.) First, define the move command,

```
% command key MOVE
% cend
```

But this only defines the verb. The item being moved must be added. Since the item is the direct object of the sentence, the item is expressed as a parameter. Because the command is not meaningful without an item to move, the parameter should be required. Since there is no preposition that identifies the direct object, there should be no key. For simplicity, assume only one item may be moved at a time. This means that the parameter has a single value.

```
% command key MOVE
% parameter required, value
% pend
% cend
```

However, each parameter must contain a value definition which will be the actual parameter. The items being moved are solids so the arbitrary type called SOLID will be used as the value type.

```
% command key MOVE
% parameter required, value
%   value SOLID %OBJECT
%   vend
% pend
% cend
```

Each item also had colors that might need to be specified. Since these are adjectives, they are modeled as modifiers with no associated values.

```
% command key MOVE
% parameter required, value
%   value SOLID %OBJECT
%     modifier optional, key BLUE
%     mend
%     modifier optional, key RED
%     mend
%   vend
% pend
% cend
```

Each item could also be specified by its position. This specification was done by an adjective phrase so it is expressed as a modifier with an associated value.

```
% command key MOVE
% parameter required, value
%   value SOLID %OBJECT
%     modifier optional, key BLUE
%     mend
%     modifier optional, key RED
%     mend
%     modifier optional, key AT, value
%       value POSITION %PLACE
%       vend
%     mend
%   vend
% pend
% cend
```

Each item being moved has a destination. If the intent is to move an item off another so as to move the other, then the destination is unimportant and need not be specified. But the destination is important most of the time, so the user must be able to specify the destination. So the destination should be optional.

The destination is an adverb phrase, and not an indirect object. We can model the phrase as either a verb modifier or as a parameter. Because the destination information is not needed in determining the start position, the destination phrase may occur after the direct object. This is achieved by modeling the adverb phrase as a parameter. The preposition does not really aid in the analysis of the command, so its corresponding key can also be optional.

Since an item can only be moved to one place, the parameter is single-valued.

```
% command key MOVE
% parameter required, value
%   value SOLID %OBJECT
%     modifier optional, key BLUE
%     mend
%     modifier optional, key RED
%     mend
%     modifier optional, key AT, value
%       value POSITION %PLACE
%       vend
%     mend
%   vend
% pend
% parameter optional, value, key TO optional
% pend
% cend
```

The value for this parameter is more complex because we have two cases to consider. The object is being moved to the side, or to a specific position. The word "side" can be used as a key value to specify movement to the side. The position value can be used to give a specific position.

```
% command key MOVE
% parameter required, value
%   value SOLID %OBJECT
%     modifier optional, key BLUE
%     mend
%     modifier optional, key RED
%     mend
%     modifier optional, key AT, value
%       value POSITION %START
%       vend
%     mend
%   vend
% pend
% parameter optional, value, key TO optional
%   value key SIDE
%   vend
%   value POSITION %STOP
%   vend
% pend
% cend
```

In cases where keyword values might be confused with other types of values, the keyword values should come first. This way the keyword will not be treated as an instance of another type of value.

4.2.4 Expansion Code

The final step is to specify the code that is to be expanded as the command is recognized. Default information must be set up before the elements of the command are recognized. The elements modify that information if they are present. And, finally, the command executes code to perform the necessary action.

```
% command key MOVE
destination = unknown
% parameter required, value
%   value SOLID %OBJECT
item = %OBJECT
color = unknown
source = unknown
%     modifier optional, key BLUE
color = blue
%     mend
%     modifier optional, key RED
color = red
%     mend
%     modifier optional, key AT, value
%       value POSITION %START
source = %START
%       vend
%     mend
%   vend
% pend
% parameter optional, value, key TO optional
%   value key SIDE
destination = side
%   vend
%   value POSITION %STOP
destination = %STOP
%   vend
% pend
moveit ( item, color, source, destination )
% cend
```

Given the command definition above, some sample commands and the code they generate are:

```
move sphere

destination = unknown
item = sphere
color = unknown
source = unknown
moveit ( item, color, source, destination )
```

```
move pyramid blue to side

destination = unknown
item = pyramid
color = unknown
source = unknown
color = blue
destination = side
moveit ( item, color, source, destination )

move cylinder at (1,2) to (3,6)

destination = unknown
item = cylinder
color = unknown
source = unknown
source = (1,2)
destination = (3,6)
moveit ( item, color, source, destination )

move cube blue red at (3,2) to (1,1)

destination = unknown
item = cube
color = unknown
source = unknown
color = blue
color = red
source = (3,2)
destination = (1,1)
moveit ( item, color, source, destination )

move ellipsoid at (3,6) at (2,1)

destination = unknown
item = ellipsoid
color = unknown
source = unknown
source = (3,6)
source = (2,1)
moveit ( item, color, source, destination )
```

Chapter V

THE PROTOTYPE SYSTEM

This chapter describes the use and general internal structure of the prototype implementation of the system. This implementation is meant solely to show the validity of the command approach and not to be a production system. As a result, convenience features are not present.

The system has been implemented on Digital Equipment Corporation's VAX/VMS system in Pascal. This system converts commands into the VAX/VMS Digital Command Language (DCL). Some of the lexical information that is specific to DCL is hard coded into the system and can not be modified without difficulty.

This chapter first discusses the lexical entities that the system can use, then how to set up and use the definitions, and finally the internal structure of the system.

5.1 LEXICAL ENTITIES

Each lexical entity is composed of characters. These characters are divided into classes. The classes of characters and their members are:

letter	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z $ _
digit	0 1 2 3 4 5 6 7 8 9
flag	a character defined with the commands
comment	a character defined with the commands
space	blank or tab
single quote	'
double quote	"
dot	.
colon	:

semicolon	;
left bracket	[
right bracket	]
separator	all other characters

Lower case letters are treated as upper case letters.

There are two types of lexical entities, those that are recognized by their form only, and those that are also recognized by their presence in a table. In this prototype system, only keys and variables are recognized by their presence in a table. When keys and variables are defined, they are recognized by their form only.

The lexical items recognized by the system for the command calls are:

key	a letter followed by any number of letters and digits
number	one or more digits
comment	a comment chararacter through the end of the line
filename	corresponds to the VAX/VMS standard without pattern matching and length restrictions
queuename	one or more letters or digits
separator	a separator character

The string ABC may be either a key, a file name, or a queue name. See appendix A section 2.1.

The lexical items recognized by the system for command definition are:

code	a sequence of lines which do not begin with the flag character
key	a letter followed by any number of letters and digits
variable	a flag followed by one or more letters or digits
number	one or more digits
comment	a comment chararacter through the end of the line
separator	a separator character
quoted	a single quote through the next single quote on that line or the end of the line

See appendix A section 2.2.

The implementation restricts keys, numbers, file names, variables, and queue names to a length of sixteen characters.

5.2 SYSTEM USAGE

The system takes as input two files, a file of command definitions and a file of command instances (or calls). The system creates two output files, a file containing the DCL code resulting from the expansion of commands, and a listing file.

A command procedure has been provided to control the execution of the interpreter. The first form,

```
$ @EXPAND filename
```

takes a file of definitions (filename.DEF) and a sequence of commands from the terminal terminated by the end of file marker (control Z), and outputs the expanded code to the terminal, and sends a listing to the file (filename.LIS).

The second form,

```
$ @EXPAND filename FILE
```

takes a file of definitions (filename.DEF) and a file of commands (filename.CAL), and produces a file of expanded code (filename.EXP) and a listing file (filename.LIS).

The third and final form,

```
$ @EXPAND filename EXECUTE
```

takes a file of definitions (filename.DEF) and a sequence of commands from the terminal terminated by the end of file marker (control Z), and outputs the expanded code to a file (filename.EXP), creates a listing file (filename.LIS), and then executes the command file (filename.EXP).

5.2.1 Definition File

In the definition file, the first line contains only the flag character. The second line contains only the comment character. These two lines define the flag and comment characters for the rest of the file. The rest of the file contains a sequence of command definitions as discussed in chapter 3 section 4.

5.2.2 Call File

The call file has a sequence of command instances imbedded in other text. Each command is terminated by an end of line.

5.2.3 Expanded File

The expanded file contains the expanded DCL code corresponding to the commands in the call file.

5.2.4 Listing File

The listing file contains an echo of the source file preceded by line numbers, an echo of the call file preceded by line numbers, and any error messages relating to the definition and analysis of commands.

5.3 INTERNAL STRUCTURE

The internal structure of the prototype implementation is fairly simple. The major objective was to implement the prototype quickly.

5.3.1 Lexical Analysis

The lexical analysis is table driven. Each entry in the table has an action and a next state. The action and its successor actions are done until the action is either `Return (token)` or `Null`. Then the machine advances to the next state.

5.3.1.1 States

The states are:

InDefine InLiteral (for control and code line distinction)
InBlank InKey InVariable InNumber (for simple recognition)
EndNode InDisk EndDisk InDirectory InName InExtension
        InVersion (for file name recognition)
InSingleQuoted EndSingleQuoted InDoubleQuoted
        EndDoubleQuoted (for quoted strings)
CharState LineState EndState (for control)

5.3.1.2 Actions

The actions manipulate the token to be returned by the Return action. The actions and their successors are:

Action	Successor
CreateNumber	AppendNumber
CreateFilename	AppendFilename
MakeFilename	AppendFilename
AppendNumber	GetNextChar
AppendFilename	GetNextChar
CreateKey	GetNextChar
AppendKey	GetNextChar
CreateVariable	GetNextChar
AppendVariable	GetNextChar
CreateQuoted	GetNextChar
AppendQuoted	GetNextChar
CreateLiteral	GetNextLine
AppendLiteral	GetNextLine
GetNextChar	Null
GetNextLine	Null
CreateSeparator	Return
CreateLineEnd	Return
CreateFileEnd	Return
CreateDefine	Return
IllegalChar	Return

Note that the Null and Return actions do not have successors.

5.3.2 Command Definition Syntactic Analysis

The syntactic analysis is driven by a recursive descent parser which builds a linked structure representing the command.

5.3.2.1 Definition Elements

The command definition parser recognizes (with procedures) the following elements:

1. empty control lines,
2. the rest (unused portion) of a control line,
3. a comma,
4. the final delimiting line of an element definition,
5. one of the keys REQUIRED or OPTIONAL,
6. a variable,
7. the key KEY followed by a literal key,
8. a separator character in quotes,
9. the single value specification or the list of values specification,
10. a code literal,
11. the heading line of a value definition,
12. a value definition,
13. the heading line of a modifier definition,
14. a modifier definition,
15. the heading line of a parameter definition,
16. a parameter definition,
17. the heading line of a command definition,
18. a command definition,
19. and finally a list of command definitions.

5.3.2.2 Linked Representation

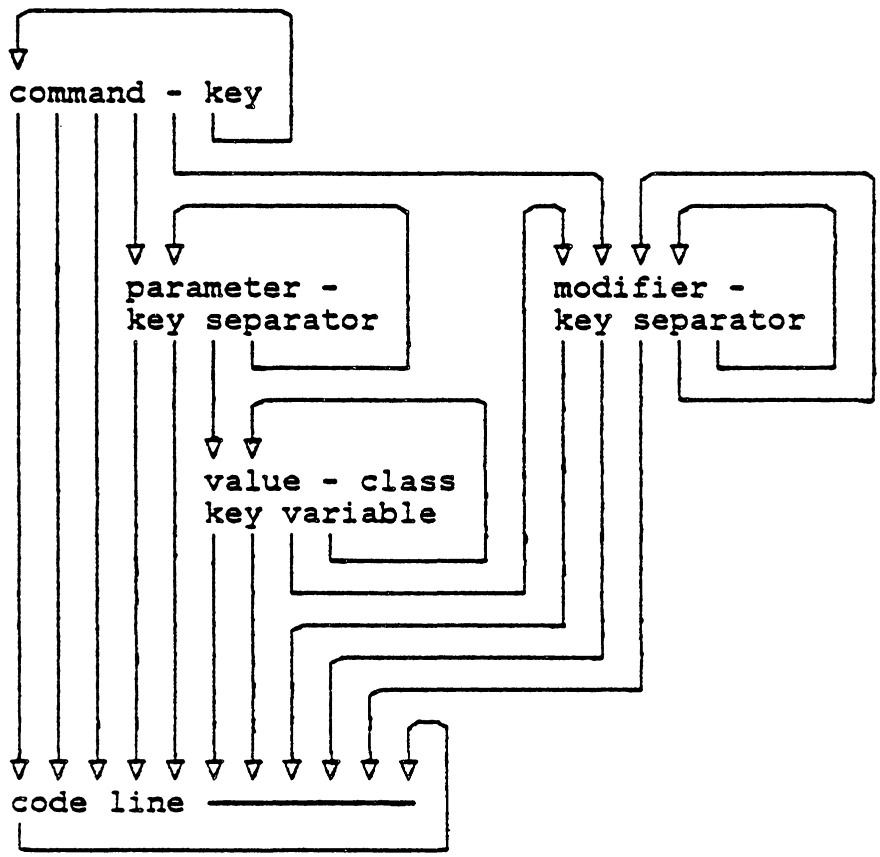
The command definition parser builds a linked list representing the command. For example, the command definition:

```
% command key COPY
% parameter required, value, key FROM optional
%   value %INFILE
$ from := %INFILE
%   vend
% pend
% parameter required, value, key TO required
%   value %OUTFILE
$ to := %OUTFILE
%   vend
% pend
$ copy 'from' 'to'
% cend
```

would generate the linked structure:

```
command key COPY
|    |
|    v
|    $ copy 'from' 'to'
v
parameter required  ----> parameter required
single value              single value
key FROM optional         key TO required
|                         |
v                         v
value                     value
filename %INFILE          filename %OUTFILE
|                         |
v                         v
$ from := %INFILE         $ to := %OUTFILE
```

In general, the possible links between the types of command components can be represented as follows:



5.3.3 Command Instance Syntactic Analysis

The command parsing is done by recursive descent guided by the linked representation of the command. The command instance is constantly matched to the linked representation of the command. The system matches depth first, so any substructures are matched (if possible) before continuing on the current structure.

Chapter VI

CONCLUSIONS

While the definable commands in this system may look less English-like than some earlier systems ([BARNM69], [GROVL82]), this system provides a much greater structural similarity to the English language. The approach of building a language to resemble English could allow (with sufficient development) computer interaction that achieves near native language flexibility while still being clearly defined and readily analyzable. Instead of trying to analyze English, we define a language approaching English.

6.1 FUTURE WORK

In order to make commands even more English-like, certain areas need further work. Some suggestions for the next steps in further refinement of commands are:

1. several key choices for an element to allow synonyms,
2. several keys before the keyed element to allow multi-word keys,
3. determiners to distinguish between singular and general referencing expressions,
4. modifiers with more than one value or list, and
5. adjectives in front of the value modified.

6.2 IMPLEMENTATIONS

The semantic capabilities of commands in this system are totally dependent on the host language. Therefore, a more powerful host language than the VAX/VMS Digital Command Language should be used. In addition, interactive commands require ability to dynamically create and execute procedures. This is possible in DCL, although not as convenient as in languages such as LISP.

The lexical environment of the host language should probably be defined with the commands instead of being defined in the interpreter code as has been done in the prototype. An extension to nonlexical values is a distinct possiblity. However, it will require cooperation between the command processor and the base language processor. The proper combination of this type of system with some of the compiler-compilers available, could provide a very powerful tool for the creation of interactive languages.

Interactive communication is a two way process. This paper has concentrated solely on one direction of communication. With more facilities designed for interactive use, such as prompting for missing required information, this type of system could provide a flexible method of defining significantly better interactive systems.

BIBLIOGRAPHY

The call numbers listed below were used by the Virginia Polytechnic Institute and State University Library at the time of this writing. They conform to the United States Library of Congress system, although the specific numbers may vary slightly from library to library.

[AKMAA79]   P 121 A4384

Adrian Akmajian, Richard A. Demers, and Robert M. Harnish; "Linguistics: An Introduction to Language and Communication"; The MIT Press; 1979.

[CHOMN65]

Noam Chomsky; "Aspects of the Theory of Syntax"; The MIT Press; 1965

[HILLI72]   QA 76 C55

I. D. Hill; "Wouldn't it be nice if we could write computer programs in ordinary English - or would it?"; The Computer Bulletin; June 1972, Volume 16 Number 6 Page 306.

[LEDGH80]   QA 76 A772

Henry Ledgard, John A. Whiteside, Andrew Singer and William Seymour; "The Natural Language of Interactive Systems"; Communications of the ACM; October 1980, Volume 23 Number 10 Page 556.

[BEECD79]   QA 76.7 I19

David Beech; "What is a Command Language?"; Command Language Directions; Proceedings of the IFIP TC 2.7 Working Conference on Command Languages; North-Holland Publishing Company; 10 - 14 September 1979, Page 7.

[SCHNM79] QA 76.7 I19

M. L. Schneider, R. L. Wexelblat and M. S. Jende; "Designing Control Languages from the User's Perspective"; Command Language Directions; Proceedings of the IFIP TC 2.7 Working Conference on Command Languages; North-Holland Publishing Company; 10 - 14 September 1979, Page 181.

[DOLOT79] QA 76.7 I19

T. A. Dolotta and J. R. Mashey; "Using a Command Language as the Primary Programming Tool"; Command Language Directions; Proceedings of the IFIP TC 2.7 Working Conference on Command Languages; North-Holland Publishing Company; 10 - 14 September 1979, Page 35.

[BARNM69] QA 76.5 B2943

Michael P. Barnett; "Computer Programming in English"; Harcourt, Brace and World, Inc.; 1969.

[GROVL82] QA 76 S16

Lindsay J. Groves; "Using Simple English Sentences to Call Procedures"; SIGPLAN Notices; November 1982, Volume 17 Number 11 Page 31.

[HALPM64] TK 7885 A1 J6 1964

Mark I. Halpern; "XPOP: A Meta-Language Without Metaphysics"; Proceedings of the AFIPS 1964 Fall Joint Computer Conference; 1964, Volume 26 Part 1 Page 57.

[BROWP67] QA 76 A772

P. J. Brown; "The ML/I Macro Processor"; Communications of the ACM; October 1967, Volume 10 Number 10 Page 618.

[SASSM78] QA 76 S6

Masataka Sassa; "A Pattern Matching Macro Processor"; Software - Practice and Experience; June 1978, Volume 9 Number 6 Page 439.

[LEAVB66] QA 76 A772

B. M. Leavenworth; "Syntax Macros and Extended Translation"; Communications of the ACM; November 1966, Volume 9 Number 11 Page 790.

[TRIAJ80] QA 76 A772

J. M. Triance and J. F. S. Yow; "MCOBOL - A Prototype Macro Facility for Cobol"; Communications of the ACM; August 1980, Volume 23 Number 8 Page 432.

Appendix A

FORMAL GRAMMARS

These grammars are based on a regular expression where the items enclosed in parentheses '()' may occur zero or more times, the items enclosed in brackets '[]' are optional, and the items enclosed in braces '{}' and separated by a bar '|' represent a set of alternatives, one and only one of which must be chosen.

A.1 SYNTACTIC

Spacing is not specified in these grammars, but is assumed between all elements except where separators or terminators intervene. The separator is specified by the macro and the terminator is specified by the macro expander.

A.1.1 Call

The following grammar represents the model over which the individual commands are formed. The possible instances for a command represent a subset of the language specified by this grammar.

```
command =  key ( modifier ) ( parameter ) line_end
modifier =  key ( modifier ) [ { value | list } ]
parameter =  [ key ] { value | list }
list =  value ( separator value )
value =  entity ( modifier )
entity  =  { key | file_name | queue_name | number }
```

A.1.2 Definition

This section defines the grammar for specifying the set of instances for a command. The items defined in "?_start" and "?_end" definitions must be on the same line. Items in other definitions must be on separate lines.

```
command = c_start [ code ] ( modifier ) [ code ]
          ( parameter ) [ code ] c_end

c_start = 'COMMAND' 'KEY' key

c_end = 'CEND'

parameter = p_start [ code ] value ( value ) [ code ] p_end

p_start = 'PARAMETER' { 'REQUIRED' | 'OPTIONAL' }
          ',' { 'VALUE' | 'LIST' 'BY' separator }
          [ ',' 'KEY' key { 'REQUIRED' | 'OPTIONAL' } ]

p_end = 'PEND'

value = v_start [ code ] ( clause ) [ code ] v_end

v_start = 'VALUE' { 'KEY' key | 'FILENAME' subst_var
          | 'QUEUENAME' subst_var | 'NUMBER' subst_var }

v_end = 'VEND'

modifier = m_start [ code ] ( modifier ) [ code ]
           ( value ) [ code ] m_end

m_start = 'MODIFIER' { 'REQUIRED' | 'OPTIONAL' } ',' 'KEY' key
          [ ',' { 'VALUE' | 'LIST' 'BY' separator } ]

m_end = 'MEND'
```

A.2 LEXICAL

Keys, numbers, variables, file names, and queue names must be separated from each other by one or more spaces, or one of the other lexical items.

A.2.1 Call

The following represent the structure of the lexical items recognized in an instance of a command.

```
key = letter ( { letter | digit } )

number = digit ( digit )

comment = comment_character ( any ) line_end

filename = [ { letter | digit } ( { letter | digit } ) colon
             [ colon ( { letter | digit } ) colon ] ]
           [ left_bracket ( { letter | digit | dot } )
             right_bracket ]
           ( { letter | digit } )
           [ dot ( { letter | digit } ) ]
           [ semicolon ( { letter | digit } ) ]

queuename = { letter | digit } ( { letter | digit } )

separator = separator_character

quoted = double_quote ( any ) { double_quote | line_end }
```

Note that there is an overlap in the definition of a filename, a queuename and a key. The interpreter will accept a key as the instance of a filename or a queuename.

A.2.2 Definition

The following represent the structure of the lexical items recognized on a control line in the definition of a command.

```
key =  letter ( { letter | digit } )

variable =  flag ( { letter | digit } )

number =  digit ( digit )

comment =  comment_character ( any ) line_end

separator =  separator_character

quoted =  single_quote ( any ) { single_quote | line_end }
```

Appendix B

EXAMPLES

Each example in this appendix contains three parts -- a command definition, a command instance (or call), and the resulting expanded code. In all cases, the flag character is the % character.

B.1 COPY

```
% command key COPY
% parameter required, value, key FROM optional
%   value filename %INFILE
$ from := %INFILE
%   vend
% pend
% parameter required, value, key TO required
%   value filename %OUTFILE
$ to := %OUTFILE
%   vend
% pend
$ copy 'from' 'to'
% cend
```

```
copy A.B to C.D;1
```

```
$ from := A.B
$ to := C.D;1
$ copy 'from' 'to'
```

B.2 PRINT

```
% command key PRINT
$ default_header := /noheader
$ current_header := /noheader
$ print_queue := sys$print
% modifier optional, key WITH, value
%   value key HEADER
$ default_header := /header
%   vend
% mend
% modifier optional, key WITHOUT, value
%   value key HEADER
$ default_header := /noheader
%   vend
% mend
% modifier optional, key ON, value
%   value queuename %QUEUE
$ print_queue := %QUEUE
%   vend
% mend
% parameter required, list by ',', key FILES optional
%   value filename %PRINTFILE
$ current_header := 'default_header'
%     modifier optional, key WITH, value
%       value key HEADER
$ current_header := /header
%       vend
%     mend
%     modifier optional, key WITHOUT, value
%       value key HEADER
$ current_header := /noheader
%      vend
%    mend
$ print/queue='print_queue''current_header' %PRINTFILE
%   vend
% pend
%cend
```

```
print with header files A:B, [C].D without header, E
```

```
$ default_header := /noheader
$ current_header := /noheader
$ print_queue := /sys$print
$ default_header := /header
$ current_header := 'default_header'
$ print/queue='print_queue''current_header' A:B
$ current_header := 'default_header'
$ current_header := /noheader
$ print/queue='print_queue''current_header' [C].D
$ current_header := 'default_header'
$ print/queue='print_queue''current_header' E
```

B.3 COMPILE

```
% command key COMPILE
$ optimize := /nooptimize
$ debug := /nodebug
$ list := /nolist
$ symbols := /nocross
$ code := /nomachine
$ source : string ;
% modifier optional, key WITH, list by ','
%   value key OPTIMIZE
$ optimize := /optimize
%   vend
%   value key DEBUG
$ debug := /debug
%   vend
% mend
% parameter required, value, key FILE optional
%   value filename %SOURCEFILE
$ source := %SOURCEFILE
%     modifier optional, key LISTING
$ list := /list
$ listing :=
%       modifier optional, key WITH, list by ','
%         value key SYMBOLS
$ symbols := /cross
%         vend
%         value key MACHINE_CODE
$ code := /machine
%         vend
%       mend
%       modifier optional, key ON, value
%         value filename %LISTFILE
$ listing := "=%LISTFILE"
%         vend
%       mend
%     mend
%   vend
% pend
$ fortran'optimize''debug''list''listing'-
'symbols''code' 'source'
% cend
```

```
compile with optimize A::B:E.F;1 listing with symbols on G
```

```
$ optimize := /nooptimize
$ debug := /nodebug
$ list := /nolist
$ symbol := /nocross
$ code := /nomachine
$ optimize := /optimize
$ source := A::B:E.F;1
$ list := /list
$ listing :=
$ symbols := /cross
$ listing := "=G"
$ fortran'optimize''debug''list''listing'-
'symbols''code' 'source'
```

The vita has been removed from the scanned document

A MACRO SYSTEM FOR ENGLISH-LIKE COMMANDS

by

Lawrence Alan Crowl

(ABSTRACT)

This thesis describes a system that allows the designer of an interface between humans and computers to implement an English-like set of commands easily. Designers model commands on imperative English sentences containing verbs, direct and indirect objects, adverbs, adjectives, and prepositional phrases. The system restricts the ordering and presence command components as little as possible while still allowing the commands to be analyzed in a deterministic manner. Thus, commands formed with the system can mirror not only the form of English, but the flexibility of English as well.

This system is based on the macro expansion of a command into a base language, and therefore has only the expressive power of the base language. Because of this, the system can only aid in the definition of commands, and not in the definition of a language.

The language for describing the commands is structural in nature. The programmer defines the parts of a command instead of the syntax of the command. With each part of the command, the programmer defines sections of code that will be generated at various points during the recognition of the command. An implementation can execute generated code at some later time, at the end of the translation, or during the expansion. A prototype implementation expands of a file of commands into the VAX/VMS Digital Command Language.