A TECHNIQUE FOR IMPLEMENTING INTERACTIVE CONVERSATIONS

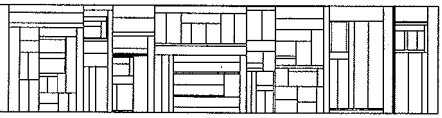
by

Michael J. Manthey

DAIMI PB-57

May 1976

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55



A TECHNIQUE FOR IMPLEMENTING INTERACTIVE CONVERSATIONS

by
Michael J. Manthey

ABSTRACT

A general purpose parser for interactive command language construction is described. The method takes advantage of the characteristic of mini-computers that they are very often dedicated to a single user and thus during interaction can devote their computing capability entirely to the parsing (i.e. the parsing algorithm can be "wasteful" of computer time). Tables for the command language are generated by a different program which accepts an LL(1) grammar of the command language and produces output suitable for an assembler or compiler. The parsing algorithm is non-recursive and suitable for either character or graphic screens, but probably not for hard-copy devices.

Keywords:

command language
minicomputer
interactive parsing
table-driven parsing

1. INTRODUCTION

This paper describes the design considerations and some implementation aspects of a parser specifically intended for interactive conversations. As such, the parsing technique must cater to those aspects of conversational interaction which differ from batch-interaction, to wit:

1. Naturalness of expression, such that what the user should type next is immediately apparent without reference to a manual, and thus more consistent with the "off the top of the head" nature of normal conversation;

2. Rapid feedback of results, so that the conversational atmosphere is maintained;

3. Rapid feedback of errors in a non-disruptive fashion, such that the conversation can be continued rather than repeated.

It can rightly be argued that these characteristics apply to all forms of interaction with a computer; the argument here is that in the intimacy of interactive conversation, they are dominant, and if not satisfied will result in clumsy (and in the long run, annoying) conversations.

In addition to these general requirements, the design of the parsing technique was also influenced by some local, though not uncommon, considerations: an unsophisticated user community in possession of a minicomputer and CRT terminal. [The actual problem was to implement a real-time sound synthesizer for the Music Department.]

2. A SOLUTION

The above considerations led to the design of a table-driven command language parser which is based on three assumptions:

1. There is an excess of compute time when the program is communicating with the user;

2. There is a fast (e.g. ⩾ 1200 baud) *silent* terminal attached;

3. Experienced users will not be annoyed by extra text as long as it is the computer, not them, who is typing it.

The first of these assumptions means that the parsing algorithm can be computationally lengthy, since in all cases it will still be fast enough to provide instant response. This recognizes the fact that using a computer to interpret typical commands is basically analogous to using dynamite to pull dandelions; and that response times under 0.1 second are meaningless to humans.

The second assumption means that the user is interacting via a screentype terminal which can write as much as several lines of text essentially instantaneously. There is thus no problem of users becoming annoyed by waiting for the computer to type text, nor by any associated noise such as one would get from a hard-copy terminal.

The third assumption builds on the second by postulating that as long as the experienced user doesn't have to wait for or hear explanatory text, she will be unannoyed by it. In fact, after having seen it N times, she will cease to see it at all. On the other hand, the inexperienced user can receive reinforcement for what a given command means each time she uses it, thus giving her a security blanket and educating her at the same time.

2.1 Basic Language Characteristics

The explanation of how the command language itself looks starts with an example of a typical (synthesizer) command seuquence, in this case a command CREATE, which the user invokes to define a sound. in our *first* (quick and dirty) command interpreter she typed (italics letters are what the user types):

C...VIOLIN ↵

where " ... " means any number of blanks are allowed, and " ↵ " means carriage return. She was then presented with a series of questions regarding this sound she wished to define. This type of single letter command is unfortunately

endemic, and was clearly unacceptable for long term use.

In the new command interpreter the following happens:

*CR*EATE A SOUND PATTERN CALLED: *VIOLIN*

Thus, the user types only a sufficient number of characters to disambiguate the command from any others, whereafter the command with explanation is zipped out without any further action (e.g. ↵) on the part of the user. At each point where the user is asked for input, she may respond with a "?" (and no ↵) and receive a further explanation, which afterwards returns her to the point of questioning. (Note: this "returning" mechanism is not shown here, but is done by specifying *{<create 1>}; see below.)

The idea of requiring only disambiguating characters from the user is not very startling – variations on this theme appear in a number of operating systems [1,2]. Perhaps some originality can be found in completing not only the command mnemonic itself but also adding a supplementary explanation. However, the most interesting part is how this command was specified:

```
<create>::='CR'  (EATE A SOUND PATTERN CALLED:)  <create1>;
<create1>::=<ident> | '?' <info.create>;
```

The above specification reads "the command called create is defined to be the user-typed letters CR followed by the prompting text EATE A SOUND PATTERN CALLED: followed either by an identifier (including ↵) or a ? followed by the prompting text called info.create".

In general, the syntax of the command language is specified in an augmented BNF, as shown in Table 1. This "language for specifying command languages" is called a command metalanguage, hereinafter referred to as the metalanguage or grammar. The metalanguage is processed by a (SNOBOL 4) program, which generates a (mildly optimized) tree representation of the input grammar (see Figure 1). This tree representation is input to an assembler together with the (hand coded) parsing algorithm and language dependent semantic routines, resulting in a complete command language interpreter.

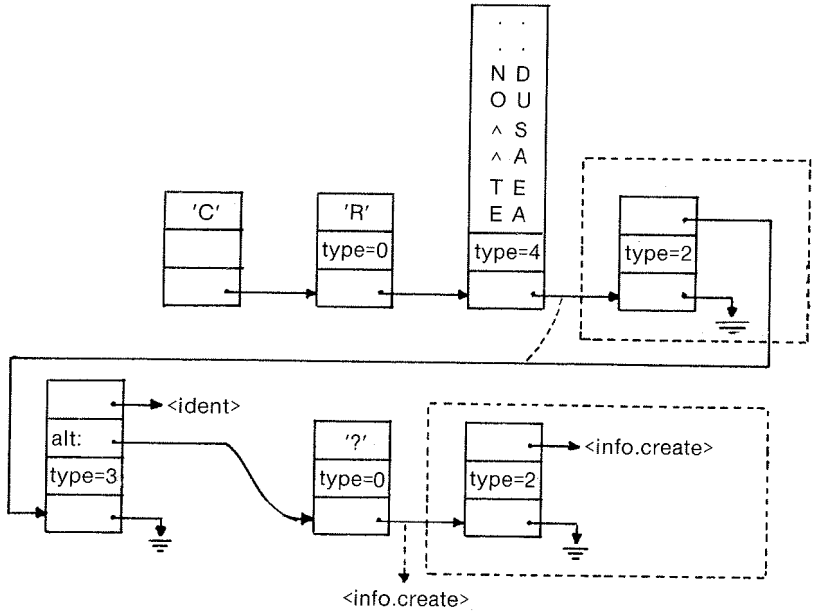
Symbol	*Terminology*	*Meaning*
' ' or " "	Terminal character/symbol	a user-input character
ANY(...)	string of alternative terminal symbols	a user-input character from the listed set
< >	non-terminal symbol	another object in the grammar
( ) or [ ]	prompting text	text to appear automatically on the screen in pass one
1.ident or 1.ident(...)	pass 1 routine	a piece of code to be executed in pass one of the parse
2.ident or 2.ident(...)	pass 2 routine	a piece of code to be executed in pass two of the parse
ident or ident(...)	pass 1 and 2 routine	a piece of code to be executed in both passes of the parse
*error	error mark	marks a place in the grammar to which backup will be done if there is a pass one error
*[...]	repetition	specifies zero or more occurrences of the included syntax
empty	empty symbol	generated by the table generator program in the reduction of repetition clauses
|	alternative	"OR"
::=		"is defined to be"

Table 1
Syntactic Constructs in the Command Meta-Language

2.2 The Parsing Algorithm

As mentioned above, the data structure operated upon by the parsing algorithm is a tree representation of the input grammar. This tree is a set of nodes, each in the form of a block of contiguous memory cells, which contain the type of the node, a pointer to next node, an optional pointer to an alternative node, and optional values. It is the job of the parsing algorithm to march through this tree, choosing a path based on the user 's input. As this is done, the algorithm maintains a stack (called the history stack) of each node which is on the path of the final successful parse, as well as a parse stack which is a mechanical aid for keeping track of where it is in the tree. It also executes, as it stumbles over them, the pass one routines specified by nodes of type "pass 1 routine" and writes out the prompting text specified by nodes of type "prompting text".

When the algorithm has associated the entire syntax tree of a command with characters from the input stream, it is said to have generated a complete parse, and it therefore will switch over to pass two. (This changeover can also be forced by specifying a pass one routine which has this effect.) The second pass of the algorithm starts at the base of the history stack, and examines each of the nodes in order from earliest to most recent. In this scan of the history stack, only terminals (i.e. terminal character and ANY nodes) and pass-two-routine nodes are of interest. The encounter of a terminal node causes the input character to be rescanned; the routine specified in the pass-two-routine node is executed. When the last history node has been consumed, the history stack is cleared and parsing continues from the point in the grammar specified by the parse stack.



<create>::='CR' (EATE A SOUND PATTERN CALLED) <create1>;
<create1>::=<ident> | '?' <info.create>;

Notes

1. The "ground" symbol signifies a null pointer value i.e. end of list.

2. The values of the "type" field are shown in Figure 2.

3. The dotted lines show the principal from of optimization performed by the SNOBOL 4 program: deletion of non-terminals used only one time.

4. The 'CR' of the command definition becomes 'C' followed by 'R' to ease the backspace process (see 2.3.2).

Figure 1.
Data Structure for the <create> Command

2.3 Other aspects of the parse technique

2.3.1 Lexical Analysis

A lexical prescan of the input stream in which such things as identifiers, numbers, and keywords are gathered together was held to be unnecessary or even a disadvantage in the case of an interactive command language. Such a prescan is commonly included in compilers to remove such trivial processing from the actual parsing, usually in the interests of speed. However as stated earlier, such considerations do not apply here . On the other hand, the code for a lexical analyzer could easily consume a large percent of the total for the algorithm; this code also represents a function which is within the capabilities of the (already existing) parsing algorithm. From a different point of view, a prescan which unrecoverably groups terminal symbols together considerably complicates error recovery, e.g. in the case of user typing errors.

2.3.2 Backspacing

Continuing the discussion of user typing errors, another use of the history stack is now revealed. When the user discovers she has made a typing error, it must be easy for her to backspace over the error(s) and then reenter new symbols. It is not really acceptable that she be required to reenter the entire line or command. Thus upon receipt of a backspace character, the parse stack is backed up by using the information contained in the history stack.

At this point it is important to remember that the user backspaces according to what she sees on the screen, which includes both things she has typed and machine generated prompting text. The algorithm therefore backs up to the preceding terminal for each received backspace character. The screen is correspondingly blanked. It is worth noting that all intervening prompting text (between two user inputs) is erased by a single backspace character.

2.3.3 Pass One and Pass Two

It was not explicitly stated in the previous discussion that backspacing may only take place in pass one. This is consistent with the underlying philosophy of having two passes, in which the first pass takes care of the syntactic correctness of the input, and the second, the semantic execution implied by the input. In practice, the pass one functions are used to check the superficial semantic correctness of the input e.g. that input identifiers exist and numbers are within range. It is convenient for the user that such checking is done in the first pass, since here she retains her ability to backspace and reenter the information. It is

convenient as well for the implementor, since she can then undertake (possibly unretractable) semantic actions in pass two with the knowledge that a certain level of semantic consistency is assured.

There is of course nothing which prevents the implementor from semantic execution of the command in the first pass, but this should only be done with an awareness of its implications for error recovery. It should perhaps be noted that the distinction between the two passes is invisible to the user, except insofar as an intervening execution of pass two inhibits backspacing over the input involved.

2.3.4 Syntactic Error Detection

The basic requirement laid on the grammar is that it be an LL(1) grammar, which means that at any point where there are multiple paths in the grammar, the decision as to which path is the correct one can be taken by looking at he next input symbol. Thus it is impossible that the decision taken is the wrong one i.e. the decision process is deterministic. This means that each alternative in the grammar *must* begin with a contextually unique symbol, or a non-terminal which eventually yields such a terminal (in which case, no other path could lead to a match). (Not allowing such non-terminals showed itself to be unacceptable during implementation, since it precluded the usage of common non-terminals such as <number> and <ident>; the result was much greater table size.) To conserve space, the actual implementation allows a limited amount of backtracking. However, the LL(1) property of the grammar (verified by a separate program) ensures that any attempt to backtrack over a terminal node always indicates a user syntax error.

2.3.5 Semantic Error Recovery

Using the LL(1)-ness of the grammar is one method of trapping user syntax errors. Another technique is to include an explicit alternative in the grammar which matches "anything else" the user might write. In either case the user must be signalled that she has made an error, perhaps by writing some text on the screen or (as we have done) refusing to accept further characters from the keyboard and blinking the erroneous character. The user having been signalled, it is now up to her to use the backspace function to correct her mistake by backing up the parse. We now discuss recovery from semantic errors.

As shown in Table 1, there is a node type called ERROR which, when encountered by the parsing algorithm, is merely placed on the history stack. If during execution of a pass one routine it is desired to signal an error, the appropriate text can be written on the screen and the algorithm requested to back up to the most recently encountered ERROR node on the stack. Since it is still in

pass one, the parsing will restart, and any prompting text encountered on the restart will be written onto the screen. This means that the implementor, in designing her grammar, has control over what errors are to be detected where, where the parse is to be restarted, and what the user thereafter sees on the screen.

2.3.6 Applications to Other Dialogue Forms

The parsing technique could also be used in systems having a light pen or using a "frame" type of presentation. In the former, some of the terminal symbols specified in the grammar would correspond to lightpen hits. In the latter, the mechanisms provided by the grammar could be used to control the sequencing from one information frame to another, where an information frame is a screen-sized prompting text. Clearly, the two techniques could be combined in a dialogue form featuring light-pen selection of items displayed in a menu frame.

3. IMPLEMENTATION

3.1 The Meta-language

This rules for writing a grammar are:

1. A meta-language identifier may be any string of characters (not beginning with "$", or containing ">");

2. Both quote (") and apostrophe (') are available for enclosing terminal symbol strings;

3. Both parentheses ( ) and brackets [ ] are available for enclosing prompting text;

4. Input is free format and blanks are not significant except within text strings;

5. Special characters such as carriage return, line feed and all the other non-printing characters may be assigned mnemonics and values of the (meta-language) user's choice;

6. Output from the generator program is easily modifiable for input to a variety of assemblers and compilers.

These syntactic demands, coupled with a desire for quick implementation of a machine-portable program, resulted in a program being written in SNOBOL 4 which accepts the grammar for the command language and produces output (the tree form of the grammar) suitable for input to the assembler of the machine in question. Clearly, however, any language with reasonable string handling and recursion could be used.

3.2 The Parsing Algorithm

The command language interpreter exists as a free-standing task which is activated by messages from the screen driver, each message containing one character. It is the command interpreter's job to echo this character and otherwise feed it to the parsing algorithm, which consumes it and returns to a state of waiting for the next character. For the purposes of the second pass, a stack of the "recognized" terminal symbols is maintained which is pushed by terminal nodes and popped during backspacing.

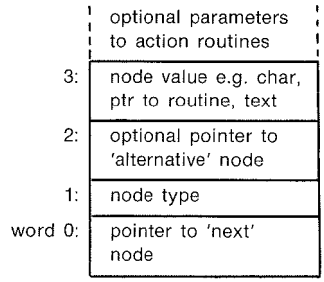
The parsing algorithm itself is non-recursive and consists of a short code sequence for each node type.

3.3 The Tree

As mentioned before, the grammar is represented as a tree whose nodes specify the node type, value, and pointers to other nodes. The detailed format of a node is shown in Figure 2. It should be noted that in the interests of conserving memory space, nodes having no alternative have no cell devoted to that purpose; also that auxiliary node values such as the value of the terminal symbol(s), prompting text, and action routine parameters appear with the node itself, rather than having a pointer devoted to that function.

Depending on the size of the grammar and the capabilities of the receiving assembler or compiler, it would be advantageous to pack these fields more tightly.

Node Format



Node type	*Memory Consumption*	*Description*
0	3	Terminal symbol
1	4	Terminal Symbol with alternative
2	2	Nonterminal Symbol
3	3	Nonterminal Symbol with alternative
4	> 2	Prompting text
5	> 3	Prompting text with alternative
6	⩾ 3	Pass Two action routine
7	⩾ 4	Pass Two action routine with alternative
8	2	Empty Symbol
9	3	Empty Symbol with alternative
10	> 2	ANY
11	> 3	ANY with alternative
12	⩾ 3	Pass One action routine
13	⩾ 4	Pass One action routine with alternative
14	2	Error mark
15	3	Error mark with alternative

Figure 2
Format of a Tree Node

Figure 3 shows a sample grammar and the output of the SNOBOL 4 program. One pleasant but unanticipated side-effect is also illustrated: that by defining commonly used prompting texts with a non-terminal, non-trivial savings in string storage can be realized painlessly.

```
<ENVG>          ::= [TYPE THE POINTS CONSTITUTING ] 1.WRITESTR(EGG.IDENT) [:]
                <SYN.NEWLT2> 1.SETZERO(EGG.COUNTER)
                1.PASS2
                *[1.INC(EGG.COUNTER,150) <ENVIN> ] ≠CR≠;

  [NBR].....<ENVELOPE POINT>

<ENVIN>         ::= *ERROR <SYN.NEWLT2> 1.WRITENBR(EGG.COUNTER) <SYN.BLANKS10>
                <SYN.NUMBER> <CHECKIT> 2.MOVTOLIST(EGG.NBRPTR,EGG.COUNTER)
                1.PASS2;
<CHECKIT>       ::= 1.BETWEEN(1,15) / [...MUST BE BETWEEN 1 AND 15 ]
                [- TRY AGAIN.] [BELL] 1.P1ERROR;
```

Figure 3a
A Sample Grammar

The resulting screen contents after a few repetitions of <envin> are:

```
TYPE THE POINTS CONSTITUTING    XYZ:
    1     13
    2      3
    3     25 ... must be between 1 and 15 - try again.
    3     15
    4      7
    5
```

Notes

1. <syn.nlt2> is shared prompting text which contains carriage return, line feed, and indenting spaces.

2. Pass two is forced twice to minimize history stack space, not because it is logically necessary.

3. "INC" is a function which increments the first argument and fails when the second argument is exceeded, thus terminating the repetition.

4. "BETWEEN" is a function which fails if the value of EGG.NBRPTR falls outside of the arguments. This value is in essence the 'value' of <SYN.NUMBER>, the shared definition of the syntax of a number. The error problem mentioned in Section 4.2 does not occur here because <SYN.NUMBER> contains no error nodes.

5. "CR" and [BELL] are examples of strings (representing carriage return and terminal bell resp.) which have been given special meanings in the grammar compiler. The CR is necessary to allow termination of the data entry before INC necessarily fails.
   [Note: This should not be confused with the 'CR' in the CReate example, which in actual practice would be written 'C' 'R' to avoid translation to a carriage return.]

```
;
;
;<ENVG>
ENVG        DATA    F2,4,X≠5459≠,X≠5045≠,X≠2054≠,X≠4845≠,X≠2050≠ ;

            DATA    X≠4F49≠,X≠4E54≠,X≠5320≠,X≠434F≠,X≠4E53≠,X≠5449≠ ;

            DATA    X≠5455≠,X≠5449≠,X≠4E47≠,X≠2000≠ ;

F2          DATA    P3,12,WRITESTR,EGG.IDENT ;

P3          DATA    N4,4,X≠3A00≠          ;

N4          DATA    F5,2,SYN.NEWLT2       ;

F5          DATA    F6,12,SETZERO,EGG.COUNTER ;

F6          DATA    N7,12,PASS2           ;

N7          DATA    T10,2,XXREP1          ;

T10         DATA    X≠0000≠,10,X≠0000≠    ;

;<XXREP1>
XXREP1      DATA    S11,13,XXEMPTY,INC,EGG.COUNTER,150 ;

S11         DATA    N12,14                ;
N12         DATA    F13,2,SYN.NEWLT2      ;

F13         DATA    N14,12,WRITENBR,EGG.COUNTER ;

N14         DATA    N15,2,SYN.BLANKS10    ;

N15         DATA    N16,2,SYN.NUMBER      ;

N16         DATA    C17,2,CHECKIT         ;

C17         DATA    F18,0,MOVTOLIST,EGG.NBRPTR,EGG.COUNTER ;

F18         DATA    XXREP1,12,PASS2       ;

;<CHECKIT>
CHECKIT     DATA    X≠0000≠,13,P20,BETWEEN,1,15 ;

P20         DATA    F21,4,X≠2E2E≠,X≠2E4D≠,X≠5553≠,X≠5420≠,X≠4245≠ ;

            DATA    X≠2042≠,X≠4554≠,X≠5745≠,X≠454E≠,X≠2031≠,X≠2041≠ ;

            DATA    X≠4E4A≠,X≠2031≠,X≠3520≠,X≠2D20≠,X≠5452≠,X≠5920≠ ;

            DATA    X≠4147≠,X≠4149≠,X≠4E2E≠,X≠0700≠ ;

F21         DATA    X≠0000≠,12,P1ERROR    ;
```

Figure 3b
The Compiled version of 3a

4. DISCUSSION

The algorithm described above has been in use for over a year and has been used not only for the synthesizer's rather large (200 productions) grammar , but also for the coding and decoding of the system's binary tape format (a demonstration of the misuse as well as the robustness of the technique). It was used with great success in the creation of an interactive debugger, and has been re-implemented on several other machines. In spite of these rather gratifying results, however, a number of weak spots have revealed themselves, the discussion of which is the topic of this section

4.1 The Choice of an LL(1) Grammar

The primary attraction of the LL(1) grammar is that a syntactically incorrect character is immediately recognized and can be brought to the typist's attention. The value of this property should not be underestimated, but nevertheless, this strictness can be an annoyance to the implementor when she wishes to direct the parse based on computed, rather than input, values. A typical example arises with typed identifiers: depending on the type of an identifier, one of several possible grammatical possibilities must be chosen. The differentiating information is found in the symbol table, and *not* in the input string.

In the implementation described above, this problem is solved by allowing pass one routines to return a 'failure' indicator which causes the parsing algorithm to back up and try the next alternative in the parse stack. While this solution is adequate, it is hardly elegant (the grammar is in fact no longer LL(1)), and therefore it might be worthwhile investigating other types of grammars.

4.2 Error Nodes

The idea behind the introduction of Error nodes was to allow the implementor to express where the parse should be backed up to when a semantic error occcured. This idea is fine as far as it goes, e.g.

```
        <ident>::=*ERROR.....<chklength>.....;
<chklength>::=1.TEST(IDLENGTH, MAXIDLENGTH) | {text} 1.signalerror;
```

The example shows the checking of an identifier's length against the maximum allowed identifier length - the pass one routine TEST fails (as explained in the preceding section) if the maximum length is exceeded, causing the parser to try the alternative, which first writes some explanatory "text" on the screen and then signals the parser to back up to the preceding error node.

Unfortunately, a problem arises at the next level in the grammar e.g.

<newid>::=*ERROR.....<ident><chkid>;
<chkid>::=1.doesntexist | {text} 1.signalerror;

The routine 'doesntexist' checks to make sure that the new identifier which is being entered is indeed new. The intent of the above is to restart the parse at <newid> if the identifier already exists, but the error signal causes a retreat to the first error node in the history stack, which examination of the previous example will demonstrate to result in a restart at <ident>.

Although this example is perhaps slightly artificial, in practice the problem considerably complicates the construction of a consistent approach to the handling of semantic errors. The problem stands in clear analogy to the problem of exit from nested loops in programming language design, for which the following three solutions have been proposed: exit to a labelled point in the program, exit from N of the loops, or exit to the exclosing loop. The latter represents the parser's current (unsatisfactory) approach, while the specification of a constant number of Error nodes to be ignored makes the grammar's correct functioning highly dependent on the grammar writer's understanding of the parsing technique. The first alternative, labelled Error nodes, is very attractive but complicates the "grammar compiler" if one desires to check that the labelled nodes can indeed be reached from where they are referenced (by 1.signalerror(label)).

4.3 Reentrant Grammars

The synthesizer implementation of parser is written in assembly language, and it was natural that the grammar tables also be presented in that language. Without much forethought, the parameters to the action routines were typical of that language: references to actual addresses in memory. As it later turned out, this could have been (but in the case at hand was fortunately not) a grave error, since highly recursive use of productions is effectively precluded, not to mention shared usage of the same grammar by multiple users.

As example of the usefulness of reentrant grammars, consider a command language where in the middle of conversational sequence A it becomes clear that sequence B should have been performed first. In most systems, the user has no alternative but to drop the current sequence and perform B, then reperform A. If the grammar is reentrant, however, then B can be made an alternative of A at the appropriate point, thus allowing the user to 'parenthetically' perform the omitted sequence, and then resume. Furthermore, this philosophy can be applied throughout the grammar, leading to a much friendlier user interface.

4.4 Miscellany

1. It is difficult to arrange for unconditional escape from an arbitrary command sequence given the intent of the two-pass parsing strategy. Besides building in such unconditional escape, one could provide it explicitly as an alternative to all or selected (non-critical) productions, or as we have done, insist that the user finish a command she has started upon. {The beginning dialogue can namely always be backspaced out of.}

2. It was stated earlier that syntax errors are signalled by blinking the offending character, but what this means is not obvious if the character is a blank or otherwise non-printing. We chose to echo such characters as a cross-hatch, but this is permissable only if such characters are infrequent and/or erroneous. A related problem is selective suppression of echoing, which must be integrated with care into the algorithm because of the backspacing.

3. It is important to gather the constituent characters of <number>, <ident>, and suchlike so that backspacing does not imply updating of non-parser data structures. This is done by remembering where the character stack pointer is when the production is begun and then gathering the characters together when the closing symbol (e.g. carriage return) is received.

5. CLOSING COMMENTS

The author's most notable experience in implementing this system and then using it came when it was time to construct the grammar for the actual command language. It was striking what a difficult task it is to design clear, precise, and meaningful interactive dialogue. The fact that I could (luxuriously) worry about *what* to say rather that how to say it testified strongly for the suitability of the algorithm for the intended application.

Acknowledgements

I would like to thank Erik Meineche Schmidt for his help in the formal language theoretical aspects of this work; Henning Jeppesen, who has labored diligently throughout; and Peter Houman, who helped greatly in the early implementation.

References

1. TENEX, a Paged Time Sharing System for the PDP-10. Bobrow, Burchfield, Murphy, Tomlinson. CACM 15,3 March 72, p.139.

2. Scope Operators Guide, Control Data Corp. Pub. 60327300, 1971, p.1-3.