1985

Creating Havoc: Havoc Development Program

David Cohn
Dartmouth College

Stephen Madancy
Dartmouth College

CREATING HAVOC - HAVOC DEVELOPMENT PROJECT

Stephen Madancy and David Cohn

Technical Report PCS-TR86-121

Creating Havoc
Havoc Development Project

An undergraduate senior honors thesis
submitted by David Cohn and Stephen Madancy
in partial fulfillment of the
requirements for the degree of
Bachelor of Arts in Computer Science

Table of Contents

Part I: Preliminary Status Report

Part II: User's Guide to Havoc

Part III: Havoc's Assembly Language Primitives for the 68000

```
From pablo Tue May  8 16:10:43 1984
Received: by dartmouth.CSNET (4.12/1.7) id AA01859; Tue, 8 May 84 16:10:31 edt
Date: Tue, 8 May 84 16:10:31 edt
From: pablo (David Cohn)
Message-Id: <8405082010.AA01859@dartmouth.CSNET>
To: mahler
Status: RO

hey.... in spiration. Y' know how you've been talking about doing a
project that involves competer interaction with real world (the word back
there
should have been computer, not competer), ahardware?.....
and how I've always been babbling about language design instead?
HOW ABOUT.....: A specialized language e/environment for real world
interactions? complete with necessary engineered hardware.......
hmmmm..... sherman has pointed me to some refernces.

                                                        -pablo
```

Summary

To some extent, programming languages are designed to meet some specific need of a field of work or application; COBOL was designed to meet the needs of business computing, and the purpose of BASIC was to allow people to program without extensive computer background or mathematical training.

One area where use of the computer is essential is in the modern scientific laboratory. High speed computation, data storage and analysis enable scientists to perform experiments that would otherwise be entirely impractical. To meet these needs, a large amount of computer hardware has been designed and constructed specifically for use in the laboratory. A problem inherent to the effective use of this equipment, however, is the fact that it has generally been developed for highly specific lab uses, and has either tried to cope with existing high level languages or has abandoned the attempt and required the user to program in a low level assembly or machine language.

Our idea was to study a variety of laboratory computer applications currently in use around campus, and to use this information to design, develop and implement a programming language that is suited to the needs of a lab scientist. It would be developed to facilitate the description of the functions needed for effective laboratory interfacing at a high level and in a natural and transparent manner.

Our results have led us to believe that the best way to achieve this was using an interpretive/compiled programming environment (similar in spirit to FORTH) in which large programs could be built in small coherent pieces, which could easily be tested on as high or low a level as the programmer desired. Havoc adheres to these principles, while providing many of the more widespread and useful language features not found in

FORTH.

PROJECT BACKGROUND

Havoc is, strictly speaking, not a programming lanuage in the conventional sense, but rather an integrated, semi-interpretive programming environment. In its most elementary form, it is a command interpreter, not unlike a lisp interpreter that is capable of storing function definitions and implementing basic flow of control. The system is 'fleshed-out' by means of a number of basic assembly language primitives such as 'read from memory', 'add long int', and 'write to user console' that are used as building blocks for more extensive and complex functions. These primitives form a basic I-code which can be directly translated into the actual assembly language of a given host machine.

These primitives are by no means limited by the predefined instructions; implicit in the design of the system is the mechanism that allows the custom tailoring of the so-called builtin primitives and permits the system to make use of any user defined primitives.

For example, with a basic set of long word math primitives, Havoc acts as a very powerful programmable calculator. Adding file and I/O primitives extends it into a full and powerful programming language. Havoc's architecture also supports the use of typing to make it a high-level structured programming tool.

This document describes the approach, implementation, future plans, aspirations, reflections, and random thoughts of the authors with respect to the Havoc project. References to the language refer, in general, to the basic system and operators defined in the User's Guide to Havoc.

The original intent in designing Havoc was to fill a gap between software and hardware in a physical science laboratory. There is a need for a language that is flexible enough to accomodate the particular quirks of a data aquisition system yet high-level enough so that algorithms can be formulated and expressed in terms of functionally clear instructions as

opposed to the

```
'WRITE($F7) =  READ($E4, 0);   /* get user input */
```

common among laboratory oriented languages.

Such a language should have the ability to perform in an interactive environment. Progress in programming seems to correlate directly to feedback from a computer system. An interpretive system is ideal for this; a strictly interpretive system, however, is sadly lacking in one additional aspect that is essential for a laboratory oriented language: speed.

What is optimal is a semi-interpretive system that handles both incremental compilation and execution. Low-level specialized routines can be built and tested. When satisfied with the performance of these routines, the user may incorporate them into a higher level routine. Depending on how time or space critical the program is, these primitives could either be called as subroutines or inserted inline as macros. The resulting routine would be compiled and stored for future use. It could be tested to the user's satisfaction and incorporated into an even higher level routine where needed.

The code produced by such a system would then satisfy the three major criteria previously described: it would allow high-level access to low level functions, it could be built in an interpretive environment, and the resulting code would be executed in compiled form for maximum speed.

HOW WE GOT STARTED (MOTIVATIONS)

Having both taken the same laboratory interfacing course in the Fall Term (Science 40), we realized that the level of communication between computer and laboratory equipment was unnecessarily low-level. There was no lack of hardware for the computer to control, nor any lack of computing power to control the equipment; the problem was somewhere in the interface between the two.

First we looked at the problem from a functional standpoint. We decided that there was a need for an evaluation of what purpose a

computer interface served in a scientific laboratory. Data transfer, storage, display and device control, and timing emerged as the most prominent uses.

We then began to examine what approaches had already been taken toward fulfilling the the need. One thing that we noticed was that most tools (hardware and software) had been taken out of their normal domain where they served other uses and had been slightly redesigned to perform a slightly different function in the lab. In assembling a system for laboratory use, the scientist was typically required to jerry-rig bits and pieces of programs and boards.

Our aim became the design of a system designed with this end use in mind. One of us had developed a keen interest in the theory and implementation of programming languages, while the other had been interested in the idea of real-time computer/real-world interactions.

WHAT WE STARTED WITH

Our thesis would consist of designing, and implementing as far as possible, a specialized programming language/environment for doing real-world interfacing. Although we had originally envisioned the project as including specialized hardware, the lack of time, experience, and money forced us to scuttle our soldering plans. Instead we tried to come up with a series of minimum requirements, that such a system should involve. We came up with "the ideal" solution for each problem and then sought out a tool or approach that best met or approximated it. To avoid digging ourselves too deeply into the task of language design, we decided to adopt the strategy put forth (no pun intended) by Ellis Horowitz in his Fundamentals of Programming Languages -- that the task of the designer of a programming language is primarily to consolidate, and not to innovate.

A brief study was made of a number of different existing languages, and we investigated both the practical and theoretical aspects of their various features and design philosophies. A consolidation of our ideals produced an imaginary language that most nearly resembled FORTH. The user is given enough power to accomplish what he deems necessary; he may certainly staple himself to the wall with this power if he is not careful. The language is intelligent enough, however, to take care of many of the powerful and dangerous features if the user does not want to be

bothered by them.

WHAT WE DESIGNED

The result of our design quest yielded a language description that was theoretically very pretty, and was extremely self-consistent in both semantics and syntax, but had a number of practical difficulties. Most of these problems stemmed from transplanted features being "rejected" by the recipient language. For example, a compiled procedural program has all type checking done at compile time. After compilation, all its input comes in a controlled form, via typed input statements. A stack based language may be called with any type of data on the stack. Facilities are provided for checking stack data type, but none of this can be determined at compile-time; it must be carried around by the program.

In relation to FORTH, major differences include the addition of data typing, the use of local (automatic) variables, user-definable software interrupts and a more structured approach to program specification (line input vs. command input, words vs. directives). The last of these is part of an effort to counteract the criticism that FORTH often receives about being a "write-only" language.

WHAT WE HAVE DONE SO FAR

We currently have a system running Havoc with local variables and a simple set non-typed primitives. These are roughly equivalent to the basic Forth arithmetic and stack manipulation functions. The user dictionary can load procedure and variable definitions from a file or enter them from a keyboard and call them as needed. The short-range additions to the primitives include:

1) Interrupt definitions.
2) Simple typing

Long range additions include:

1) Extensive typing. This will be accomplished using a separate type stack, which holds a type descriptor for each corresponding stack item on the user stack.

2) Macintosh toolbox access

3) Integrated editor. The idea is to base it after the MacFORTH implementation. This will allow the capability for making instantaneous changes and additions, which will help decrease program development time.

4) "Device" implementation

5) Records

THE BASIS OF HAVOC

The problems of laboratory interfacing seem to arise from the fact that a manufacturer of software or hardware cannot anticipate all the uses to which their product will be put. The best they can hope for is to design a tools powerful enough to accomplish certain specific tasks and yet be general and flexible enough to allow the 'bending' of their product to unanticipated uses.

The following are several brief examples of computer interfacing setups with which we are familiar. From these, common recurrent problems are examined and used as criteria for the design of the language.

- A storage scope is used to record sound waves picked up by a microphone. The oscilliscope is attached to microcomputer that allows the user to specify in which mode the scope should operate and permits him to save and retrieve curves using a modem connection and a disk drive. [see figure 1]

- A microcomputer is required to control and read from a series of electric motors and sensors attached to the stage of a microscope for the purpose of allowing a scientist to note a position while scanning a slide and return to it.

- A microcomputer acts as the front-end of a multiboard laboratory computer setup, controlling access to the bus or, at a slightly greater removal, controlling a one board computer that controls the bus.

These situations are ones that were encountered or postulated in Science 40, a laboratory interfacing course.

The first was an lab exercise using New England Digital minicomputers running XPL. From the software end, the main problem was the lack of a high level language interface between the computer and the oscilliscope. Timing had to be done by means of successive reads and writes of seemingly random numbers to the bus. This had the effect of activating an external timer which waited the desired length of time in a

'hung' state before returning.

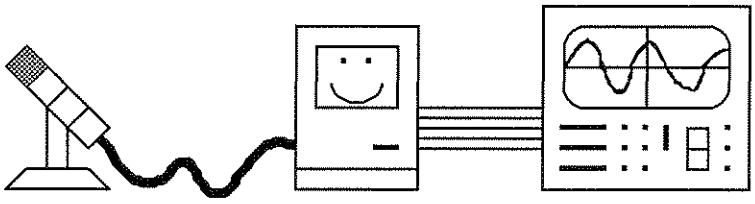
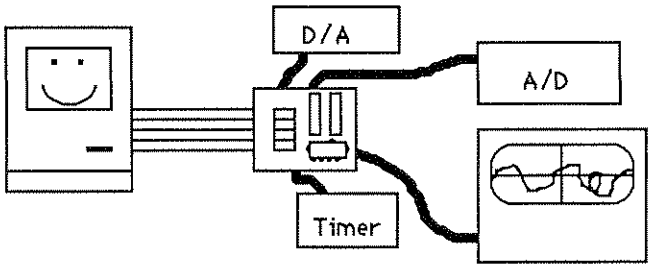


figure 1

The second experiment was postulated as a project when the physics department acquired a set of electronic calipers with a digital readout and serial output port. A major problem with implementing such a system is the need to control and read several possibly asynchronous ports while still giving the user access to system time when he needs it.

The third situation, as proposed by Professor Elisha Huggins, reflects a desire to have a high-level setup that takes advantage of the user-friendliness of a desktop, menu-driven environment available for students in a physics lab [see figure 2].

The problems with implementing this type of system are the most profound. Too rigid an environment can rob the equipment of its capabilities, restricting the user to circuitous workarounds to perform an experiment that was not forseen by the designers. If there is not enough access to the low-level aspects of the machine, the user will be unable to use the power he needs; however, if too much of the computer's bare bones are visible the user will spend all his time and effort trying to keep track of all the details.



[figure 2]

An additional requirement of such a system is speed. With massive amounts of data flying back and forth to keep the master computer and the user informed, the system must be able to cope with the blocks of data it needs to record from the acquisition devices as well as the control codes it needs to send out to specify what information it requires next.

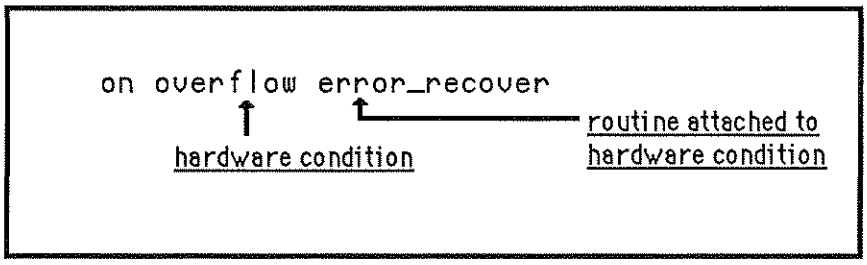
The needs that we eventually decided upon for havoc can be broken down into a series of abstract ideals that are implemented by one of a number of possible realistic representations of that ideal. The ideal language for our purposes would support event-driven actions, and would have the ability to represent complex data structures and device interactions in a manner that allows the user to ignore physical representations of the object in question. Furthermore, it would be supported by a dynamic environment that allows fast turnaround time for experimenting and debugging. Such a language must be high-level enough to allow construction of programs that represent structured flow of control in a clean and transparent manner, yet one that accurately reflects the way the machine actually works.

The first of these ideas is the simplest to concisely define and has the nearest realistic implementation. In PL/1, various 'conditions' of the machine are defined. Some of these are hardware conditions such as 'overflow', 'zerodivide' and 'external interrupt'. The user is allowed to define additional conditions, without any physical representation, such as 'job_done' or 'need_input'.

Blocks of code or procedures may be attached, locally or globally, to the occurrance of any of these conditions. When the processor detects a condition, it transfers control to the specified code. Similarly, if the user decides to, he or she may tell the processor explicitly that one of their conditions has happenned and let the processor act accordingly.

The need that these conditions fulfill in laboratory interfacing is one of handling the real world at the speed at which it happens. An action, such as the gathering of a data point, or changing an output voltage may have to occur on a regular basis, depending on a clock interrupt. A corrective action may need to be taken or a calculation aborted if a zero divide or overflow is detected. A user may want to take corrective action or perform a task and return to the original calculation in progress, or the

interrupt may tell him that the calculation has been corrupted and should be aborted.



simple exception-handling statement

In order to support these actions, then, the language should have a construct that specifies a condition as either a predefined hardware state or as a special user defined state. The user must be able specify a procedure to be executed when the given condition is met. Since the user defined conditions will never be met by the hardware, the ability to 'signal' a given condition must be included. By default, it seems that the best convention, and coincidentally, the easiest to implement, would be to have the interrupt routine return to where was signalled from after completion. A provision must be made, then, for breaking out of a calculation and returning control to some other part of the program.

Additionally, there are times, such as within an interrupt procedure, that one wishes to disable the interrupt handling. So there must be a provision for disabling and re-enabling the interrputs.

The second of of the ideals, abstract representation of data structures and devices, presents a much more complex problem. By dividing this category into two subparts, data and devices, we can come up with individual solutions for each that can satisfy our need.

Data typing has long been a method that allows convenient abstraction of physical memory. A variable is declared to be of a given type and then may be referenced by operators that pertain to that type. The string "rabbit food" may be concatenated or indexed by a simple command, but we are warned that there is no logical way to subtract it from the the integer '42'. Similarly, typing allows overloading of an operation. When an operation makes sense, such as adding the integer '1' to the floating point value '6.02e23', the user does not have to remember that the former is

stored a 16-bit two's complement quantity while the other is an 80-bit excess16K exponent with a hidden bit mantissa. The system checks the types of each of the operands and chooses the correct addition operation for them. This not only makes for compact, more readable programs, but reduces the chance of programmer error.

A problem often associated with data typing, however, is that a language designer, when specifying legal and illegal combinations of operands can rarely take into account all reasonable combinations that a programmer might want to use. As a result, the user should be allowed to override the default by telling the compiler "trust me, pretend that this is an integer and we won't have any problems".

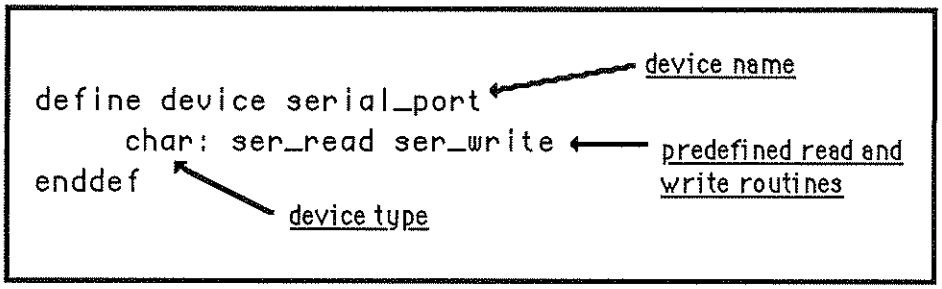
A common complaint about Pascal is the inability to perform operations that, though logically compatible, suffer from excessive typing restriction. This stems from the theoretical nature of the language requiring the user to describe his data in a precise abstract manner. Our decision was to adopt a more lenient, C-style approach to typing. In Pascal, finding the difference between the ASCII codes of an 'a' and an 'A' requires conversions from char type to byte type for each letter before the operation, while C lets the user simply subtract them.

Devices represent a less standardized area of abstraction. We want a high-level way of saying "start timer", "stop timer" or "read serial port". Simula has the notion of a class with procedures defined on it. An arbitrary function defined as part of the class may be called with a specific instance of the class to perform whatever actions necessary on it. A similar result may be obtained, it seems, by defining a "device", as we call it, and its constituent parts as having whatever data types it can be accessed by.

For example, a timer would be a data record with two devices as constituent parts: the current integer value of the timer and a boolean flag whether or not the timer was running. The timer could then be accessed merely by reads and writes to the parts of the device. To stop the timer, the user would write a 'FALSE' to the timer_running part of the device.

There must be a provision, then for defining what actions must be taken when the device, or its component parts are read from or written to. Each component of the device must either have a defined read or write operation on it or an amount of storage that is associated with the default

variable read and write routines.



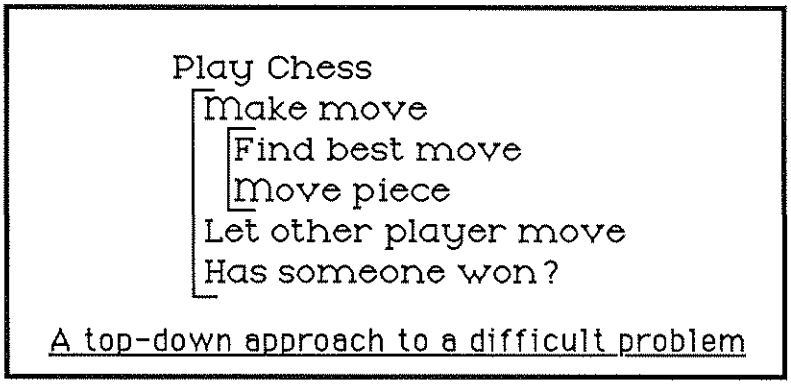
The final feature that we outlined for havoc is by far the most ephemeral and subjective. We need to choose a language structure and environment that is best suited to laboratory use. We realized that the language structure should promote the building of well defined, functionally integral units that could be bullet-proofed individually and easily integrated into a larger, equally sound unit that performs a more complex task.

From personal experience and conversations with fellow students we realized that an essential part of laboratory computer interfacing is the "hands on" aspect that is generally looked down upon in more theoretical circles of computing. Due to the innumerable unseen factors that arise in a laboratory situation, a remote, full system compilation and testing setup can make the turnaround time almost unbearable. This is where Forth has made a significant step forward. By allowing incremental compilation, using small, debugged building blocks to build and debug larger blocks, progress towards a correctly functioning system can be hastened significantly.

A difficulty this approach is that it conflicts with the concept of top-down construction. A programmer's task is made simpler if he is able to start at the highest level and divide the task at hand into a number of smaller, more precisely defined tasks. In building-block languages such as LISP and FORTH, the user is required to precisely define his lowest level routines before he can call them in a higher level, more abstract routine.

A specialized programming environment can aid a programmer to overcome the top-down conflict in a building block language (**Software P & E, Oct 84, p921**). Additionally, we realized from actual experience in the lab that the vast majority of the time spent in an interfacing problem

was at the low level. It was here that the workings of the system were the least well defined and required the most experimental tinkering. For this reason, we chose to adopt the bottom up building block approach for a language environment.



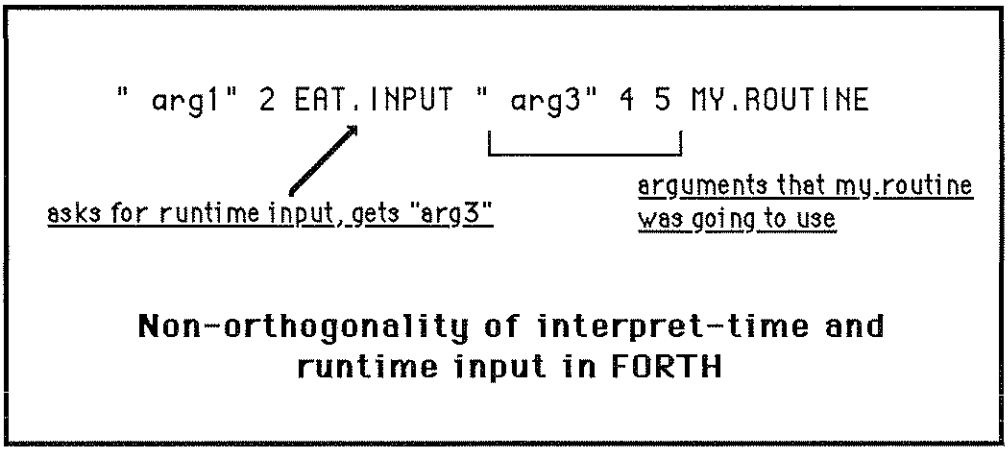
For the overall program structure, we should choose a format that allows maximum flexibility to the programmer. For this criterion, the approach taken by FORTH, seems to be the best. The programmer is given free reign to code in almost a "stream of consciousness" manner, choosing what he or she wants to do next without a restricting command format. An instruction should be expressed in the smallest atomic unit necessary to complete an action. More complex instructions could then be built up for convienience, but the lack of original constraints would allow the programmer a degree of freedom similar to that found only in assembly language.

Certain formats, however, must be constrained. To prevent "write-only" coding and make creating a compiler a realistic task, control structures, such as begin-end, case and if statements must have a structured format rigid enough to form a skeleton to support the rest of the code.

As for the actual constructs that the language should use, we should aim for absolute economy. If a specialized tool is needed by a programmer, we should give him the pieces with which to build such a tool rather than incorporating it as part of the language as a whole. The simpler the language is, the more easily the user can master it.

Design Specifics

For exact program representation, we examined the language that came closest to out ideal: FORTH. Here we had a 'stream of input' based language, but one that allowed unnecessarily convoluted programming style. A programmer may create a procedure which asks for input from the terminal. This may be constructed in such a way that the input stream containing the call is used as the source of input. As a result just looking at the call of a word is not enough; one must understand all the components of a line of code completely.



Non-orthogonality of interpret-time and runtime input in FORTH

Havoc divides all input into two classes: command stream input and runtime input. It parses and executes the command stream input. If a word in this stream requires input, havoc pauses to get input from the user. Compile time and runtime remain orthogonal resulting in a simpler system.

The words in the command stream input are further divided into two classes. Procedural words are calls to defined dictionary entries and act, essentially only on any input they get from the terminal and items already pushed onto the stack. Directive words, in contrast, are signals to the compiler, indicating that the words to follow are to be processed in a special way. A simple example of a directive word is a procedure definition. The word 'define' indicates that the words that follow, up until the 'enddef' are to be entered as part of a definition. The two main compile-time directives in havoc are 'define', entering a definition in the dictionary, and 'on', assigning a procedure to a given condition.

Flow of control is also handled by means of directive words. The

structure 'iftrue then < ... > endif' specifies that at run-time, the item on top of the stack is tested. If it is true then the commands contained in the directive are executed, otherwise they are bypassed.

Procedures in havoc are all external and obey static scoping limitations. Any procedure word can invoke itself and any other procedure and can similarly access external (global) variables. A procedure also has free reign on the user stack; it may push or pop any data necessary and need not (and usually will not), leave it in the same condition as it found the stack in.

The external nature of procedures gives rise to a number of consequences for scope of variables: any variable that is defined outside of a procedure is global and can be accessed and changed by any subsequent procedure. Any variable that is defined inside a procedure is local to and exists only within that procedure. Consider the following example:

```
    define variable fred_1 long   (* global vars. *)
                    fred_2 long
    enddef

    define procedure add_to
    variable fred_1 long      (* set up a local variable *)
    begin
        fred_1 inc                          (* increments the local
*)
        fred_2 inc              (* no local, so increments global
*)
    enddef

    fred_1 inc                  (* increments the global variable
*)
```

Variables are bound at compile time to the innermost definition of the identifier. Since there are only two levels of variable definition, global and local, a variable will either be bound to a local definition, if it exists, or a global definition.

The extent of global variables is global. Once defined, they exist until the havoc system is exited or until they are specifically removed from the dictionary.

Variable typing, ***though not yet fully implemented*** follows the example of C. Most variables can be handled in terms of the commonly used integer, character, pointer types etc., so these are made base types. When required, though, a more complicated type may be defined. This may be done in terms of a specific array or size of a type or may be used to add levels of abstraction to the data. For data types that require more than one field, a record may be defined made up of individual fields, each of which may be a basic type or a more complex, previously defined type.

```
define type
    curve integer[0:256] (* an  array of integers *)
    sample record          (* a more complex type *)
        title string[0:32]
        time  integer
        data  curve
        ;
enddef
```

Sample type definition

A type may either be defined in the dictionary and given a name in a type definition or it may be anonymous, being described in a variable definition and attached to the variable it describes.

Type equivalence is handled in what may be for a Pascal programmer, a lax manner. An operation between variable is examined for types. If the operation has a valid logical action for variables of the given types, then that action is taken. For example, subtracting an integer from a character should yield a result of the less constrained type, in this case, an integer.

Project Review and Future Considerations

As with any tool, the utility of a newly designed language cannot be determined until it is actually put to use in solving problems. The shortcomings in the original design can be detected by working with it, as can the places where the design has succeeded. Havoc, in its current

implementation, as well as in its original specified design, has several notable hits and misses on the criteria specified at the start of the project.

We will first describe incidental results from the language design, planned into havoc, but included more as a consideration of the shortcomings of other languages, than as a specific tool for laboratory interfacing.

Use of local variables is a feature that yielded very positive results. The number of calculations that can be simplified by using local storage instead of having to juggle the user stack exceeded our expectations. An example of this simplification is the implementation of the 'dups' and 'swaps' string words defined in the file 'Startup.hvc'. By allocating local storage, a simple, efficient and intuitive method can be used to manipulate large strings on the user stack, making possible the definition of functions that would be difficult or impossible to implement efficiently in an environment without them.

```
define procedure swaps
variable
  temp_str1 string[0:132] (temporary string for swap)
  temp_str2 string[0:132]
begin
   temp_str1 writes (save a copy of each string)
   temp_str2 writes

   temp_str1 @s  (restore them in opposite order)
   temp_str2 @s

enddef (of dups)
```

Allowing direct recursion is actually an offshoot of the implementation of local variables, following the need that a word must be useable as soon as there is enough information about it to use it. FORTH requires that a definition be completed and installed before it may be referenced, however, since a local variable is never permanently installed in the dictionary, this convention would prevent it ever being used. A procedure, then, as well as a variable, may be called as soon as adequate information exists as to its nature.

The benefits of allowing recursion are more evident in non-laboratory computing than in the lab. While few interfacing situations require the feature, many mathematical problems may be simply expressed by use of direct recursion. An example of this is the Pascal's Triangle program defined in the file "Pascal.hvc".

The features designed specifically for aiding laboratory computing met with mixed results. In most cases, the actual performance of the code could not be tested due to the as yet incomplete implementation of the havoc compiler. In these cases, the solution must be tested by virtual means, writing a test program and manually evaluating how it would perform.

Although interrupts have not been actually implemented writing code using them may still be tested on 'vaporware' for ease of coding and understanding. Here, the results were suprising, meeting the specified criteria in some areas, failing in another, and additionally providing an unforeseen new possibility for programming style.

The hardware conditions proved valuable, as expected, in trapping errors. Their greatest utility appears to lie in either alerting the user to some event and taking minor corrective action or detecting some fatal error and aborting execution. A feature that would greatly extend their usefulness would be the ability to perform some degree of non-local goto, short of returning to parser control. As it currently stands, there is no way, without a convoluted system of function returns and conditional structures, to use interrputs to have a program gracefully abort part of a computation, such as the grabbing of a specific curve, without stopping the entire program. Although in a purely structured language this breaking across structure barriers should not be allowed, it is too much a useful and necessary part of laboratory computing to be left out.

Software interrupts at first seemed to be redundant when we bagan writing programs with user defined conditions. If we could 'signal' an interrupt attached to a word, then there seemed to be no reason for not just calling that word. The effect is merely one of aliasing the word with the name of the condition.

The difference lies, however in the binding of the call. Once compiled, a word is bound into the definition and cannot be altered. A condition, however, may be disabled, re-enabled and reattached to a different word,

even during the execution of the program! While this presents the possibility for writing a sort of high-level, self modifying code, it restricts it to a set of predefined cases.

```
define procedure get_input  (get the user's answer)
begin
  "Please input answer: " pops
  signal get_answer
enddef

define procedure input_routine  (choose which input routine)
begin
   key                          (if there's a key down, they)
   iftrue then                  (want keyboard input)
     on get_answer get_keyinput
   else
     on get_answer get_mouse_input
   enddef
enddef
```

In effect, the user, if he takes the proper steps beforehand, is allowed to re-bind certain words to new meanings at both compile time **and** at runtime. This bypasses a problem found in FORTH and some other threaded languages of not being able to 'tweak' procedures once compiled. A low-level routine may be debugged by means of attaching various trial routines to a signal in the high-level routine. Although this should probably not be reccomended at standard programming technique, it does present a viable debugging strategy.

Data typing and device specification were partially implemented and yielded good results in all cases. Type recognition was not implemented, so no programs could be tested that relied on type overloading to operate correctly. Instead, each individual typed routines was left exposed to the user, requiring him to choose for himself which was the correct operation for the arguments on the stack.

For file and terminal I/O, these patch routines were still sufficient to allow effective control. Although the presence of type descriptors before each I/0 call clutters the appearance of the program somewhat, we feel that the approach taken has considerably simplified the coding of interfacing routines without unnecessarily constraining the user.

One place where the design fell short of expectation was in the

handling of file and port I/O control. Although the 'control' word is specified as passing the necessary control information on to the operating system, it still leaves the nature of the control information itself unnecessarily vague. At this level, to specify an operation such as a file reset or a baud rate change, the format of the control information was left unclear, requiring the definition of arbitrary constants much like the ones we were hoping to avoid by this design.

The nature of the file system and serial ports and the diverse operations that a user may want to perform on them still present a problem, then, that must be addressed at a low level. The only apparent solution is one of modifying the compiler to include a veritable battery of predefined functions to help the user build the control information he desires at a high level.

The input format of the environment proved very supportive for the form of programming that we anticipated needing for a laboratory environment. This style of "type in a line and see if it works" programming seems to be the ideal form of environment for laboratory. The growing popularity of languages that use this form of interaction in laboratory situations further supports this hypothesis.

One deviation from the standard FORTH-style user interface, adds additional friendliness to the environment. The allowing of any form of atomically executable code to be tested before being enclosed in a definition extends the utility of incremental compilation in a logical manner. Specifically, in FORTH, any control structure, such as a conditional or a looping statement, must be enclosed in a definition. This unnecessarily restricts the user by preventing him from "test driving" them from the interpreter. The ability to execute control structures outside of definitions has proven to be a significant benefit when trying to formulate definitions that must perform the same actions.

Conclusions

The havoc system has yet to be extensively tested in the laboratory; in fact, without several additions such as a functioning 'control' statement, it may not yet be testable. It has, however shed light on many problems encountered in the design of a programming language and their solutions.

The original design considerations can in fact be met, but in some cases, such as the development of an effective set of 'control' information constants, the cost of meeting the specifications may outweigh the benefits anticipated in the original design.

Havoc should probably be seen as a steppingstone between the first interpretive-compiler languages and later, more sophisticated ones incorporating that advantages of earlier ones while avoiding their shortcomings.

Appendix A:
BUILTIN FUNCTIONS

The following is a list of builtin functions that exist as of havoc version 0.85. To be added with later versions are type words such as "TYPE?" and "SIZE?" and constants to handle specific I/O, such as to serial ports and the screen.

"POP" (* n -- *) Pop the top number off the stack to the terminal.

"POPLN" (* n -- *) Pop the top number off the stack to the terminal, but add a carriage return afterward.

"POPS" (* s -- *) Pop the string on the stack to the terminal.

"CR" (* -- *) Print a carriage return to the screen.

"KEY" (* -- b *) Pushes a TRUE if there is keyboard input pending, otherwise false

"GET" (* t -- n *) Pop the type descriptor off the stack and get input of that type from the terminal.

"PUT" (* n\t -- *) Pop the type descriptor from the TOS and pop the next item on the stack in that format.

"FGET" (* t\f -- n\b*) Same as GET, but first pop a file descriptor and get from that file. Leaves a boolean success value on TOS.

"FPUT" (* n\t\f -- b*) Same as PUT, but first pop a file descriptor and get from that file. Leaves a boolean success value on TOS.

"+" (* n1\n2 -- n1+n2 *) Adds top two numbers on stack.

"-" (* n1\n2 -- n1-n2 *) Subtracts top two numbers on stack.

"*" (* n1\n2 -- n1*n2 *) Multiplies top two numbers on stack.

"/" (* n1\n2 -- n1/n2 *) Divides top two numbers on stack.

"MOD" (* n1\n2 -- n1 mod n2 *) Takes modulus of n1 by n2.

"NEG" (* n -- (-n) *) Negates number on stack.

"NOT" (* n -- n' *) Takes one's complement of n.

"INC" (* v -- *) Increments variable on top of stack.

"DEC" (* v -- *) Decrements variable on top of stack.

"LOAD" (* s -- *) Switches input to read from file with name on TOS. If

"OPEN" (* s1\s2\v -- *) Opens a file of name s1 with permissions s2 ("read", "write", or "append") and stores the file pointer in variable v.

"CLOSE" (* v -- *) Complement of open. Closes the file that has file pointer stored in v.

"CONTROL" (* n\v -- *) Writes control information n to the file or port with file pointer in v.

"FIRST_ENTRY" (* -- v *) Global variable. Address of first dictionary entry.

"DICT_START" (* -- v *) Global variable. Address of start of dictionary.

"DICT_TOP" (* -- v *) Global variable. Address of top of dictionary.

"RETURN" (* -- *) Control word. Returns from the current procedure.

"EXIT" (* -- *) Control word. Exits the innermost "DO" loop.

"BREAK" (* -- *) Control word. Breaks out of innnermost control

structure.

"QUIT" (* -- *) Quits the havoc system and returns to the finder.

"ABORT" (* -- *) Control word. Resets the user stack, returns to outer level of parser.

"@" (* v -- n *) Dereferencing operator. Gets contents n of variable v and leaves on top of stack.

"WRITE" (* n\v -- *) Writes number n into variable v.

"DUP" (* n -- n\n *) Duplicates top item on stack.

"DROP" (* n -- *) Discards top number from stack.

"DROPS" (* s -- *) Discards top string from stack.

"OVER" (* n1\n2 -- n1\n2\n1 *) Copies second item on stack to top.

"SWAP" (* n1\n2 -- n2\n1 *) Swaps top two numbers on stack.

"COPY" (* n -- nth item *) Copies nth item from top of stack (not counting the number popped), to TOS.

"=" (* n1\n2 -- b *) Pops top two numbers on stack, returns 1 if they match condition, otherwise 0.

"<>"
">"
"<"

"AND" (* n1\n2 -- b *) Logical AND of top two stack items.

"OR" (* n1\n2 -- b *) Logical OR of top two stack items.

"DUMP" (* -- *) Special compiler debugging words that are extracted

"DUMPSTACK" at parse-time and are executed then. they allow a peek at the workbench and user stack.

BIBLIOGRAPHY

Brian Kernighan & Rob Pike, The UNIX Programming Environment Prentice-Hall, Inc., Englewood Cliffs 1984

P. Pepper (editor), Program Transformation and Programming Environments Springer-Verlag, New York 1984

Thomas McIntire, Software Interpreters for Microcomputers, John Wiley & Sons, New York 1978

Ellis Horowitz, Fundamentals of Programming Languages, Computer Science Press, Rockville 1984

Leo Brodie, Starting FORTH, Forth Inc.

C.A.R. Hoare, Hints on Programming Language Design, Stanford Artificial Intelligence Laboratory Memo AIM 224, Computer Science Dept. Report No. CS-403, Oct 1973

David Patek & Willis Tompkins, A Fast Microcomputer Language for Signal Acquisition, Processing and Display, *Computer Programs in Biomedicine* 12, 1980, pp230-242

Charles Wetherell, Array Processing Languages, *Software Practice & Experience,* April 1980, pp265-

Philip Leith, Top-Down Design Within a Functional Environment, *Software Practice and Experience,* Oct 1984, pp921

Gordon Bull & Alan Lewis, Real-Time BASIC, *Software Practice & Experience,* Nov 1983, pp1075

Gary Feierbach, FORTH, The Language of Machine Independence, *Computer Design,* June 1981, pp117

Plus personal conversations with: Elisha Huggins, Mark Sherman, James Baumgartener, David Levine, Carol Fowler and the HyperDrive in room 2-3.

HAVOC DEVELOPMENT PROJECT

USER'S GUIDE TO HAVOC

Modification History:	First Draft	(really havoc) (SM) dark ages
	Second Draft	(havoc 0.2) (SM) 21 Feb 85
	Third Draft	(havoc 0.75) (SM) 15 May 85
	Forth Draft	(havoc 0.80) (DC) 26 May 85
	Fifth Draft	(havoc 0.80) (DC) 28 May 85
	Sixth Draft	(havoc 0.8532) (SM) 3 Jun 85

Significant changes since last revision:

Descriptions of file, device and record handling corrected, previous omissions (DROP) added.

CHAPTER I

BASIC DATA TYPES

NUMERIC TYPES

Numbers may be either integral or fractional ***,although floating point is currently not supported, fixed-point arithmetic will be available in havoc version 2. For the Macintosh, Toolbox access will probably be provided into the SANE (floating point manipulation) and ELEMS (transcendental functions) packages for this purpose***. Havoc supports numeric data in three bases, and uses the following notation:

Hexadecimal (base 16) numbers consist of a hexadecimal number preceded by a dollar sign, such as $530A or $4E71. ***Note that hexadecimal numbers are not supported in the current implementation of havoc.***

Octal (base 8) numbers consist of an octal number preceded by a backslash, such as \377 or \2726. ***Note that octal numbers are not supported in the current implementation of havoc.***

Decimal (base 10) are standard, run-of-the-mill numbers. Any digits not preceded by either a dollar sign or a backslash are interpreted to be in base 10.

CHARACTER STRINGS

A string is a sequence of characters enclosed by quotation marks. To (include) a quotation as part of a string, use a double quotation mark.
Examples:

"My porcupine is feeling bloated today."
"Please answer ""YES"" or ""MAYBE"" at this time"

CHAPTER II

COMMENTS

No program should be without them. Any text that appears between opening and closing parentheses is ignored by the compiler. Nested comments are allowed.

There is still some squabbling amongst the authors concerning the appearance of comments in havoc. Although the lexical analyzer in havoc 0.8 regards anything between opening and closing parentheses as a comment, the example programs in this manual use the "(*" and "*)" notation, which is entirely compatible with the parentheses.

CHAPTER III

STACK ITEMS

Stack items constitute the fundamental data elements in havoc. Stack items are pushed on the stack by entering their name directly into the input stream.

example: Let's assume that we want to add two numbers, say, five and six. To do this in havoc, we need to first push the two numbers onto the stack, and then invoke the addition operator, which adds the two topmost numbers on the stack, and leaves the sum on the top of the stack. Our command to accomplish this would look like `5 6 +`
The occurrence of the 5 pushes a 5 onto the stack, similarly for the six, and the + adds the two and pushes the result onto the top of the stack.

Strings, arrays, variables, and records are all considered stack items, and can thus be manipulated in a similar fashion (provided, of course, that the operations occur on compatible types). Therefore, an occurrence of the string "Hello there" will push it onto the stack.

CHAPTER 10
THE INFIX PREPROCESSOR

There is NOOOO infix preprocessor. Probably never will be. Forget you ever heard about it.

CHAPTER 11

WORDS

Like FORTH, havoc is a word-based language. Words constitute the fundamental modular element of a program. They are kept in a 'dictionary' in memory that tells havoc what to do when it encounters that word. A word may typically push a value onto the stack, perform an operation (such as an add or multiply of items on the stack), or execute an entire program.

Havoc words may be divided into two distinct classes: procedural words and directive words. Procedural words operate on the user stack. Directive words, on the other hand, tell havoc that action is to be taken based on the words following (either procedural or directive) until the end of the directive is detected.

The most common example of a directive word is the definition of a dictionary entry, while a typical procedural word is one which defines a data item. The associated procedure just tells havoc to fetch the address of the data item and place it on the stack.

PROCEDURAL WORDS

Procedural words perform a certain task, or evaluate a function when invoked. Procedural words are given access to the user stack, and they are free to read from it or alter it in any way. Procedural words are the equivalent of the procedures and functions of other high-level languages, like BASIC, PL/1, or Pascal.

> A word which represents a data item is a procedural word just like any other, except that the procedure is implicit, and says to just push that particular data item on the stack when invoked.

Parameters and Procedural Words.

Unlike the languages just mentioned, words do not explicitly pass or receive parameters. For example, a common PL/1 construct is

```
call FRED( x, y, z );
```

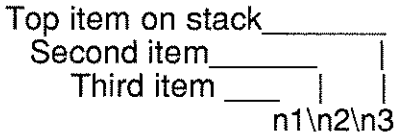
The effect of this statement is to call the procedure FRED, and to pass the variables x, y, and z as parameters. In havoc, when a word is invoked, it is always passed the user stack, which may be thought of as being in a certain state. The word operates on the stack, possibly changing the initial state, and implicitly returns the stack upon returning. Thus, to communicate with FRED, one would push the data onto the stack, and then invoke FRED. FRED would then start up and immediately look at the top of the stack for the data it needs. The example above, when translated into havoc, reads

```
x y z FRED
```

Basic Havoc Words and What They Do

Since most operations directly affect the stack, we have adopted the Forth-style notation to describe stack operations in shorthand.

The stack is represented by a series of items separated by backslashes to show their order on the stack.



An operation is usually represented by a comment, enclosed between a "(*" and a "*)" indicating the state of the stack before and after the operation. For example, "+", a word that pops the top two items on the stack and pushes their sum could be described as

(* m\n -- (m+n) *).

Usually, only the portion of the stack affected by an operation is included in its description.

Arithmetic Operator words:

" + " (* m\n -- m + n *) (* m$\n$ -- m$ + n$ *)
Addition operator. Adds the top two elements on the stack and replaces them with the sum.
If m and n are both strings, then they are removed from the stack, and the concatenation of the two strings becomes the new TOS.
String concatenation will be implemented in havoc version 1.0

" - " (* n1\n2 -- n1 - n2 *)
Subtraction operator. Subtracts the TOS from the item underneath it, pops them off, and then pushes the result onto the TOS.

" * " (* n1\n2 -- n1 * n2 *)
Multiplicative operator. Multiplies the top two items on the stack and replaces them with their product.

" / " (* n1\n2 -- n1/n2 *) Division operator. Divides the TOS into the item next on the stack, and replaces them with the result.

" MOD " (* n1\n2 -- n1 mod n2 *)
Modulo operator. Removes the top two items from the stack, and pushes the remainder of n1 / n2 onto the TOS.

NEG (* n -- -(n) *) Negation operator.
Negates the TOS. Works on singular and aggregate numeric types.

SGN (* n -- sign(n) *) Signum function operator.
Replaces TOS with it's algebraic sign;
1 if TOS is greater than zero
0 if TOS is equal to zero
-1 if TOS is less than zero
SGN will be implemented in havoc version 1.0

INC (* var -- *)
Increment operator. Reassigns the variable on the TOS to be the old value of the variable plus one. Generates an error if the TOS is not a defined variable. This is a shorthand notation, and is equivalient to the following havoc code:
"SUM INC" is the same as "1 SUM @ + SUM WRITE"

Example:

```
(* The following program fragment computes the sum
     the first three positive integers. Equivalent
   lines of PL/1 code are included as comments next to
     the relevant instructions *)

  DEFINE
       VARIABLE sum INTEGER;    (* Added total *)
  ENDDEF

   0 sum WRITE          (* Initialize sum at zero *)
  sum INC             (* SUM = SUM + 1 *)

(* End example *)
```

DEC (* n \ var -- *)
Decrement operator. Same as the additive reassignment operator, but value one is subtracted from the top stack variable.

Logical Operator Words

NOT (* n -- ~n *) Complement operator.
Replaces TOS with its logical complement if of type BOOLEAN;
One's complement (bit by bit) otherwise.
Returns an error if TOS is of type CHAR.
*** Rules for aggregate types not yet decided ***

" = " (* n1\n2 -- TRUE or FALSE *) Equality operator.
Replaces the top two items on the stack with the boolean value TRUE if n1 = n2, and FALSE otherwise.

" <> " Inequality operator. Similar to the equality operator.

" >, <, <=, >= " Greater than, Less than, Greater than or equal to, and Less than or equal to, respectively. Similar to equality operator.

Data Storage and Retrieval

" @ " (* addr(n) -- n *) Read address. Reads the memory location pointed to by the TOS, and replaces it with the contents of that memory location.

example:

```
    (* This program fragment prints the value of the
   integer variable SUM on the terminal.  SUM has
already been created elsewhere in the program. *)

          SUM         (* Push address on stack *)
          @           (* Replace address with value *)
          POP         (* Print value on terminal *)
```

" @S " (* addr(s) -- s *) Read address. Reads the string in the memory location pointed to by TOS, and places it on the top of the stack.

WRITE (* n\addr(m) -- *) Write address. Writes the value n into the memory location pointed to by the TOS.

example:

```
(* Integer variable SUM has already been created *)
        3 4 +                   (* Add 3 and 4 *)
        SUM WRITE               (* Store sum in SUM *)
```

WRITES (* s\addr(m) -- *) Write string to address. Writes the string into the memory location pointed to by the TOS.

Basic Stack Operations

DUP (* n -- n\n *) (Pronounced "DUPE") Duplicates the TOS.Note: In the file "Startup.hvc" the word DUPS is defined for duplicating the string on the top of the stack.

ROT (* n -- stack(n) *) Moves the n^{th} item from the TOS and places it on top.
ROT will be implented in havoc version 1.0

COPY Works just like ROT, but doesn't extract the item from its original place on the stack.

SWAP (* n1\n2 -- n2\n1 *) Swaps the top two stack items. Note: In the file "Startup.hvc" the word SWAPS is defined for swapping two strings on the top of the stack.

DROP (* n -- *) Drops the top number on the stack off into oblivion.

DROPS (* s -- *) Drops the string on top of the stack off into oblivion.
***Note: This is a temporary instruction that will be incorporated into DROP once full typing is implemented.

(* topitem\i -- topitem[i] *) Subscript operator.
Legal only on string and array types. Removes TOPITEM and I from the stack, and places the character (element) TOPITEM[i]on the TOS. Causes an error if i is not within the size limits of the topitem.
This will be implented in havoc version 1.0

Flow of Control Words

"BEGIN" BLOCKS.

syntax:

```
<begin block> ::= BEGIN <command list> END
```

description:

When a BEGIN word is encountered, control transfers to the word immediately following, and continues until an END is encountered. BEGIN blocks allow you to lump several words into a single packet. As a common (and useful) example, consider the IFTRUE...THEN structure, which ordinarily lets you execute only a single word if the TOS contains a TRUE value. Using a BEGIN block allows you execute several words if a certain condition is true:

```
(* Variables ANSWER and RESPONSE have already been
    created *)

 ANSWER @ RESPONSE @   (* Push variables on the stack *)
=                       (* Make the comparison *)
IFTRUE THEN
    "That's correct.  Now let's see if you can glue the
     hourglass to the tablecloth."
    POPS                        (* Print the message *)
     difficulty INC  (* Increment difficulty *)
ELSE                    (* Wrong answer entered *)
     "My hovercraft is full of eels."  POPS
ENDIF
```

Control Structures

Although most words exist by themselves as complete and independent entities, some words in havoc only make sense when invoked with other words. This is the only way that modern flow control structures can be constructed. Although havoc is a relatively free-form language, we feel that structured programming techniques are necessary in order to construct reliable, large systems.

Control structures in havoc consist of a conditional execution construct, a loop construct, and a case selection mechanism.

```
<control structure> ::= <conditional structure> |
                        <iterative structure>   |
                        <begin block>

<conditional structure> ::= <iftrue structure> | <case structure>
```

-- Conditional Structures

The IFTRUE Structure

syntax:

```
<iftrue structure> ::=  IFTRUE THEN <command list>
                        <else clause>
                       ENDIF

<else clause> ::=  { } | <command list>
```

description:

The IFTRUE...THEN...ELSE construct allows conditional execution of a block of code. If more than a single word is to be executed, the words must bracketed between the words BEGIN and END. Conditional structures may be nested.

example:

```
(* The following program fragment is equivalent to the
     following statements of "conventional" code:
               if amount > maximum
               then call correct;
               else call update( amount );     *)

amount @               (* Push amount on the stack   *)
maximum @              (* Push maximum on the stack  *)
>                      (* Make the comparison        *)
IFTRUE THEN correct
 ELSE BEGIN amount update END
ENDIF
```

The CASE Structure

syntax:

```
<case structure> ::= CASE <case list>  <default clause>
                     ENDCASE

<case list> ::= { }  |  <case list>  |  <stack item>  :  <command>

<default clause> ::= { }  |  DEFAULT : <command>

<command> ::= <stack item> | <control structure>
```

description:

The CASE structure expects a number to be on the top of the stack, and looks for a match in the list of numbers following the CASE word. If a match is found, then control is transferred to the command immediately following the colon. If no match is found, then control transfers to the OTHERWISE clause. When the semicolon is found, control is transferred to the word immediately following the word ENDCASE.

example:

```
(* Find out where the user lives.  This fragment assumes
  that the string variable LOCALITY has already been
  defined *)

"Enter your zip code:" POPS
 INTEGER GET                (* Get code from the user *)
 CASE
     22003 : BEGIN          (* User lives in Annandale *)
                 "Annandale" locality WRITES
             END
     03755 : BEGIN
                 "Sorry about that." POPS    (* Too bad *)
                 "Hanover" locality WRITES (* Lives in NH *)
             END
   DEFAULT: BEGIN
                 "Unknown" locality WRITES
             END
 ENDCASE
```

The Iterative Structure

syntax:

```
<iterative structure> ::= DO <command list> LOOP
```

description:

The iterative structure repeats the <command> between the DO and LOOP words forever. The loop may terminated by using the EXIT word, or by pulling the plug on the computer you are using.

example:

```
(* Print the integers from 1 to 25.  Integer variable
    "i" has already been created. *)

DO
   BEGIN
   i INC              (* Increment loop index *)
   25 >                     (* Compare index to 25   *)
    IFTRUE THEN EXIT ENDIF    (* Break out of loop     *)
   i @ POP                   (* Print integer *)
   END
LOOP
```

BREAK (* -- *) Breaks out of the innermost control structure.

EXIT (* -- *) Exits the innermost DO loop.

RETURN (* -- *) Returns from the current procedure.

ABORT (* -- *) Aborts the executing procedure and returns control to the havoc parser. Effectively does a reset of the system.

I/O Words

POP (* n -- *) Pops the top item off the stack and prints it on the terminal. Won't work with items such as records.

***Implementation note: in the current implementation, data typing has not been added. POP, then, has no way of determining the size of the top stack item. For now, POP works for any data object that is a single long word in size, such as integers and characters. POPS has been added to pop off a string sitting on the stack until typing is implemented. ***

POPLN (* n -- *) Like POP, but prints a newline character after the top stack item is printed.

POPS (* s -- *) Like POP, but prints the string on top of the stack. Note: if the top item on the stack is not a string, things get messy. This instruction will become part of POP once typing is implemented.

CR (* -- *) Prints a carriage return on the terminal.

GET (* <type descriptor> -- n *)
Get input from terminal. Pops the type descriptor on TOS and gets input which is interpreted as being of the type described by the descriptor. ***In havoc version 1.0, an exception will be generated if the input cannot be properly interpreted.***

PUT (* <item>\<type descriptor> -- n *)
Put typed itemto terminal. Pops the type descriptor on TOS and puts item which is interpreted as being of the type described by the descriptor. ***In havoc version 1.0, an exception will be generated if the input cannot be properly interpreted.***

OPEN (* <file name>\<mode>\<file variable> -- *) Opens a file that has the name <file name> and assigns it to <file variable>. <mode> is a string that is either a "r", a "w", or "a" specifying if the file is to be opened for reading, writing or appending. Open is also used to access the serial ports. They have the special file names of "serialA" and "serialB" and are treated like files.

CLOSE (* <file variable> -- *) Closes the file specified by <file variable>. Also works on serial ports.

FGET (* <type descriptor>\<file variable> -- n \<success> *)
Get input from a file or device. Pops the type descriptor on TOS and gets input which is interpreted as being of the type described by the descriptor. It leaves a boolean success/failure word on top of the returned item.

FPUT (* <item>\<type descriptor>\<file variable> -- *)
Put typed itemto terminal. Pops the type descriptor on TOS and puts item which is interpreted as being of the type described by the descriptor.

LOAD (* <string> -- *) Tries to open the filename sitting on top of the stack. If successful, then the file is read into the input stream.

"KEY" (* -- flag *) Pushes a TRUE if there is keyboard input pending, otherwise pushes a FALSE.

File Handling

File I/O is handled by means of file variables. A file is opened for a read or write by means of the 'open' command. After this,I/O is handled directly by calls to fget and fput. In the final implementation of havoc, the user will be able to hide the actual typing involved in a file read or write. For now, though, one may simulate a device of type 'file' by defining a procedure that acts on a given file in a predefined manner.

example:

```
define variable
      in_file file              (* in_file will be our input file *)
      out_file file             (* our output file *)
enddef

"MyData" "r" in_file open     (* open "MyData" for reading *)
"OutData" "w" out_file open

string in_file fget           (* get a string of input *)
iftrue then
    string out_file fput      (* if the file get was successful, *)
endif                         (* write it to output *)

in_file close                 (* close up, we're done *)
out_file close
```

'Put' and 'get' send output to the terminal and work like fput and fget with one major difference: For 'get', there is no flag left on the stack to indicate success or failure. Presumably, the program will never reach an end-of-file from the terminal.

"CONTROL" (* n\v -- *) Writes control information n to the file or port with file pointer in v. N is an integer that defines the control information, such as

`BAUD_2400` or `FILE_RESET`

Since these numbers typically represent bit patterns, and are usually entirely indecipherable in their raw integer format, we recommend that they be defined as constants.

Not yet implemented*

Type Words

These words represent the different data types in havoc. Invoking any of them pushes a corresponding type descriptor onto the TOS. ***A type descriptor may be thought of as the name of the desired data type.*** If you wanted input an integer from the terminal, you would use the following sequence:

```
INTEGER GET          (* Get an integer from user *)
```

syntax:

```
<type descriptor> ::= INTEGER | LONGINT | BOOLEAN | CHAR
                    | FLOAT | STRING
                    | <user defined type>

<user defined type> ::= <identifier>
```

One possible use for this would be the conditional execution of code fragments based on what data type is on the top of the stack. One could, for example, do one thing if the TOS is an integer, and something else altogether if the TOS is a string.

example:

```
INTEGER =
  IFTRUE THEN FRED
   ELSE "Type mismatch, squidbreath!" POPS ABORT
  ENDIF
```

Type-related Words

Simple data typing (and, conseqently, all of these words) will be implemented in havoc version 1.0

TYPE? (* n -- n \ type *) Type query.
Pushes the data type of the TOS onto the stack.

SIZE? (* n -- n \ size(n) *) Size query.
Pushes an integer containing the size (in bytes) of the top stack item.

COERCE (* n \ <type> -- n *) Type coercion.
Changes item n into one of type <type>. Returns an error if the data types are incompatible.

DIRECTIVE WORDS

Procedural words always operate on the stack, and are thus affected only by words which precede them in the command stream. Directive words, however, operate on the words in the command stream which appear *after* them.

We've already discussed a special form of directive: flow of control words. The other type of directive word is a definition. This directs havoc to add a new entry to the dictionary. The words in the directive describe what the new word should represent and what should be done when it is encountered.

Definitions

Definitions provide the means by which you can effectively extend the language by creating new words other than the pre-defined havoc words, and place them in the dictionary. Definitions allow you to define data types, variables, procedures, "magic constants", and error conditions, and use them in future programs.

definition syntax:

```
<definition> ::= DEFINE <definition body> ENDDEF

<definition body> ::= <constant definition> | <type defintion>
                    | <condition definition>
                        | <variable definition> | <procedure definition>
```

Constant Definitions

syntax:

<constant definition> ::= CONSTANT { <identifier> <constant body> }

<identifier> ::= <word>

<constant body> ::= <number> | <string> | <type name>

description:

Constant definitions allow you to assign a descriptive word to a number or string that might otherwise mean nothing to someone reading the code of a program. When an identifier is defined to be a constant, invoking the identifier causes the compiler to make a textual substitution of the <constant body> for the <identifier>. Note that the <constant body> cannot be a variable, but must evaluate to a numeric or string constant.

example:

```
DEFINE CONSTANT phone_number 6436135 ENDDEF
```

Type Definitions

syntax:

```
<type definition> ::= TYPE <type list>

      <type list> ::= () | <type list> <word> <type name> <range>

<range>   ::=  '[' <integer> : <integer> ']'

<type name> ::= <type kind> | <file definition>
                | <record definition> | < device definition>

<type kind> ::= INTEGER | CHAR | LONG | BOOLEAN | FLOAT
                | STRING | <identifier>

<file definition> ::= FILE <name> <type kind>

   <record definition> ::= RECORD
                <field name> <type kind>
                    { <field name>  <type kind> } ';'

   <device definition> ::= DEVICE <name>
                    <type kind> ':' <identifier> <identifier>
```

description:

Type definitions allow you to define arbitrary data types out of the base types INTEGER, CHAR, LONG, BOOLEAN, FLOAT, STRING, FILE, and RECORD.

The sub-fields of the type 'RECORD' are accessed by means of a period (.) after the variable name, followed by the field name. This leaves the address of the subfield on the stack.

example:

```
(* Define a curve to be a 256-point array of integers *)
DEFINE TYPE curve INTEGER[0:255] ENDDEF

(* Define a generic "person descriptor" *)

DEFINE TYPE person RECORD:
            age        INTEGER
            salary     INTEGER
            weight     FLOAT
            shoe_size FLOAT;
 ENDDEF

(* Now define "fred" to be a "person" *)

  DEFINE VARIABLE fred person ENDDEF

42 fred.age WRITE       (* Write fred's age as being 42 *)

"Fred's shoe size is " POPS
fred.shoe_size @ POPLN (* What's his shoe size? *)
```

Devices are really just constructs that hide the inner workings of a device from the user and allow him or her to simply read from and write to a given device. A device definition specifies what size data the device works with and gives the names of the operations to be executed on a read and write of that device.

example:

```
DEFINE PROCEDURE serial_read (* read a char from the *)
BEGIN                         (* serial port *)
     DO
        CHAR serial_in FGET

        IFTRUE THEN EXIT (* if we did get a char *)
        ELSE DROP
        ENDIF
     LOOP
ENDDEF

DEFINE PROCEDURE serial_write   (* write a char *)
BEGIN
     CHAR serial_out FPUT
ENDDEF

DEFINE portA DEVICE     (* set up device definition *)
     CHAR: serial_read serial_write
ENDDEF

portA @ DUP         (* get a char and copy it *)
POP
portA WRITE             (* echo it back through port *)
```

Condition Definitions (Exception Handling)

syntax:

```
<condition definition> ::= CONDITION <condition name>
                                     <condtion type>

<condition name> ::= <identifier>

<condition type> ::= USER | <hardware trap condition>


<hardware trap condition> ::= OVERFLOW | CONVERSION | etc...
```

description:

Condition definitions provide a means for trapping and dealing with errors that would otherwise cause the program to fail. This is especially useful anytime that a program must accept input from the user, since it is never possible to predict what kind of garbage he/she will try to enter, no matter how clear and explicit your instructions are. The predefined hardware trap conditions include:

OVERFLOW for detecting numbers which are too big for the computer to handle;

UNDERFLOW for detecting numbers which are too small for the computer to handle;

CONVERSION for detecting attempted conversions between incompatible types (like trying to convert the string "dead frog" into an integer);

As a qualified user, you can define your own interrupt conditions, in much the same way that you can define your own data types. To define your own condition, the word CONDITION must be followed by the word USER, which indicates that you are about to supply your own condition, and then the name of your condition. This offer is valid for a limited time only, so act now and define your own conditions. Operators are standing by.

Condition definitions are always made to the dictionary, and may not be included inside other definitions.

examples:

```
(* The following example is the "add" routine from a
   hypothetical calculator program, and demonstrates the
   use of the predefined OVERFLOW condition.  We assume
   here that an input routine has been called earlier
   which inputs the numbers and pushes them on the stack.
   As usual, havoc words are in caps, while user-defined
   words appear in lower case *)

DEFINE PROCEDURE OVER_ERROR
BEGIN
```

```
    "Number too big!!"      (* Print error message *)
     POPS ABORT
  ENDDEF

  ON OVERFLOW OVER_ERROR (* If we get an overflow, call *)

  DEFINE PROCEDURE add    (* n1\n2 -- n1 + n2 *)
      +       (* Add the numbers. Pretty tricky, huh? *)
  ENDDEF

  ============================================================
  (* This example demonstrates the use of a simple user-defined
     condition.  The user has been asked to enter a series of
     numbers, which the program will store into an array.  If the
     user tries to enter more numbers than allowed , the condition
     is signalled and an error message is printed.
  *)

  (* Define the condition *)

DEFINE CONDITION USER too_many ENDDEF

  DEFINE PROCEDURE many_error
  BEGIN
      "Whoops, too many numbers!!" POPS   (* Print message *)
     ABORT
  ENDDEF

ON too_many many_error    (* Assign routine to condition *)

DEFINE PROCEDURE num_ent

    VARIABLE i INTEGER;    (* Loop index *)

  "Enter the numbers" POPS
  0 i WRITE                      (* No numbers entered yet *)
  DO
     i INC                       (* Increment loop index *)
    i @ max_nums >              (* If too many *)
    IFTRUE THEN
          too_many SIGNAL       (* signal the condition *)
    ELSE
          INTEGER GET           (* Input a number *)
          store_it              (* Store the number *)
    ENDIF
  LOOP

ENDDEF   (* procedure "num_ent" *)
```

Generating and handling exceptions:

Havoc supports software-generated interrupts and user-defined conditions in much the same way as PL/1 does. Conditions may be defined and signalled at any time in the program. Since there is no direct information passing allowed for the processing of interrupts, execution of the code is faster than regular procedure

invokations. This is well suited for applications such as reading a measurement at regular intervals, or periodic checking to see if a certain event has occurred.

syntax:

```
<on unit> ::= ON <condition> <command>

<condition> ::= <hardware trap condtion> |
                <user-defined condition>

<hardware trap condtion> ::= OVERFLOW | CONVERSION | etc.

<user-defined condtion> ::= <identifier>
```

description:

The ON directive marks the start of the code that is to be executed when the <condition> is signalled. Control is transferred to the <command> after the <condition> when the SIGNAL directive is encountered.

When the SIGNAL directive is encountered, the interpreter will search the dictionary for an <on unit> with a matching <condition>. If a match is found, control immediately transfers to the <command>. After the <command> has finished execution, control then transfers back to the word immediately following the <condition> that was SIGNAL'd. (got that?) If no match is found, a runtime system error occurs.

Individual interrupts may be enabled, disabled and signalled by the user.

syntax:

*** Parsing for these directives is not in place in the current version of havoc, although they will be completely supported in version 1.0***

DISABLE (* condition -- *) Disables specified interrupt of the <condition> on the TOS.

ENABLE (* condition -- *) Enables specified interrupt of the <condition> on the TOS.

SIGNAL (* condition -- *) Signals specified interrupt of the <condition> on the TOS.

example:

(see above examples)

Variable Definitions

Variables can be defined and stored into the dictionary to allow for static and global attributes to be associated with data objects.

syntax:

```
<variable definition> ::= VARIABLE <name> <type body>
                                    { <name> <type body> }

<name> ::= <identifier>

<type body> ::= <type kind> [ <range> ]

<range> ::= '[' <integer> : <integer> ']'

<type kind> ::= INTEGER | CHAR | LONG | BOOLEAN |
                FLOAT |STRING | <file definition>|
                <record definition> | <identifier>
```

description:

When the name of a defined variable is encountered in the command stream, its memory address is pushed on the stack.

The scope of locally defined variables is given higher precedence , so variables may be defined internally without regard to the names of any externally defined variables.

example:

```
(* The following definition sets up a number of variables which
store a value that any procedures can access *)

DEFINE VARIABLE number   INTEGER
                counter INTEGER
                message STRING
ENDDEF
```

Procedure Definitions

syntax:

```
<procedure definition> ::= PROCEDURE <name> <procedure body>

<name> ::= <identifier>

<procedure body> ::= [ <local definitions> ] <procedure block>

<local definitions> ::= [ <constant definition> ]
                        [ <type definition> ]
                        [ <variable definition> ]

<procedure block> ::= BEGIN { <command list> }
```

Compiler Words

FIRST_ENTRY (* -- addr *) Pushes the address of the compiler variable first_entry, a pointer to the first link in the dictionary.

DICT_TOP (* -- offset *) Pushes the offset from the start of user memory of the top of the dictionary.

DICT_START (* -- addr *) Global variable. Address of start of dictionary.

DUMP, DUMPSTACK (* -- *) These two words are picked up by the lexical analyzer and never actually make it to the parser. When encountered, the system prompts for a range of memory to display and dumps this to the screen along with a primitive attempt at disassembly.

The following macros make up havoc's assembly language. These pieces of code are inserted inline onto the workbench. The primitives that are not represented here are handled interpretively, by calls to assembly language routines built into havoc. The words that fall in this category are mostly the I/O routines that must set up calls to C routines or do any substantial word manipulation.

```
TEST_USP        TEST & POP (USP)

 The following instruction loads a CR into D0 for output purposes
LOAD_CR         MOVEQ #$D, D0

CMP_USP1        MOVE.L -(USP), D0
CMP_USP2        SUB.L $-4(USP), D0

BNE_NEXT        BEQ $#6(PC)
                JMP #<data>

BNE_SP1         BNE.S *+2
BNE_SP2         BRA.S *+4
                MOVE.L (SP), -(SP)
                RTS

BEQ_SP1         BEQ.S *+2
BEQ_SP2         BRA.S *+4
                MOVE.L (SP), -(SP)
                RTS

BRA_SP

POP             SUBQ.L #4 USP
POP2_SP         ADDQ.L #8 SP
POP_NEXT_SP     ADDA.L #<data> SP
DROPS_USP       MOVE.L -(A0), A0

PUSH_NEXT       MOVE #<data> (USP)+
PUSH_NEXT_SP    MOVE #<data> -(SP)

PUSH_OFFSET1    PEA #<data>
PUSH_OFFSET2    MOVE.L (SP)+,(USP)+

BREAK1          MOVE.L -4(USP),(USP)+
BREAK2          RTS

DUPLICATE       MOVE.L $-4(USP) (USP)+
```

```
SET_SFP         LINK #<data>
RESET_SFP       UNLK
USER            space filler

JUMP_NEXT       JMP #<data>
JSR_NEXT        JSR #<data>
RTS             RTS

ADD_USP1        MOVE.L -(USP), D0
ADD_USP2        ADD.L -(USP), DO
                MOVE.L D0, (USP)+

SUB_USP1        MOVE.L -(USP), D0
SUB_USP2        SUB.L DO, -(USP)
                ADDQ.L #4, USP

MUL_USP1        MOVE.L -(USP), D0
MUL_USP2        MOVE.L -(USP), D1
                MULS   D1, D0
                MOVE.L D0, (USP)+

DIV_USP1        MOVE.L -(USP), D0
DIV_USP2        MOVE.L -(USP), D1
 DIV_USP3       DIVS   D0, D1
                MOVE.L D1, (USP)+
                CLR.W #-4(USP)

MOD_USP1        MOVE.L -(USP), D0
MOD_USP2        MOVE.L -(USP), D1
MOD_USP3        DIVS   D1, D0
                CLR.W  D1
                SWAP   D1
                MOVE.L D1, (USP)+

NEG_USP         NEG.L -4(USP)

INC_ADDR         MOVEA.L -(USP),A2
                ADDQ.L #1, (A2)

DEC_ADDR        MOVEA.L -(USP),A2
                SUBQ.L #1, (A2)

NOT_USP         NOT.L -4(USP)

COPY_NTH1       MOVEA.L A0,A2
```

```
COPY_NTH2       MOVE.L -(A0),D0
COPY_NTH3       ADDQ.L #1, D0
                LSL.L #2,D0
                SUBA.L D0,A2
                MOVE.L (A2),(A0)+

AT_USP          MOVEA.L -(USP), A2
                MOVEA.L (A2), (USP)+

WRITE_USP       MOVEA.L -(USP), A2
                MOVE.L  -(USP),(A2)

SAVE_USP        MOVE.L A0, -(SP)
RESTORE_USP     MOVE.L (SP)+, A0

SWAP1           MOVE.L #-8(USP), D0
SWAP2           MOVE.L #-4(USP), #-8(USP)
SWAP3           MOVE.L D0, #-4(USP)
SWAP4

OVER            MOVE.L #-8(USP), (USP)+

QUIT_SYS        Trap: quit

TEST_EQ1        MOVEQ #1, D1
TEST_EQ2        MOVE.L -(USP), D0
TEST_EQ3        CMP.L   -(USP), D0
                BEQ   *+4
                CLR.L   D1
                MOVE.L D1, (USP)+

TEST_NE1        MOVEQ #1, D1
TEST_NE2        MOVE.L -(USP), D0
TEST_NE3        CMP.L   -(USP), D0
                BNE   *+4
                CLR.L   D1
                MOVE.L D1, (USP)+

TEST_GT1        MOVEQ #1, D1
TEST_GT2        MOVE.L -(USP), D0
TEST_GT3        CMP.L   -(USP), D0
                BLT   *+4
                CLR.L   D1
                MOVE.L D1, (USP)+
```

```
TEST_LT1        MOVEQ #1, D1
TEST_LT2        MOVE.L -(USP), D0
TEST_LT3        CMP.L   -(USP), D0
                BGT    *+4
                CLR.L   D1
                MOVE.L D1, (USP)+

AND_USP1        MOVE.L -(USP),D0
AND_USP2        BNE *+8
                CLR.L -4(USP)

OR_USP1         MOVE.L -(USP),D0
OR_USP2         BEQ *+8
                ST.L -4(USP)

LOAD_STR        LEA $E(PC),A2
```