UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT

Eh Reference Manual

Reinaldo S. C. Braga
November 1976
CS-76-45

Eh Reference Manual

Reinaldo S. C. Braga

Department of Computer Science
University of Waterloo

This research was supported by the National Research Council.

Table of Contents

Eh Reference Manual

Reinaldo S. C. Braga

1. Introduction

Eh is a programming language designed for writing portable systems and real-time software for minicomputers.

Programs written in Eh are *not* automatically rendered portable, in the same sense that programs written in high level languages are *not* correct simply because they compile. However, the language Eh facilitates the writing of programs which can be executed in many computers with few, if any, changes.

Eh evolved from B, a language developed at Bell Laboratories, described in "The Programming Language B", by S. C. Johnson and B. W. Kernighan, Bell Laboratories Computing Science Technical Report #8, January 1973.

Eh retains most of B's structure, which was slightly modified to eliminate portability hazards. Eh introduces the concept of controlling interrupts and incorporates a means of adding machine instructions, which cannot be generated by the compiler.

This manual covers every aspect of the Eh language. Except for the last section, examples consist of segments of programs; the variables referenced are assumed to be correctly defined.

Eh programs are usually executed under Thoth, which is a real-time executive. It has functions to perform I/O and a process monitor. Thoth is described in "A Portable Real-Time Executive – THOTH", by Lawrence S. Melen (Master's Thesis, University of Waterloo-1976).

Other general use functions are described in "The Portable Eh Library", by Michael A. Malcolm.

These documents include program listings which are examples of good programming style.

2. The Eh Machine

The Eh language is designed in a such a fashion that it can be implemented on most minicomputers. Due to the fact that minicomputers have different addressing capabilities and arithmetic units, a computer model has been chosen to specify the semantics of the language. This model is based on characteristics of real minicomputers, and is what we call the "Eh machine".

The Eh machine has the following characteristics:

It is a binary computer, but its arithmetic unit is not defined. That is, the arithmetic unit of the target machine is used as it is. Therefore the Eh machine might use one's complement, two's complement or even signed-magnitude arithmetic.

It is *word* oriented. All operators require (with one exception) word operands, and produce word results. A word is assumed to be at least 16 bits long.

Eh has one other type of data: the *string*. Strings are arbitrarily long sequences of *bytes*. A *byte* has at least 8 bits and no longer than a word. A *byte* is not a data type of the language. It exists only in strings. Strings are terminated by a special byte ('*0'). This byte cannot occur within a string.

Each word has a unique *address*. Addresses are represented by integers which fit within a word. The address of a word plus one is the address of the next consecutive word. Word addresses are referred to in the text as *Eh-addresses*, since on some hardware they differ from machine addresses. The primary assumption here is that the target machine has a linear (piecewise) store.

The Eh language makes use of a run-time stack (see Section 8). This stack is used by functions to allocate automatic variables, temporary results, arguments for function calls and other quantities. The part of the stack used by a function invocation is called a *stack frame*.

Elements in the stack frame are referenced in a arbitrary order. Therefore, it is highly desirable that the target computer has an index register which can be used for addressing the stack frame. The lack of a dedicated index register will adversely affect the execution efficiency of Eh programs.

Another implementation detail, which also affects the efficiency of execution, is that if the target computer is not word addressed, an Eh-address is a multiple of the machine address. This results in substantial overhead in converting Eh-addresses to machine addresses.

3. Basic Symbols

To enhance the portability of Eh programs, the ASCII character set is used (with one exception). Eh programs are written with the following subset of the ASCII character set:

-the *letters*, uppercase and lowercase.

-the *digits*: 0 1 2 3 4 5 6 7 8 9

-the special symbols : \ _ . , ; : { } [ ] ( ) ' " < > ? + − * / % ! & = ↑ ~ |

-the non-printable characters: space, horizontal tabulation and form feed. These can be used interchangeably to improve the style of Eh-programs.

The non-printable *new line* is the only change to the ASCII set: It denotes the end of a line, and takes the place of the ASCII character SO (octal 016).

-the keywords:

auto break case default disable else enable extrn for goto if next return select while

In this manual, keywords are printed in boldface, the other symbols are printed in roman. They are distinguished only in the manual, to stress usage. In source programs they are represented with ASCII characters.

Other printable ASCII characters can be used only within strings, character constants and comments.

3.1. Comments

The character '\' is used to begin *comments*. A comment is the text following the '\' up to the first new line character. Comments can appear anywhere.

Examples:

```
\ comments....
current_line ; \ variable used to store the current line number of the input file.
```

The last example shows a comment which is called a *remark*. Remarks are used to comment examples in the following sections.

4. Metalanguage

Eh is described with a metalanguage similar to Backus Normal Form. This metalanguage is used to define the syntax of Eh by giving rules for the synthesis of all the possible Eh programs.

The set of Eh's basic symbols is called *terminals* and is denoted by Σ. Σ^*, the reflexive and transitive closure of Σ, is the set of all possible phrases from Σ. The set of syntactically correct Eh programs is a subset of Σ^*.

The metalanguage is used to define subsets of Σ^*, in terms of other defined subsets and terminals. These subsets are denoted by names written in italics and they are referred to as *non-terminals*.

Non-terminals are defined by *productions* which are of the form:

$$\begin{aligned} subset_i : \quad & list1 \\ & list2 \\ & \vdots \\ & listn_i \end{aligned}$$

where $subset_i$ is the non-terminal being defined; listj ($1 \leqslant j \leqslant n_i$) are sequences of terminals and/or non-terminals. At most one of listj can be the empty set.

A production defines a non-terminal as the set produced by the union (for all j) of the concatenation of the terms in each listj. Here, *concatenation* is defined as:

If x and y are sets;

xy is the set $\{ ab \mid$ for all $a \in x$, for all $b \in y \}$

A terminal names the set consisting of itself.

There are some non-terminals which are more easily described in English. Non-terminals which fall in this category, are described as follows:

null is the empty set.
digit is the set { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }.
non-zero-digit is the set { 1, 2, 3, 4, 5, 6, 7, 8, 9 }.
octal-digit is the set { 0, 1, 2, 3, 4, 5, 6, 7 }.
hexa-digit is the union of *digit* and { a, b, c, d, e, f, A, B, C, D, E, F }.
letter is the union of the sets { a, b, c,...,z } and { A, B, C,...,Z }.
printable-ASCII is the set of printable ASCII characters except the character '*'.
alpha-character is the union of *letter* and { ., _ }.
new-line is the new line character.

5. Identifiers and Constants

5.1. Identifiers

Syntax:

alphanum-character : *alpha-character*
digit

identifier : *alpha-character*
identifier alphanum-character

Restrictions:

Identifiers consisting of the same letters which differ only in their case are regarded as the same identifiers. Identifiers must not exceed 32 characters in length.

Note that although the set of keywords (see Section 3) are valid identifiers according to the rules above, they cannot be used as such.

Identifiers beginning with '.' are used to designate system functions and external variables used in the Thoth operating system, the profiler, debugger, etc. Therefore, to avoid naming conflicts, the ordinary user is advised against using '.' as the first character of an identifier.

Examples:

a
first
i12.5
Key_word
.Nargs

5.2. Constants

Syntax:

constant : *constant-expr*
string-constant

constant-expr : *unary-op constant-token*
constant-token binary-op constant-token
constant-token

constant-token :	*octal-constant* *decimal-constant* *hexa-constant* *byte-constant* (*constant-expr*)
octal-constant :	0 *octal-constant octal-digit*
decimal-constant :	*non-zero-digit* *decimal-constant digit*
hexa-constant :	$ *hexa-digit* *hexa-constant hexa-digit*

Examples:

octal	decimal	hexadecimal
0	2	$F
0100	785	$0FE1

These forms of constants in addition to the *byte-constant* (described in next section) can be used to form expressions which can be used anywhere a *constant* is required.

5.3. Byte and String Constants

There are 2 ways to define constants which are used in character manipulation.

These constants are used to specify ASCII characters. The ASCII control characters must be specified by an *encoding* because they have special meaning to the I/O software. This encoding is done with printable characters. The character "*" is used as an *escape* symbol. Following are the escaped characters defined in Eh:

*0	- null character, also used as string delimiter
*b	- backspace
*e	- eot
*f	- form feed
*n	- new line
*r	- carriage return
*l	- line feed
*t	- horizontal tabulation
**	- asterisk
*'	- single quote
*"	- double quote
*#ooo	- ooo is an octal encoding
*$hh	- hh is a hexadecimal encoding

Syntax:

escaped-character is the set containing all escape sequences as defined above.

character-element :	*escaped-character* *printable-ASCII*
byte-constant :	*' character-element '*
string-element :	*null* *string-element character-element*
string-constant :	*" string-element "*

Semantics:

Except in global initialization (see Section 8), *byte and string constants* can be used wherever numeric constants can be used. Note that the definition of a string constant cannot start in one line and be continued on the next.

Internally, strings are stored in a packed format, several bytes per word.
Following the last specified byte of a string constant, an *end of string* delimiter (the ASCII null character *0) is automatically added.

Examples:

byte constants:

```
'a'
'0'
'*n'            \ the new line character
'*#034'         \ the ASCII FS
```

string constants:

```
"hello"
"Jul/6/1976 *n"
""              \ a null string
```

6. Names, values and pointers

Each identifier is associated with a unique word of the Eh machine. An identifier is said to *refer to*, or *name* its associated word. Syntactically, this is expressed by:

name : *identifier*

An identifier has a value which is the contents of the word it refers to. Given a name, its value is always accessible. Therefore,

value : *name*

specifies that the contents of the word referenced by name is to be used.

Constants have only a value. They cannot be used where a name is required. The value of a string-constant is a *pointer* to the word containing the first byte of the string. A pointer is a value which is used as an Eh-address, or as a relative displacement to a word.

There is an operator which can refer to bytes within strings. This operator has the following syntax:

byte-name : *value* { *value* }

The first value is a pointer to a string, and the second value is the offset which specifies which byte from the string is to be referred to. A byte-name can be used anywhere as a value. That is,

value : *byte-name*

means that the byte referred to by byte-name is available (right-justified in a word) as a value.

To store a value into a byte within a string it is necessary to use the operator "=", described in the Section 7.5.

There are two operators for converting pointers into names, these operators are the indirection operator and the array indexing operator, which are described in the Sections 7.6.3 and 7.6.4; and there is one operator for converting a name into a pointer, it is the address operator and is described in the Section 7.6.2.

It should be noted that the non-terminals *name* and *value* can represent expressions, as it is described in the next section.

7. Expressions

Eh has a large set of operators. These operators correspond closely with operators available on most computers. Thus one can code expressions using the most convenient operators and the expressions can be compiled to efficient machine code. Eh does not check the validity of arguments during execution. An incorrectly written program could try to address nonexistent memory locations causing hardware traps. The results of such hardware traps are computer dependent. Wherever possible, such exceptions are ignored.

The operators are formed with one or more symbols. If an operator is represented by two or more symbols, then there must be no spaces between the symbols.

Each operator in an expression has a *precedence* value. This precedence is used along with association rules to determine the order of evaluation of the expression. If two neighboring operators have the same precedence, the left operation is performed before the right, except in the cases of assignment operators and the query for which the right operation is performed before the left.

The highest-precedence operation is designated by the *parentheses* which are used only for controlling the order of evaluation. Its syntax is:

value : (*value*)

name : (*name*)

The order of operand evaluation is also fully defined. Most binary operators require that the left operand be evaluated before the right, after which the operation is performed. In function calls, the arguments are evaluated in a left-to-right order, then the function name is evaluated, and the function call is performed. The assignment and array operators evaluate the right operand before the left.

Note that the order of evaluation is fully defined and hence side effects are well-defined and therefore portable.

Operators can be roughly divided into the following categories:

1 arithmetic
2 bitwise
3 relational
4 logical
5 assignment
6 miscellaneous

7.1. Arithmetic Operators

Syntax:

value :	*value binary-arith-oper value* *unary-arith-oper value*
binary-arith-oper :	+ − * / %
unary-arith-oper :	−

Semantics:

+ results in the algebraic *sum* of its operands.
− (binary) results in the *difference* of its operands.
* results in the *product* of its operands.
/ results in the *quotient* of its operands.
% results in the *remainder* of the division of its operands. This result conforms with the value most computers return as a remainder when dividing two numbers. That is, given x and y, the relation

x ≡ (x % y) + (x / y) * y

is true.

The unary operator − results in the *negation* of its operand.

7.2. Bitwise Operators

Syntax:

value :	*value bitwise-oper value* ~ *value*
bitwise-oper :	<< >> & ↑ |

Semantics:

~ results in the logical *one's complement* of its operand.
<< results in the *shifting* of the left operand to the *left* by the number of bits specified by the right operand.
>> is similar to << but the *shifting* is to the *right*.
& results in the *bitwise and* of its operands.
↑ results in the *bitwise exclusive-or* of its operands.
| results in the *bitwise or* of its operands.

The operations above treat the operands as strings of bits. The shift operators always fill vacated bit positions with zeros.

7.3. Relational Operators

Syntax:

value :	*value relational-oper value*
relational-oper :	==
	!=
	<
	<=
	>=
	>

Semantics:

== is the *equality* operator;
!= is the *inequality* operator;
< is the arithmetic *less than* operator
> is the arithmetic *greater than* operator;
<= is the arithmetic *less than or equal* operator;
>= is the arithmetic *greater than or equal* operator.

The result of any of these operators is 1 if the relation is true; 0, if the relation is false.

7.4. Logical Operators

Syntax:

value :	*value binary-logical-oper value*
	unary-logical-oper value
binary-logical-oper :	&&
	||
unary-logical-oper :	!

Semantics:

&& is the logical *and* operator;
|| is the logical *or* operator;
! is the logical *complement* operator.

The result of these operators is:

&& : 1 if both operands are non-zero, 0 otherwise

|| : 1 if either (or both) of its operands is non-zero, 0 otherwise

! : 1 if its operand is zero, 0 otherwise

7.5. Assignment Operators

Syntax:

value : *name assign-oper value*
byte-name = value

assign-oper :
=
+=
−=
*=
/=
%=
>>=
<<=
&=
↑=
|=

Semantics:

The operator = is the conventional assignment; the value replaces the contents of the object referred to by *name*. (or *byte-name*.) Expressions using the other operators are equivalent to

name = name **x** value

where **x** is one of the symbols which can precede "=", in the *assign-oper* set, except for the fact that name is evaluated only once.

A *byte-name* can only be used as the left operand for the simple assignment operator or as a *value*. For example,

x { y } += 1 is illegal.

7.6. Special Operators

There are several operators which are used mainly in conjunction with array addressing, or indirect addressing. These operators are also used in other ways.

7.6.1. Increment/Decrement Operators

Syntax:

value : *name inc-dec-op*
inc-dec-op name

inc-dec-op : ++
−−

Semantics:

The value of the *named* word is incremented (or decremented) by 1. If the operator precedes the operand, then the result is the value of the operand after the increment (decrement). If the operator follows the operand, then the result is the value of the operand before the increment (decrement).

Examples:

If the identifier x initially has the value 10, evaluation of the expression

i = ++x

causes both x and i to have the value 11.

But after evaluation of the expression:

i = x++

the value of i is 10 and the value of x is 11.

7.6.2. Address Operator

Syntax:

value : & *name*

Semantics:

The address operator results in the Eh-address of the operand.

7.6.3. Indirection Operator

Syntax:

name : * *value*

Semantics:

The indirection operator results in a name.

7.6.4. Word Indexing Operator

Syntax:

name : *value* [*value*]

Semantics:

This operator is used to reference an element in a word array. This operator is commutative (except for side effects), that is one of the operands is regarded as a absolute pointer, the other is regarded as an offset (or a relative pointer.)

The address, indirection and word indexing operators have the following properties (except for side-effects in X and Y):

X[Y] ≡ Y[X] ≡ *(X + Y) ≡ *(Y + X)

*&X ≡ &*X ≡ X

The symbol ≡ denotes equivalence.

*&X is another name for the word referred to by X, whereas &*X yields the value of X.

7.6.5. The Query

Syntax:

value : *expr-test* ? *expr-true* : *expr-false*

expr-test : *value*

expr-true : *value*

expr-false : *value*

Semantics:

The *expr-test* is evaluated. If this value is nonzero then *expr-true* is evaluated and is the result of the operator. Otherwise the *expr-false* is evaluated and is the result of the operator.

7.6.6. Function Calls

Syntax:

value : *name* (*arg-list*)

arg-list : *null*
non-null-arg-list

non-null-arg-list : *value*
non-null-arg-list , *value*

Semantics:

The arguments within *arg-list* are evaluated left-to-right and the results of these expressions are passed as arguments to the function referred to by the *name*. If the function returns a result, it is the value of the expression *name* (*arg-list*); otherwise the resulting value is undefined.

7.7. Precedence of Operators

The operators are listed below in decreasing levels of precedence. All operators on a given line have the same precedence.

1 ++ and −− postfix, the array operators, function calls.
2 ++ and −− prefix, −, ~, &, *, !, (all unary operators)
3 <<, >>, &, ↑, |
4 *, /, % (binary)
5 +, − (binary)
6 ==, !=, <, <=, >, >=
7 &&
8 ||
9 ?
10 = and all other assignments.

8. Storage Class

An Eh program has the following syntax:

eh-program : *module*
eh-program module

module : *external*
function

Semantics:

An identifier can be defined either with a global scope, making it potentially accessible by all functions in the program, or with a local scope, in which case the identifier is accessible only within the function in which it is defined. Local variables are allocated from a run-time stack. A function allocates a frame from the stack for its local variables every time it is invoked. The frame is released when the function returns.

Since Eh has one global level of identifiers, modules are defined sequentially, unlike algol-type languages, where functions can be defined within functions.

Different modules may be defined with identical names, but the loader loads the first instance of a module and ignores any others with the same name.

Within a function there is another type of identifier, the *label*. A label has local scope and can only be used in **goto** and **twit** statements within the same function. (See Section 10.)

An identifier with global scope is called *external* and a local scope identifier (except for labels) is called *automatic*.

8.1. Externals

Syntax:

external : *identifier initial-value* ;
identifier { *size* } *initial-string* ;
identifier [*size*] *initial-list* ;

size : *null*
constant

initial-value : *null*
constant
identifier

initial-string : *null*
string-constant

initial-list : *null*
list

list : *constant*
identifier
string-constant
[*list*]
list , *list*

Semantics:

The values of the initializers (elements from *initial-list*, *initial-string* or *initial-value*) are:

null : 0
constant : its value
string-constant : a pointer to the string bytes
identifier : a pointer to the *external* or *function* referred to by the identifier. The identifier must be defined elsewhere in the program, with a global scope.
[*list*] : a pointer to an area where the values of elements in *list* are stored consecutively.

The *identifier* defined as external refers to a word; its *value* depends on the form of its declaration.

The form: *identifier initial-value*

sets the value of the *identifier* to the value of *initial-value*.

Examples:

```
A ;          \ value is zero
B 10 ;       \ value is 10
C A ;        \ value is a pointer to A
```

The form: *identifier* { *size* } *initial-string*

sets the value of the *identifier* as a pointer to an area. If a null *initial-string* is given, the area is allocated to contain (*size*+1) null bytes. If there is a non-null *initial-string*, the area is initialized with the string, and the allocation is done according to the larger of (*size*+1) and the size of the string constant.

Examples:

```
C { 10 } ;          \ Reserves a 11-byte area.
D { 15 } "hello" ;  \ The 16-byte area contains "hello".
E { } "january" ;   \ The 8-byte area contains "january".
F { 2 } "two" ;     \ The 4-byte area contains "two".
```

The form : *identifier* [*size*] *initial-list*

sets the *identifier*'s value to a pointer to an area. This area is allocated in terms of words. Values contained in *initial-list* are stored in consecutive words of this area. As with the previous form, the size of the area is the larger of the specified (*size*+1) and the actual number of elements in the list.

Examples:

```
G [ 10 ] A, B, C, D, E, F;
```

This statement defines an 11 word area, the first 6 words contain pointers to A, B, C, D, E and F. The rest are zero. The value of G is a pointer to this area.

```
H [ 1 ] 0, "data", 'a';          \ The vector is 3 words long.
input [ 100 ] ;                  \ The vector is 101 words long
FUNCS[ ] act1, act2, act3;       \ The vector is 3 words long.
Ident2by2[ ] [ 1, 0 ], [ 0, 1 ];
```

The last statement shows how a 2-dimensional 2-by-2 identity matrix could be defined.

8.2. Automatic variables

A function allocates all its automatic variables, plus environment variables (e.g. return address) and temporaries, in a stack frame. A new frame is allocated every time the function is invoked.

Identifiers are defined to be automatic with the **auto** statement.

Syntax:

auto-stmt :	**auto** *auto-defns-list*
auto-defns-list :	*auto-defn* *auto-defns-list* , *auto-defn*
auto-defn :	*identifier* *identifier* [*size*] *identifier* { *size* }
size :	*constant*

Semantics:

The **auto** statement is used to define identifiers with a local scope.

Identifiers defined with the form: *identifier* [*size*]

have as value the pointer to an area, also allocated from the run-time stack. The area has (*size*+1) words.

The form: *identifier* { *size* }

sets the identifier's value to a pointer to an area of (*size*+1) bytes, which is also allocated from the stack. These pointers are set at run time upon function entry.

Examples:

```
auto i;
auto a, b1, c, d;
auto j[ 20 ], k{ 10 };
```

The last statement sets the value of j to be a pointer to a 21-word area and the value of k to be a pointer to a 11-byte area.

9. Functions

Syntax:

function :	*identifier* (*list-of-args*) *statement*
list-of-args :	*null* ? *formal-arg-list*
formal-arg-list :	*identifier* *formal-arg-list* , *identifier*

Semantics:

The function definition has several effects: code is generated for *statement*; the identifier's name refers to the first word of the code; and the formal arguments (if any) are allocated in the stack frame, as if they are automatic variables.

The form of function definition which uses *formal-arg-list* allows reference to the arguments by their variable names. The function can be called with as many actual arguments as desired. That is, there is no necessity for the caller to pass all arguments specified in the function definition. The function can determine the number of arguments actually received, by calling the function .Nargs. For example,

```
na = .Nargs();
```

is used to store in the variable na the number of actual arguments passed to the current function being executed.

The formal argument "?", together with the functions .Nargs and .Arg is used in functions which can potentially receive an arbitrarily large number of actual arguments. Within such functions, the expression

```
x = .Arg( n );
```

sets x to the value of the n-th argument.

Example:

```
.Equal ( s1, s2 )

   \ Compare s1 to s2.

   \ return:  0 if s1 != s2
   \          1 if s1 == s2

   \ s1 == s2 iff they are of equal length and each
   \ corresponding byte is the same.
  {
    auto  i, c;

    i = 0;

    while ( (c=s1{i}) && c == s2{i} ) ++i;

    return( c == s2{i} );
  }
```

This example was typeset from the Eh library. It defines the function .Equal, which determines if two given strings are identical.

10. Statements

Syntax:

statement :
simple-stmt ;
{ *statement-list* }
alternative-stmt
iterative-stmt
label : *statement*
;

statement-list :
statement
statement-list statement

simple-stmt :
decl-stmt
transfer-stmt
interrupt-control-stmt
twit-statement

Note that the form *label* : *statement*

defines the identifier *label* as a name which refers to the following statement.

10.1. Declarative Statements

Syntax:

decl-stmt :
auto-stmt
extrn-stmt

Semantics:

The declarative statements must precede all other statements within a function. The **auto** statement is described in Section 8.2.

Extrn Statement

Syntax:

extrn-stmt :
extrn *extrns-list*

extrns-list :
identifier
extrns-list , *identifier*

Semantics:

The **extrn** statement is used to establish accessibility of global identifiers in the body of functions.

Examples:

```
extrn a,b,c,d;
extrn var;
```

10.2. Transfer Statements

Syntax:

transfer-stmt : *goto-statement*
next-statement
break-statement
return-statement

Goto Statement

Syntax:

goto-statement : **goto** *label*

Semantics:

The *label* must be defined in the function. When a goto statement is executed, control is transferred to the statement referred to by the *label*.

Next and Break Statements

Syntax:

next-statement : **next**

break-statement : **break**

Semantics:

next and **break** are used within iterative statements (**for**, **repeat**, **while**).
next causes the next iteration to begin; **break** causes termination of the iteration.

Return Statement

Syntax:

return-statement : **return**
return (*value*)

Semantics:

This statement is used to return control of execution to the point where the function was called. The second form is used to return a value. (See Section 7.6.6.)

10.3. Interrupt control statements

Syntax:

interrupt-control-stmt : **enable**
disable

Semantics:

These statements are provided to allow control of the target machine's interrupt mechanism. **disable** prevents hardware interrupt requests from being honoured. In some machines it is impossible to disable all interrupts, for instance, "Power Fail" interrupts.

The **enable** statement restores the interrupt level, or mask to that which the program had before the previous **disable**. **enable** must be in the same function as its matching (previously executed) **disable**.

If the program is being executed in a disabled mode, the statements:

enable; **disable**;

allow interrupts to occur between execution of the **enable** and **disable**.

These statements are not required in most programs; they are used in some operating system functions and real-time application programs. Their indiscriminate use can cause subtle errors and even hardware traps on some computers.

10.4. Twit Statement

Syntax:

twit-statement : **twit** (*list-of-arguments*)

Semantics:

This statement allows highly machine specific code to be inserted directly into a program.

The number and form of arguments of the **twit** statement, and its semantics, are implementation dependent.

10.5. Alternative Statements

Syntax:

alternative-stmt : *if-statement*
select-statement

If Statement

Syntax:

if-statement : **if** (*value*) *true-stmt*
if (*value*) *true-stmt* **else** *false-stmt*

true-stmt : *statement*

false-stmt : *statement*

Semantics:

The *false-stmt* is associated with the first immediately preceeding *if-statement* which does not have a *false-stmt*.

The *value* is evaluated. If the result is non-zero, then the *true-stmt* is executed and the *false-stmt* statement, if any, is not executed. If the result is zero, only the *false-stmt* statement, if any, is executed.

Examples:

```
if( x == 0 ) y = 1;
```

y is assigned to the value of 1, if and only if the value of x is zero.

```
if ( !c ) ++null_cnt; else putchar( c );
```

If c is zero, the variable null_cnt is incremented. If c is non-zero, the function putchar is called with c as an actual parameter.

Select Statements

Syntax:

select-statement :	*string-select-statement* *word-select-statement*

String select statement

Syntax:

string-select-statement :	**select**{ *value* } { *str-case-statement-list* }
str-case-statement-list :	*str-case-statement* *str-case-statement-list str-case-statement*
str-case-statement :	*s-label* : *statement* *s-label* : *str-case-statement*
s-label :	**case** *string-constant* **default**

Semantics:

The *s-label* string constants must be unique within each string select statement. The execution of a string select statement is logically equivalent to a chain of **if-else** statements. That is, first, *value* is evaluated (only once), then this result, which is a string pointer, is compared (using string comparisons) with the string constants from the *s-labels*. When the comparison yields an equal result, the statement associated with the string constant is executed. If all comparisons fail, the statement labelled **default**, if any, is executed.

The select statement shown below on the left is equivalent to the statement on the right, except for the use of the variable s.

```
select { expression }                {
  {                                    s = expression ;
    case  "abc" : statement1           if( .Equal( s, "abc" ) ) statement1
    case "def" : statement2            else if( .Equal( s, "def" ) ) statement2
    .                                  .
    .                                  .
    case "wxyz" : statementn           else if( .Equal( s, "wxyz" ) ) statementn
    default : statementd               else statementd
  }                                  }
```

Word Select Statement

Syntax:

word-select-statement :	**select(** *value*) { *word-case-statement-list* }
word-case-statement-list :	*word-case-statement* *word-case-statement-list word-case-statement*
word-case-statement :	*w-label* : *statement* *w-label* : *word-case-statement*
w-label :	**case** *constant* **case** *relational-op constant* **case** *constant* :: *constant* **default**
relational-op :	< <= > >=

A *w-label* defines a range of integers. The table below gives the ranges defined by each possible *w-label.*

case *const* :	const
case > *const* :	const+1 through maxinteger
case >= *const* :	const through maxinteger
case < *const* :	mininteger through const-1
case <= *cte* :	mininteger through const
case *const1* :: *const2* :	const1 through const2.

maxinteger is the largest positive integer and *mininteger* the largest (in magnitude) negative integer available in the target computer.

The *word-select-statement* has the following semantics: The *value* is evaluated and considered to be a signed integer. The statement associated with the range which contains this integer is executed. If no range contains this integer, the statement labelled by **default**, if any, is executed.

The ranges defined by *w-labels* in a word-select statement must be mutually exclusive. That is no integer may be in more than one of the specified ranges.

The example below shows on the right a statement which is equivalent to the word select statement on the left; except for the use of the variable e:

```
select( c = getchar() )
  {
    case 'a' :: 'z' : c += 10 - 'a';
    case '0' :: '9' : c -= '0';
    case 'A' :: 'Z' : c += 10 - 'A';
    default : c = -1;
  }
```

```
{
  c = getchar();
  if( 'a' <= c && c <= 'z' ) c += 10 - 'a';
  else if( '0' <= c && c <= '9' ) c -= '0';
  else if( 'A' <= c && c <= 'Z' ) c += 10 - 'A';
  else c = -1;
}
```

In these examples, the variable c is set to a value between -1 and $f, according to the value returned by the function getchar.

10.6. Iterative Statements

Syntax:

iterative-stmt : *repeat-statement*
while-statement
for-statement

Repeat Statement

Syntax:

repeat-statement : **repeat** *statement*

Semantics:

The *statement* is repeatedly executed.

Example:

```
i=0;
repeat
   {
      if( s[i] == '*0' ) break;
      ++i;
   }
```

In this example, the **repeat** statement is used to find the length of the string s.

While Statement

Syntax:

while-statement : **while** (*value*) *statement*

Semantics:

The **while** statement is equivalent to the statement:

repeat{ if(!*value*) **break**; *statement* }

Example:

```
i = 0;
while( s{ i } != '*0' ) ++i;
```

For Statement

Syntax:

for-statement : **for** (*expr-init* ; *expr-test* ; *expr-incr*) *statement*

expr-init : *value*
null

expr-test : *value*

expr-incr : *value*
null

Semantics:

The for statement is equivalent to:

```
{
  expr-init ;
 while( expr-test )
 {
   statement
   expr-incr;
 }
}
```

Example:

```
for( i=0; s{i} ; ++i) ;
```

11. Manifests and Input File Indirection

Manifests

Syntax:

manifest : *identifier* = *manifest-text*

manifest-text is a string of characters delimited by the first of *new-line* or '\'.

Semantics:

After the manifest is defined, any occurrence of the identifier is replaced by its manifest-text. Such identifiers can be used anywhere, even within other manifest declarations.

Manifests can be defined only outside function definitions.

A cautionary note:
Because manifests are simple substitutions, the following chain

```
A = 1
B = A+A
C = B*B
```

will cause any occurrence of the manifest C to be expanded into the sequence

```
1+1*1+1
```

The precedence of the operators * and + in this case will cause a different result than was probably intended. A safe rule of thumb is to use parentheses around expressions in manifests.

Input file indirection

Whenever the ASCII character "%" occurs in the first column of a source file input line, the characters following it are regarded as a pathname of a file which contains part of the source program which, in effect, replaces this entire line containing the "%" in column one.

Appendix I

Differences between Eh and B

Identifiers can be up to 32 characters long in Eh and they are not truncated by the compiler. Identifiers longer than 32 characters are illegal in Eh.

Character constants in B can have up to 4 characters between the single quotes, they are limited to one character in Eh.

Hexadecimal constants are allowed in Eh. They begin with the character '$'.

Strings are delimited by *0 in Eh. Some escape sequences, which are available in B, are not available in Eh: *<, *>, *(and *). The character *n has the value 016 in Eh. The character *l has the value 012.

B allows only a list of constants to be initialized in external variable definitions. In Eh it is possible to initialize edge vectors. Also there are some syntactical differences. See Section 8.1 for the syntax of Eh's external definition.

The convention in B to accept alternative forms for some symbols ($(for {, etc) is not supported in Eh.

The "&" and "|" operators have dual roles in B. They are bitwise operators in normal expressions, but are logical operators in expressions which are evaluated to be tested, as in if statements. Eh introduces the "&&" and "||" operators which are the logical operators, and keeps "&" and "|" as bitwise operators.

The operators have different precedence relationships. For example, in Eh the bitwise operators (<<, >>, &, ↑, |) have higher precedence than the arithmetic operators. (See Section 7.7 for the precedence of Eh operators.)

The order of evaluation of expressions is well defined in Eh. See Section 7 for Eh's operator binding rules.

The assignment operators have been reversed, relative to those in B; for instance, B's =+ is denoted in Eh as +=. This convention eliminates ambiguities in expressions like a= -1 where the space between the "=" and the "-" is very important in B.

The B language supports strings but does not have operators to fetch and store bytes within strings. Eh has the byte indexing operator "{".

The **switch** statement in B is not implemented. It has been replaced by the **select** statements. The word-select statement is related to the switch statement in B. The major difference between the two is that the **switch** is a multi-way branch, and the **select** only executes the statement labelled by the matching case label. The select statement is logically equivalent to a chain of **if** ... **else** statements. The case labels are also different; a range can be specified as well as single constants. The ranges and constants cannot overlap in Eh, as they can in B.

The string select is available only in Eh. (See Section 10.5 for its syntax and semantics).

The commands **enable**, **disable** and **twit** do not exist in B. The first two commands control interrupts; the last is used to add inline machine instructions.

Eh introduces the formal function parameter "?" to denote an unbounded list of arguments, making it possible to write functions which receive an arbitrary number of arguments.

The library and I/O functions differ greatly from B's library of functions.

The form /*...*/ for comments in B is replaced by \...<nl>
in Eh; where <nl> is the end of line delimiter.

In Eh, manifests are delimited by '\' or <nl>, instead of ';' as in B.