University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

The NESS Reference Manual

Wilfred J. Hansen

Persistent Programming
Research Report 43
June 1987

Alan Dearle

June, 1987

Ness -
Reference Manual

Wilfred J. Hansen*

Computer Science Department
University of Glasgow

* Author's present address:
Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract: Ness is a small programming language designed to explore an algebra for substring expressions. In addition to substring functions, constants, and concatenation, the language offers assignment, if-then-else, while-do, and function definition. The language is implemented as a preprocessor that produces a C program, so C source code can be embedded in Ness programs to utilize other data types.

1. Introduction

The Ness* language has been implemented to explore the notions of an algebra for substring expressions [Algebra]. The current implementation is as a preprocessor that produces a C program; this in turn utilizes the Em subroutine package [Em].

* The name was chosen both because the language utilizes a Notation for Expressions over SubStrings and to honor the home of one of Scotland's immortals. The correct typography for the name is sans serif.

Almost all values in Ness are substrings. A constant string denotes a substring which refers to the entire constant; concatenation creates a new

string but returns a substring value referring to its entirety; all function arguments are substrings and all functions return substring values.

As an example of Ness, the following program counts the words in a text. See the discussion after the program.

```
-- wc.mc
-- count the words in a file

#include <stdio.h>

marker letters;    -- a list of the letters that may occur in words

-- incr(n) assumes n is a string representation of a positive
--      integer and modifies it to the successor integer
--
function incr(n) == {
        if n = "" then  {
                -- empty string, extend front of integer
                replace(n, "1");
                return;
        }
        -- replace last digit with its successor:
        replace(last(n),  next(search("01234567890", last(n))));

        if last(n) = "0" then           -- recur to carry to the left
                incr(extent(n, start(last(n))));
}
-- countwords(text) counts the number of sequences
-- of adjacent letters
--
function countwords(text) == {
        marker count;
        count := newbase();
        text := token(text,letters); -- find first word
        while text /= "" do {
                incr(count);            -- count the current word
                text :=                         -- find next word
                        token(next(text), letters);
        }
        return count;
}

-- the main program initializes constants and then
-- reads a file and counts the number of words in it
--
```



```
function main(args) == {
        marker filenameletters, filename;

        letters := "qwertyuiopasdfghjklzxcvbnm"
                ~ "QWERTYUIOPASDFGHJKLZXCVBNM";
        filenameletters := letters ~ ".0123456789";

        -- extract first argument from args
        filename := span(next(token(args, " \t")),
                        filenameletters);

        -- read file, count words, and print result
        print ("The text has "
                ~ countwords(readfile(filename))
                ~ " words\n");
}
```

Incr() modifies its argument recursively, processing digits from right to left. If the argument is empty, a carry has overflowed and the value is extended with a "1" on the left. Otherwise the low order digit is incremented: The "search" operator returns the one-character substring of "01234567890" corresponding to the last digit of *n*; the next() character after this substring is the next larger digit. If the resulting digit is zero, incr() is called recursively to increment the portion of the integer preceding the last digit. (It is as easy to write a version of incr() which constructs a new value rather than modifying its argument.)

The newbase() call in countwords() creates a new, non-constant empty string. It is later revised with incr() to have successive integer values. Each token() call scans forward from the current position in *text* (its first argument) looking for the next substring composed of characters from the string *letters*. If one is found, it is counted and the process continues. Otherwise the procedure exits, returning the *count* value.

The parameter to main() is given as its value the shell command line which called the program. The token() operation looks for the end of the first argument (which is the program name itself) and then the span() operation finds all subsequent letters, digits, and dots, which is how this program defines a filename. Finally, all the processing takes place in the expression

```
countwords(readfile(filename))
```

which reads the entire file into memory, passes a marker for it to countwords(), and returns the string representation of the number of words.

Many features of Ness are exhibited in this program:

The substring computation functions, newbase(), next(), replace(), . . . are written as standard function calls.

Tilde, "~ ", is the string concatenation operator.

By convention a failed search returns an empty substring at the beginning of the text which was searched.

Comments begin with "--" and extend to the end of the line.

A declaration is the word "marker" followed by a list of identifiers.

All variables and values are of type marker (except for the control values in **if** and **while**.) {It would really be preferable to have an integer data type for this example to avoid the heaviness of incr().}

Parameters in a formal parameter list are not declared, nor is the type of the function.

Execution begins with the function named "main".

Statement bracketing is done with { . . . } as in C.

String constants are as in C and may include backslash escapes like \n.

C preprocessor statements may be included.

The program could be made briefer if the language were richer. Since it is intended for study of substring expressions the language is deliberately weak in other areas: there are no embedded assignments and the only flow of control constructs are if-then-else and while-do.

The example illustrates most of the Ness language. In addition you will want to know the various Ness operators; these are described in Section 4. Further examples appear in Appendix A.

2. Lexemes

Ness comments begin with double-dash (--) and extend to the end of the line. White-space characters – (space) \n \t \f \b \v \r – may appear between tokens. Dollar signs ($) and hashes (#) begin comments that are passed through to the C compiler; these are described in Appendix C.

The tokens of Ness are identifiers, strings, reserved words, a few multi-character sequences, and certain single characters.

Identifiers. An identifier is a sequence of letters (no digits). Reserved words cannot be used as identifiers. The names of pre-defined functions can be used as variable names, but cannot be used as the names of functions.

Strings. A Ness string has the same structure as a C string, delimited with double quotes and possibly containing backslash escapes. (However, strings are not directly incorporated in the program as C strings. They are read into the executing program from the *programname*.cons file.)

Reserved words. The reserved words are

function if then else while do marker return True False

Pre-defined functions. The pre-defined function names are

newbase next base start extent replace bound readonly
end front first second rest allprevious allnext advance
last previous search match anyof span token
readfile print

Multi-character sequences. The following multi-character sequences are part of the language:

= := ~:= /= <= >= .= .> .<

Other. Other characters which are not letters or digits are treated each as a single character. The characters used are

; , ( ) { } ~ ^ | & = > <

3. Syntax

In the Syntax rules a double slash means indefinite repetition of the preceding symbol with instances separated by the following symbol, which may be blank.

3.1. Programs, Functions, and Statements

A Ness program is a sequence of declarations of global variables and functions. The body of a function is a statement and there are nine varieties of statement:

```
program ::=  global//

global ::=  'marker'  var//, ;          -- global variables
       ::=                              -- function definition
            'function'  fnm  (  var//, )  ==  stmt

stmt   ::=  'marker'  var//, ;          -- local variables

       ::=  'while'  pred  'do'  stmt
       ::=  'if'  pred  'then'  stmt
       ::=  'if'  pred  'then'  stmt  'else'  stmt
       ::=  'return'  expr  ;           -- return a value
       ::=  'return'  ;                 -- return no value
       ::=  {  stmt//  }                -- compound statement

       ::=  fnm  (  expr//, )  ;        -- function call
       ::=  var  :=  expr  ;            -- assignment of marker values
       ::=  var  ~:=  expr  ;           -- append
```

Variables must be declared before use. Globally declared variables may be referenced in any subsequent function, but local variables can only be referenced within the function where they are declared. All global identifiers (function names and variable names) must be unique and all identifiers within any one function must be unique. A local variable may have the same identifier as a global variable, and then that function can only refer to the local variable. {Note that unlike C, a variable declared within an inner compound statement is accessible outside that compound statement and must not have the same identifier as any other variable in the function.}

A function call must have the same number of arguments as there are formal parameters in the function definition. The formal parameters are implicitly declared to be of type **marker** and must not be redeclared within the function body. The parameters are call-by-value: a change to the formal parameter marker within the function has no affect on the actual argument variable passed in the call. However, if a portion of the base string of a formal parameter is replaced, the base string of the actual argument will be changed as well. The **return** statement causes an exit from its function back to the point where the function was called. A function may or may not return a value.

There must be a function with the name "main". It will be executed first, having as its argument the command line with which the program was called. {The argument will have a single blank between each segment of non-blank characters given on the command line.}

Statements end with semi-colons.

An 'else' is paired with the nearest preceding unmatched 'then'.

An assignment statement assigns the substring marker value computed on the right side to the variable named on the left. Predicates, however, compare the characters of the substrings referred to by the arguments.

The append statement

```
var ~:= expr;
```

is defined to have the same effect as

```
var := extent(var, replace(end(var), expr));
```

3.2. Expressions

Most of the forms of substring expression are implemented as pre-defined functions, so the grammar for expression is largely for predicate expressions:

```
pred  ::=  pred | pred          -- OR
      ::=  pred & pred          -- AND
      ::=  ^  pred              -- negation
      ::=  expr  relop  expr    -- comparison of substring values
      ::=  'True'
      ::=  'False'
      ::=  (  pred  )

relop ::=  =                    -- equal
      ::=  /=                   -- not equal
      ::=  <                    -- lexically less
      ::=  >                    -- lexically greater
      ::=  <=                   -- lexically not greater
      ::=  >=                   -- lexically not less
      ::=  .=                   -- same marker ?
      ::=  .<                   -- precedes ?
      ::=  .>                   -- follows ?

expr  ::=  fnm  (  expr//, )    -- function call
```

```
      ::= expr ~ expr          -- concatenation
      ::= string-constant
      ::= var

var   ::= identifier
fnm   ::= identifier
```

Predicates may be parenthesized, but expressions may not.

A function call must have the same number of arguments as there are formal parameters in the function definition. If a function is called as part of an expression, the **return** statements in the function should return values.

The precedence of operators is

```
      ~        done first
relop
      ^
      &
      |        done last
```

Operators of equal precedence are evaluated from left to right. The predicate operators | and & are conditional; they do not evaluate their second argument if they need not.

The relational operators, = /= < > <= >=, compare the string values of their operands; they do not compare the markers themselves.

The dotted equal operator .= compares its operands to see if they delimit the same substring of the same base.

The dotted comparison operators .< .> compare the relative order of the start of the two operands. The comparison has the value True only if both operands are on the same base string and the start of the first has the indicated relation to the start of the second.

Concatenation, a ~ b , is defined to have the same value as

```
base(replace(end(replace(newbase(), a)), b))
```

4. Functions of the Substring Algebra

The predefined functions are those of the algebra for substring expressions, as described in [Algebra]. See that paper for details of their definitions. The arguments to all are substring expressions and all return substring values.

4.1. Primitive Functions

newbase() - Returns a marker for a non-constant empty string. This can be used to initialize variables which will later have a substring replaced or appended.

start(m) - Returns a marker for an empty substring at the beginning of m.

next(m) - Returns a marker for the single character following marker m. If m extends to the end of its base string, next(m) returns an empty substring at the end of m.

base(m) - Returns a marker for the entire base string of m.

extent(m1, m2) - Returns a marker for all characters of the base between start(m1) and start(next(m2)); that is, from the beginning of the first argument to the end of the second. When start(next(m2)) precedes start(m1), the result is an empty marker at start(next(m2)). If the two arguments are on distinct pseudo-bases, both of which are on the same ultimate base, the result is on the ultimate base. If the arguments are not on the same ultimate base string, the program aborts.

replace(m1, m2) - Returns a marker for a copy of m2 which has replaced m1 in its base string. Other markers on the same base as m1 are adjusted as though m1 had been appended to m1 and the latter then deleted (see [Algebra]). If the base of m1 is a constant string, the program aborts.

bound(m) - Returns a marker for the same substring as m. However, the result marker will behave as a base for other operations (except that extent() between a bounded marker and another marker outside the bound but on the same ultimate base will produce a marker on the ultimate base.)

readonly(m) - Returns a marker for the same substring as m. However, the result string will be readonly; modifications with replace() and ~:= will abort the program.

4.2. Substring Functions

end(m) - Returns a marker for the empty substring just after m.

```
function end(m) == return start(next(m));
```

rest(m) - Returns a marker for all characters of m other than the first. If m is empty, so is rest(m).

```
function rest( m ) == return extent(next(next(start(m))), m);
```

first(m) - Returns the first character of m. If m is empty, so is first(m).

```
function first(m) == extent(m, start(rest(m)));
```

second(m) - Returns a marker for the character following first(m), if there is one.

```
function second(m) == return first(rest(m));
```

advance(m) - Returns a marker extending from just after the first character of m to just after next(m). In general advance(m) will have the same number of characters as m, but it may be shorter if the end of m is at the end of the base string.

```
function advance(m) == return extent(second(m), next(m));
```

front(m) - Returns a marker for the first character after start(m), if there is one, otherwise m must be empty at the end of its base and this value is returned. Unlike first(), front() returns a character for all cases other than the very end of the base.

```
function front(m) == return next(start(m));
```

allprevious(m) - Returns a marker for the substring of the base of m that precedes the start of m.

```
function allprevious(m) == return extent(base(m), start(m));
```

allnext(m) - Returns a marker for the substring of the base of m that follow the end of m.

```
function allnext(m) == return extent(end(m), base(m));
```

last(m) - Returns the last character of m. If m is empty, so is last(m).

```
function last(m) == {
        marker t;
        if rest(m) = "" then return m;
        else return last(rest(m));
}
```

previous(m) - Returns a marker for the character preceding m. If m is at the beginning of it base string, previous(m) returns the value start(m).

```
function previous(s) == last(extent(base(s), start(s)));
```


4.3. Input/Output Functions

readfile(filename) - Returns a marker for the entire contents of the file named by the argument.

print(m) - Prints the text of *m* on stdout. Returns *m*.

4.4. Pattern Matching Functions

All the pattern matching functions have two arguments, a source string and a pattern. The source string is understood as extending from the start of the given argument to the end of its base. The pattern does not extend beyond either end of its marker. The value returned by a pattern matching operation for a successful search is a marker for the substring of the source that meets the criterion. For an unsuccessful search, the function returns a empty string at the beginning of the source.

search(m, target) - If successful, returns a marker surrounding the first substring of the source that is equal to *target*.

match(m, target) - Determines whether the source begins with a substring identical to *target*; if so, it returns a marker for that substring.

span(m, clist) - Returns a marker for all characters of the source from its beginning to just before the first character not in *clist*. That is, span matches the longest initial substring of the source that is composed of characters from *clist*.

anyof(m, clist) - Finds the first character in the source that is one of the characters in *clist* and returns a marker for the character.

token(m, clist) - Searches the source for the first substring consisting of characters from *clist* and returns a marker for the substring.

```
function token(m, clist) == return span(anyof(m, clist), clist);
```

The functions below describe with fair accuracy the present implementations of the pattern matching functions. However, FindChar is implemented in a handcoded C routine for speed.

```
-- FindChar(m, c) searches extent(m, end(base(m)))
--      for the character c.
--      For failure it returns end(base(m)).
```

```
function FindChar(m, c) == {
        m := front(m);
        while True do {
                if  m = ""  |  m = c  then
                        return m;
                m := next(m);
        }
}

function search(m, pat) == {
        marker pf, startm, tm, tp;
        startm := start(m);
        m := front(m);
        if pat = "" then return startm;
        pat := bound(pat);
        pf := front(pat);
        while m /= "" do {
                -- look for first character of pat
                m := FindChar(m, pf);
                if m = "" then
                        return startm;
                -- m is first char of pat
                -- if the rest of pat follows m
                --      then return pat at m;
                tm := m;
                tp := front(pat);
                while tm /= "" & tm = tp do {
                        tm := next(tm);
                        tp := next(tp);
                }
                if tp = "" then
                        return extent(m, start(tm));
                else m := next(m);
        }
        return startm;
 }

function match(m, pat) == {
        marker startm;
        startm := start(m);
        if pat = "" then return startm;
        m := front(m);
        while pat /= "" & m = first(pat) do {
                pat := rest(pat);
                m := next(m);
        }
        -- if reached end of pat, succeed
```



```
        -- note that m has changed
        if pat = "" then return extent(m, start(m));
        else return startm;
}

function span(m, clist) == {
        marker startm;
        startm := start(m);
        clist := bound(clist); -- to bound the Search
        m := front(m);
        while m != "" do {
                if FindChar(clist, m) = "" then
                        return extent(startm, start(m));
                m := next(m);
        }
        -- group extends to end of base(m)
        return extent(startm, m);
}

function anyof(m, pat) {
        marker startm;
        startm := start(m);
        pat := bound(pat);
        m := front(m);
        while  m != ""  do
                if  FindChar(pat, m)  then
                        return m;
                else m := next(m);
        return startm;
}
```

References

[Algebra] Hansen, W. J., A Practical Algebra for Substring Expressions, Computer Science Dept., Univ. of Glasgow, June, 1987.

[Em] Hansen, W. J., Em: Reference Manual, Computer Science Dept., Univ. of Glasgow, June, 1987.

Appendix A. Roman Numeral Conversions in Ness

```
-- xroman.mc

-- Roman Numeral conversion program

--      W. J. Hansen,  5 May 1987

--      (c) Copyright WJHansen, 1987

-- This program reads a file, converts all Roman numbers to decimal,
--      prints the result, converts are decimal values to Roman, and
--      prints the result again.

#include <stdio.h>


-- **********************************
-- *
-- *    Arabic to Roman
-- *
-- **********************************

marker TimesTable;      -- for multiplying Roman digits times ten
marker ArabicTable;     -- converts an Arabic digit to Roman
marker DecimalDigits;   -- the ten decimal digits

-- RMulTen(n) multiplies a Roman number s by ten, using table lookup
--      the argument s is modified in place
function RMulTen(n) == {
        -- for each letter of n, find the 10 times larger Roman letter
        n := first(bound(n));
        while n /= "" do
                n := next(replace(n, next(search(TimesTable, n))));
}
-- AtoR(s) converts an Arabic number in s to Roman numerals
function AtoR(s) == {
        -- for each digit of s, multiply the prior result by ten
        -- and append new digits
        marker r;
        r := newbase();
        s := first(bound(s));
        while s /= "" do {
                RMulTen(r);
                r ~ := span(next(search(ArabicTable, s)), "IVX?");
```

```
            s := next(s);
        }
        if search(r, "?") /= "" then return "?";
        return r;
}

-- ConvertArabic(s) scans the entire string s,
--      converting all Arabic numbers to Roman
--      Syntactically incorrect numbers are enclosed in "?"s.
function ConvertArabic(s) == {
        marker t;
        while s /= "" do {
                s := anyof(s, DecimalDigits);
                if  s /= "" then {
                        s := span(s, DecimalDigits);
                        t := AtoR(s);
                        if t = "?" then {
                                replace(start(s), "?");
                                replace(end(s), "?");
                                s := next(s);
                        }
                        else replace (s, t);
                        s := next(s);
                }
        }
}


-- ************************************
-- *
-- *    Roman to Arabic
-- *
-- ************************************

marker RomanDigits;     -- list of Roman numeral letters
marker RomanTable;      -- conversion of Roman to Arabic digits

-- SimpleRtoA(n)    Interprets n as a roman numeral
--      and returns the decimal equivalent.
--      This version assumes that the input is correct
--      and starts at the beginning of the input string.
--      No zero suppression is done for the result.
function simpleRtoA(n) == {
        marker result, d;
        result := newbase();
        d := front(RomanDigits);  -- first search for M's at start of n
        while d /= "I" do {
```



```
                n := span(n, d);
                result ~ := next(search(RomanTable, n));
                n := next(n);
                d := extent(last(d), next(next(d)));
        }
        return result;
}

-- RtoA(n)    Interprets n as a Roman number
--      and returns the decimal equivalent.
--      This version scans the input for the beginning
--      of the Roman number and checks its syntax.
--      The result is zero-suppressed.
--      If the input is syntactically incorrect,
--      a "?" is returned as the sole result.
--      A hack is used to shorten the search for each successive step.
function RtoA(n) == {
        marker result, d, t;
        result := newbase();
        n := anyof(n, RomanDigits);       -- skip to find Roman chars
        d := front(RomanDigits);
        t := RomanTable;
        while d /= "I" do {
                n := span(n, d);
                if n = "" then
                        result ~ := "0";
                else {
                        t := search(t, n);
                        if t = "" then return "?";
                        t := next(t);  -- the Arabic translation
                        result ~ := t;
                }
                n := next(n);
                d := extent(last(d), next(next(d)));
        }
        if search(RomanDigits, front(n)) /= "" then
                return "?";   -- trailing syntax error
        -- by construction, the result cannot be zero so
        -- zero-suppression need only look for first non-zero digit
        return extent(anyof(result, "123456789"), result);
}

-- ConvertRoman(s) scans the entire string s,
--      converting all Roman numbers to decimal
--      Syntactically incorrect numbers are enclosed in "?"s.
function ConvertRoman(s) == {
        marker t;
```

(No p. -19-)

```
        while s /= "" do {
                s := anyof(s, RomanDigits);
                if  s /= "" then {
                        s := span(s, RomanDigits);
                        t := RtoA(s);
                        if t = "?" then {
                                replace(start(s), "?");
                                replace(end(s), "?");
                                s := next(s);
                        }
                        else replace (s, t);
                        s := next(s);
                }
        }
}


-- ************************************
-- *
-- *    Main Routine
-- *
-- ************************************

function main(args) == {
        marker filenameletters, file, filename;

--      The order of entries in each of these tables is significant
        RomanDigits := "MDCLXVI";
        RomanTable := "0 M1 MM2 MMM3 MMMM4"
                ~ "C1 CC2 CCC3 D5 CD4 DC6 DCC7 DCCC8 CM9"
                ~ "X1 XX2 XXX3 L5 XL4 LX6 LXX7 LXXX8 XC9"
                ~ "I1 II2 III3 V5 IV4 VI6 VII7 VIII8 IX9";
        TimesTable := "IXCM?? VLD? ";
        ArabicTable := "? 0  1I 2II 3III 4IV 5V 6VI 7VII 8VIII 9IX ";
        DecimalDigits := "0123456789";


        filenameletters := "qwertyuiopasdfghjklzxcvbnm"
                ~ "QWERTYUIOPASDFGHJKLZXCVBNM"
                ~ ".0123456789";
        filename := span(next(token(args, " ")), filenameletters);

        file := readfile(filename);
        ConvertRoman(file);
        print(file);
        ConvertArabic(file);
        print(file);

}
```


Appendix B. Compiling, Linking, and Executing Ness Programs

The compiler for Ness is called *mc* and source program files must have the extension ".mc", say prog.mc. The result of compiling a source file is two files with the extensions ".c" and ".cons". The first of these is input to the C compiler and the second is read at the beginning of execution to initialize the constants vector. Separate compilation of routines will not work because the string constants are initialized from .cons in main().

In what follows I assume that the system files needed for *mc* reside in /usr/local/mc. In case they are elsewhere, appropriate adjustments will be needed. These required files are:

mc	the mc translator
mc.cons	run-time initialization for mc
em.h	header file of declarations used by the .c file
libem.a	library of support functions

Some other files are provided for reference:

mc.mc	compiler source code
NessRef.d	this manual
EmRef.d	describes the Em subroutine package
Algebra.d	definition of the substring algebra

The *mc* compiler is itself written in Ness, so the constants initialization file mc.cons is needed to execute the compiler. Copy this file to the directory containing the file to be compiled:

```
cp /usr/local/mc/mc.cons .
```

This need only be done once. Now the program itself can be compiled:

```
/usr/local/mc/mc  prog.mc
```

The resulting C program is then compiled and linked:

```
cc -g -o prog prog.c -I/usr/local/mc /usr/local/mc/libem.a
```

Finally the program itself is executed, passing whatever args are desired:

```
prog args
```

If you copy prog to another directory or an offline medium, remember to copy and transport the prog.cons file as well as the binary.

Diagnostics.

Mc is a research tool rather than a production language; it offers only two error messages:

adding a) in call to function-name
A semi-colon has been encountered before a closing right parenthesis.

adding " to end constant "constant-text"
A newline has been found in a constant string.

Other errors usually generate incorrect C code, which generates an error during the C compilation.

During the C compilation the messages about "statement not reached" should be ignored. They arise because mc always generates a "return" statement even though a function may already end with one.

Appendix C. Inclusion of C code

Since the mc processor is a preprocessor for C and since mc does not provide all data types, provision is made for inclusion of C code within an mc program:

Outside of functions, text is passed over unchanged until meeting one of the keywords "marker" or "function". The text after the first of these extends to the the next semi-colon; the text after the second extends to the end of the subsequent statement.

There are some restrictions on inclusions after "marker". It is safest to include nothing but comments.

Within function definitions dollar signs ($) are deleted and subsequent text is passed through unchanged. If a dollar sign is not followed by a left square bracket, the dollar sign is deleted and the unchanged text extends to the end of the line. If the dollar sign is followed by a left square bracket, both are deleted and the text is then left unchanged up to the next right square bracket, which is deleted. Formal and actual parameter lists are processed so inclusions with $[. . .] are left in the parameter list so they are passed by the call rather than as values on the marker stack.

Lines for the C preprocessor begin with hash signs (#); these and the remainder of the line on which they appear are passed through to the compiler undisturbed.

Mc converts Ness functions into C functions which accept their arguments from a stack and store their result there. Consider this sample program:

```
marker glob;
function fun (a, b)
        text between right parenthesis and double equal
 == {
        marker c;
        -- assignment
        c := start(a);
        return b;
}
function main(args) == {
        glob := "global string constant";
        print(fun("for a", glob));
}
```

For this sample, the output of mc is:

```
#include "em.h"
#define EmQNConsts 2
Emark EmQCons[EmQNConsts+1];
Emark glob;

fun ( )
        text between right parenthesis and double equal
{

#define a 1
#define b 2
#define c 3

        EmFrame(2, 1); {
        ;
        /* assignment */
        (EmLoadLocal(a), EmStart()), EmStoreLocal(c);
        { EmLoadLocal(b);EmResult(0);}
}
#undef a
#undef b
#undef c
EmLoad(EMPTY); EmResult(0);

main(EmQargc, EmQargv) char **EmQargv;

{
        EmInit();
        EmQLoadConsts("t.cons", EmQCons, EmQNConsts);
        EmQConvertArgs(EmQargc, EmQargv);

#define args 1

        EmFrame(1, 0); {
        EmLoad(EmQCons[1]), EmStore(glob);
        ((EmLoad(EmQCons[2]), EmLoad(glob),fun()), EmDup(),
                EmPrint(stdout)),EmPop();
}
#undef args
EmLoad(EMPTY); EmResult(0);
}
}
```


The main program calls EmInit to initialize the Em package, EmQLoadConsts to initialize the constant string table, and EmQConvertArgs to transform the command line to the mc form. Global variables need not be initialized because they are initialized by C to NULL, which is understood as a valid former value by EmStore(). The value of EmQConvertArgs remains on top of the stack as the first argument for the EmFrame.

Useful C Routines

The following routines may be of interest when writing C inclusions:

EmLoadLocal(name), EmLoad(variable). These functions load a value onto the top of the stack. EmLoadLocal is for names specified with #define's; all other values should be accessed via EmLoad().

EmStoreLocal(name), EmStore(variable). These functions pop the stack and store the value removed. EmStoreLocal is for names specified with #define's; all other values should be accessed via EmStore(). Before the first EmStore into a non-local variable, it should be set to zero.

EmDiscard(variable). When the value of a global variable is no longer needed, it and it associated string storage can be released by passing the variable to EmDiscard().

EmString(C-string). The C-string is a normal C reference to a string. An Em string for the same value is created and a marker for it is pushed onto the stack.

EmToC(). The top value is popped from the stack and copied into a buffer, terminated with a '\0'. The address of this buffer is returned and can be passed to C routines. The buffer is re-used each time this function is called. (The marked value must be 499 bytes or shorter.)

EmWrite(fd). The top value is popped from the stack and written to the specified file descriptor.

Boolean EmRead(fd, len). Reads up to len bytes from the file attached to descriptor fd and leaves on the marker stack a marker for the bytes. If len is 0, the entire remaining file is read. Returns False if end of file or error occurs.

long EmPeek(). Returns the long value corresponding to the first character of the marker on top of the stack. Returns zero if the marker was empty.

EmChr(v). Converts the long value *v* to a character, makes a base

string containing that character, and leaves the base string atop the marker stack.

Appendix D. Compiling the Compiler

The mc compiler for Ness is in file mc.mc . Thus compiling it is similar to compiling any other mc program. The important difference is that the new version may be sufficiently incorrect that the compiler could no longer compile itself. Therefore earlier versions must be retained. I generally utilize three versions of mc: ok, new, and test.

ok. The ok version is the last one known to work well; it compiles the compiler and the suite of example programs. This version has four files: mc.ok, the binary; mc.ok.mc, the source; mc.ok.c, the output of the compile; and mc.ok.cons, the run-time constants.

new. This version is the result of compiling mc.mc using the ok compiler. There are mc.new, mc.new.c, and mc.new.cons.

test. This version is the result of compiling mc.mc using the new compiler. There are mc.test, mc.test.c, and mc.test.cons.

Here is a typical compile and test sequence:

```
# construct new compiler
cp mc.ok.cons mc.cons      # set up proper .cons file
mc.ok mc.mc          # do the compile
mv mc.c mc.new.c; mv mc.cons mc.new.cons   # save the results
cc -g -o mc.new mc.new.c -I/usr/local/mc/em.h \
        /usr/local/mc/libem.a

# use new compiler to compile itself, making the test version
cp mc.new.cons mc.cons    # set up proper .cons file
mc.new mc.mc              # do the compile
mv mc.c mc.test.c; mv mc.cons mc.test.cons     # save the results
cc -g -o mc.test mc.test.c -I/usr/local/mc/em.h \
        /usr/local/mc/libem.a

# execute the test version and check its output
cp mc.test.cons mc.cons    # set up proper .cons file
mc.test mc.mc              # do the compile
diff mc.c mc.test.c        # these two diffs should both indicate
diff mc.cons mc.test.cons  #      NO DIFFERENCES
```


When all is well, the new compiler can be made the ok version:

```
# save the ok version (this is superstition)
mv mc.ok.mc mc.old.mc
mv mc.ok.c mc.old.c
mv mc.ok.cons mc.old.cons

# set up new version
mv mc.new mc.ok
mv mc.new.c mc.ok.c
mv mc.new.cons mc.ok.cons
cp mc.mc mc.ok.mc
```

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, N0. 2, (Summer 1986) - see PPRR-23-86.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", Software - Practise and Experience, Vol. 17(3), 171-185 (March 1987)

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", Software Tool 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 9th July 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-87	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00

PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00
PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00