The Y Programming Language†

David R. Hanson

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721

1. Introduction

Y is a structured, general-purpose programming language intended for use in simple systems programming applications. More specifically, it is designed for applications similar to those described in the book *Software Tools* [ker76]. Y is, in fact, meant to replace Ratfor [ker75] for those sorts of applications and in programming courses based on *Software Tools*.

Y is a relatively simple language. Syntactically, it falls about midway between Ratfor and C [ker78]. Semantically, it leans towards C except that it does not support all of the C types. Y programs are collections of modules, which contain global and local static data and procedures. Procedures are recursive and are composed of local data declarations and statements. Statements are made up from the usual structured control flow constructs and expressions. Y supports integers, characters, and reals, and singly dimensioned arrays of them.

In addition to the intended application areas mentioned above, Y is the experimental realization of some recent ideas concerning separate compilation [han79b] and block structure [han80a]. It also provides a testbed for experimental work in program portability and code optimization [dav80,fra79,han80b]. The remainder of this paper describes the syntax and semantics of Y and illustrates its use.

1.1 Syntax Notation

Where possible, the syntax of Y is described informally using English prose. Where the syntax is more complicated, a formal metalanguage is used in which syntactic classes are denoted by *italic* type and literal characters and symbols are denoted by **bold** type. Alternatives are separated by vertical bars (|) or are listed on separate lines. Optional items are enclosed in brackets ([ ]), and ellipses (...) indicate indefinite repetition of the item they immediately follow.

In cases where the literal use of bars, brackets, and periods is not clear in context or conflicts with their metalinguistic use, they are enclosed in quotes. Program examples are given in a sans-serif type.

†This work was supported by the National Science Foundation under Grant MCS7802545.

2. Lexical Structure

Y programs are composed of identifiers, reserved words, constants, operators, and other separators. The 'official' character set of Y is ASCII [ans77]. Blanks and tabs are ignored but, unlike Fortran, are required if necessary to separate some lexical elements such as identifiers and reserved words; e.g. integera and integer a are not equivalent.

2.1 Reserved Words

Reserved words introduce language constructs and may not be used for other purposes (e.g. as a variable name). Reserved words must be given in lower-case. The reserved words are

break	**for**	**next**
case	**fortran**	**real**
character	**from**	**repeat**
char	**if**	**return**
default	**import**	**switch**
else	**integer**	**to**
export	**int**	**until**
end	**module**	**while**

2.2 Identifiers

Identifiers name language elements such as procedures and variables. An identifier is a sequence of letters, digits, or underscores that begins with a letter. Corresponding upper- and lower-case letters are treated as different. Identifiers may be of any length, but some implementations may use only the first 5 to 8 characters internally.

2.3 Integer and Character Constants

Integer constants are denoted by sequences of digits in the usual manner. If a leading 0 is specified, the constant is assumed to be given in octal.

Single character constants are treated as integers with numerical values corresponding to their ASCII code. A character constant is specified by enclosing the desired character in single quotes, e.g. 'x'. Some characters, such as the single quote, cannot be entered directly because of their special function. The following escape convention may be used to enter these kinds of characters.

character	*code*
newline	\n
single quote	\'
double quote	\"
backslash	\\
tab	\t
any character	*ddd*

The specification *ddd* represents the character with ASCII code octal *ddd*; only enough digits to specify the code need be given.

2.4 Real Constants

Real constants are specified in the standard fashion except that exponential notation (e.g. 3.45e10) is not supported. For magnitudes less than 1, a leading 0 is required.

2.5 String Constants

String constants are specified by delimiting the sequence of characters by double quotes ("). Single quotes may be used for strings of 2 or more characters. Special characters, including quotes, may be specified using the escape convention described above. In addition to the specified characters, a null character (ASCII code 0) is placed at the end of each string by the compiler.

2.6 Comments

The sharp character (#) causes the rest of the line on which it appears to be ignored and therefore serves to introduce comments.

3. Program Structure

A Y program consists of one or more *modules*. A module is simply a file whose contents have the general form

module:
  module *identifier*
    [*import/export-declaration*]...
    [*variable-declaration*]...
    [*procedure-declaration*]...
    end

Executable portions of a program appear only within procedures. Program execution begins by invoking the procedure named main, which must appear in only one module.

Generally speaking, language constructs, such as declarations, statements, and expressions, are terminated by the end of the line on which they appear much as in Ratfor and Icon [gri80]. Within a construct, however, newlines may be used as desired to improve readability provided it it is obvious that the construct is continued on the next line.†

3.1 Scope and Import/Export Declarations

Unless specified otherwise, the scope of all variables and procedures declared within a module is restricted to that module. Static communication among separately compiled modules—accessing variables and procedures declared in another module—is indicated by import and export declarations [han79b]:

import/export-declaration:
  import *identifier* [, *identifier*]... **from** *string-literal*
  export *identifier* [, *identifier*]... **to** *string-literal*

These declarations cause the compiler to access the file whose name is given by the string literal and read or write information about the listed identifiers. These files, called *description* files, are constructed and maintained by the compiler; they are not meant to be edited by programmers. By restricting access to description files, programmers have some control over the sharing of variables and procedures among separately compiled modules.

The **import** declaration lists those variables and procedures that are referenced in the current module but defined in another. The compiler reads the characteristics of the identifiers, such as the type, from the description file, provided it is accessible. References to the identifiers are external references, which are resolved during linking, but the type checking associated with operations on them is performed during compilation.

The **export** declaration lists those variables and procedures that are defined in the current module but may be referenced in another. After compiling the module, the compiler writes the characteristics of the identifiers to the description file, provided it is accessible for writing. If an entry for an identifier already appears in a description file, it is overwritten by the new entry.

Note that the import/export mechanism cannot handle 'mutual' dependencies. For example, suppose module a contains

```
export f to "f1"
import g from "f2"
f(...)
    ...
    g(...)
    ...
end
```

and module b contains

```
export g to "f2"
import f from "f1"
g(...)
    ...
    f(...)
    ...
end
```

Module a must be compiled before b and vice versa. The solution is to place both f and g in one module. Like most high-level language facilities, the import/export scheme imposes a particular structure on programs that does not suit some cases, just as most structured control statements and type systems do not cater well to every possible situation.

3.2 Variable Declarations

Variables that are declared outside of any procedure are static and are global to the module in which they are declared. The syntax is

† Technically, a newline is treated as white space except at points where it is in the *follow* set [aho77] of a construct, in which case it signals the end of the construct.

variable-declaration:
  type global-declarator [, *global-declarator*]...

global-declarator:
  identifier
  identifier '[' *integer-literal* ']'

type:
  character
  char
  integer
  int
  real

As indicated, scalars and arrays of type **character**, **integer**, or **real** may be declared. **Character** scalars are equivalent to **integer** scalars, however. For arrays, the integer literal gives the number of elements in the array, and the array bounds are from 1 to that number. A reference to an element of a **character** array (e.g. s[i]) has type **integer**.

3.3 Procedure Declarations

Procedures are declared as follows.

procedure-decl:
  [*type*] *identifier* ([*identifier* [, *identifier*]...])
    [*local-declaration*]...
    [*statement*]...
    end

local-declaration:
  type local-declarator [, *local-declarator*]...

local-declarator:
  identifier
  identifier '[' [*integer-literal*] ']'
  identifier ( )

Procedures are recursive.

If the type of a procedure is omitted, it is typeless and cannot be used in a context requiring a type. It is essentially a subroutine. Procedures of type **character** are equivalent to procedures of type **integer**.

Communication among procedures is via arguments or global variables (see above). Transmission of actual arguments is by value for scalars and by reference for array names (see below).

The declarations for local variables within a procedure must include specifications for the formal parameters. For array parameters, the array size is ignored since the storage for an array transmitted as an actual argument is allocated by the caller. It is normally unnecessary to declare other procedures referenced within a procedure. This is the case if the referenced procedure appears in the module before the procedure in which it is used, is imported, or is used in a context that does not require a type. If the type is required and the referenced procedure is as yet undefined, a local declaration of the indicated form may be given to specify the type. Such declarations must be consistent with the subsequent procedure declaration.

4. Expressions

Expressions compute values. Expression evaluation proceeds according the the precedence and associativity of the operators involved. Evaluation is generally left-to-right, but the precise order is undefined except in a few cases. Operator precedence and associativity is summarized in the following table.

operators	*associativity*	*precedence*
=	*right-to-left*	*lowest*
|	*left-to-right*	
&	*left-to-right*	
== ~= < <= >= > >> <<	*left-to-right*	
+ −	*left-to-right*	
* / %	*left-to-right*	
~	*unary*	*highest*

Parentheses may be used as usual to override the built-in precedence and associativity rules.

4.1 Variables

The most basic expression refers to a variable—either a scalar or an array element:

variable:
  identifier
  identifier '[' *expression* ']'

For an array reference, the type of the subscript expression must be **integer**. If it is not, the appropriate conversion is provided automatically. The type of a variable is determined by its declaration. The type of a reference to an element of a **character** array (e.g. a[i]) is **integer**.

4.2 Primary Expressions

The primary expressions are:

primary-expression:
  integer-literal
  real-literal
  string-literal
  variable
  identifier ([*expression* [, *expression*]...])
  (*expression*)

The type of a literal depends on its form as described in Sec. 2. The type of a parenthesized expression is the type of the expression itself. The type of a procedure call is determined by the type given in the procedure declaration. It is permissible to have a procedure without a type, providing it is never used in a context that requires one. If a procedure name is undeclared, it is assumed to be a procedure without a type, which will presumably be declared in a subsequent procedure declaration. If a type is required, a local declaration for the procedure may be given (see Sec. 3.3).

The actual arguments to a procedure are evaluated in an unspecified order. For scalar variables and expressions, *copies* of the actual argument values are passed to the procedure. For expressions consisting of only an array name, the address of the array is passed. Thus, argument transmission is by *value* for scalars and by *reference* for arrays. Note that, unlike Fortran (and Ratfor), an array reference such as a[i] is a scalar. It is not possible, therefore, to pass portions of an array to a procedure. Actual argument types and the number of arguments are *not* checked for consistency with the formal parameters given in the procedure declaration.

4.3 Unary Operators

unary-expression:
 – expression
 + expression
 ~ expression

The unary – and + operators denote negation and affirmation, respectively. Negation has its usual arithmetic meaning and affirmation is a null operation. For both operators, the type of the result is the type of the operands. If the types of both operands are the same (**integer** or **real**), the type of the result is the type of the operands. For 'mixed mode' usage, **integer** operands are converted to **real**.

The unary ~ operator returns the ones-complement of its operand. The type of the result is **integer**, but *no* conversion of the operand is performed.

4.4 Multiplicative Operators

multiplicative-expression:
 *expression * expression*
 expression / expression
 expression % expression
 expression << expression
 expression >> expression

The binary * and / operators denote multiplication and division, respectively. If the types of both operands are the same (**integer** or **real**), the type of the result is the type of the operands. In the case of 'mixed mode' usage, **integer** operands are converted to **real** and the result is **real**.

In integer division, the result is truncated as in Fortran. No check for division by 0 is made; the result in that case is machine-dependent.

The binary % operator denotes the residue operation. The result is an **integer** and is the remainder of the first expression divided by the second. The operands of % must be **integer**; the appropriate conversions are performed automatically if they are not.

The binary << and >> operator denote the left and right shifting, respectively. The result is an **integer**. The first expression may be either **integer** or **real**; *no* conversion is performed. The second expression must be **integer**; the appropriate conversions is performed automatically if it is not. For both operators, the value of the first expression is intepreted as a bit pattern and is shifted by the amount given by the second expression. For left shifting, vacated bits are filled with zeros. For right shifting, the value of vacated bits is undefined.

4.5 Additive Operators

additive-expression:
 expression – expression
 expression + expression

The binary – and + operators denote subtraction and addition, respectively. If the types of both operands are the same (**integer** or **real**), the type of the result is the type of the operands. For 'mixed mode' usage, **integer** operands are converted to **real** and the result is **real**.

4.6 Relational Operators

relational-expression:
 expression < expression
 expression <= expression
 expression == expression
 expression ~= expression
 expression >= expression
 expression > expression

The relational operators are < (less than), <= (less than or equal), == (equal to), ~= (not equal to), >= (greater than or equal to), and > (greater than). They all yield an **integer** result: 0 if the relation is false, 1 if it is true. If the types of the operands are not the same (**integer** or **real**), **integer** operands are converted to **real**.

4.7 Logical Operators

logical-expression:
 expression '|' expression
 expression & expression

The binary | and & operators denote inclusive OR and AND, respectively. When used in a context requiring a value, | returns the bitwise inclusive OR of its operands and & returns the bitwise AND of its operands. The type of the result is **integer**. Any combination of **integer** and **real** operands is permissible; *no* conversions are performed.

When the | and & operators are used in a context that does not require a value, such as in the conditional expression in an if, while, or for statement, one may not be generated. More importantly, in expressions involving several | and & operators, only enough of the expression to determine the ultimate truth value (zero or non-zero) may be evaluated. For example, in

```
if (f(x) | g(x)) x = 0
```

it is undefined whether both f and g are invoked.

4.8 Assignment Operator

assignment-expression:
 variable = expression

The binary = operator denotes assignment. The value of the expression is stored in the location denoted by the variable. The value is converted, if necessary, to the type associated with the variable. The value of the expression (after conversion) becomes the result of the = operator.

The = operator associates to the right, permitting multiple assignments, e.g.,

```
a = b = c = 6
```

associates as in

```
a = (b = (c = 6))
```

Evaluation of a single assignment is defined to be left-to-right so that, for example, in

```
a[i] = f(x,y)
```

the value of i *before* the invocation of f is used to index into a. Note that this rule is for single assignments only; the order of evaluation of the variables involved in a multiple assignment is undefined. Thus, in

```
a[i] = i = i + 1
```

it is undefined whether the value of i before or after it is incremented is used to index a.

4.9 Conversions

As indicated above, conversions between **integer** and **real** values may be performed in certain circumstances. Such conversions are provided automatically as appropriate.

Conversion from an integer to a real value corresponds to the 'float' operation in Fortran. Note that, on some machines, some precision may be lost in converting large integers to real values.

Conversion from a real value to an integer corresponds to the 'fix' operation in Fortran. Specifically, the real value is truncated to its integral part. If the result is not within the range of integers, the result is undefined. In addition, the direction of truncation of negative real values is undefined since it seems to be very machine-dependent.

Despite the machine-dependent aspects of conversion, it is intended that the results in Y be similar, if not equivalent, to the results of the corresponding operation in Fortran.

5. Statements

Statements are executed sequentially in the order in which they appear. Various control structures provide for other orders of execution.

As mentioned above, statements are usually terminated by the end of the line on which they appear. In most cases, however, statements may be spread out over several lines for readability provided they are broken at points where it is obvious that they are continued on subsequent lines.

5.1 Expression Statement

Most statements are simply expressions:

expression-statement:
  expression

Typical expression statements are assignment expressions and procedure calls.

5.2 Null Statement

A lone semicolon is treated as a null statement:

null-statement:

Null statements are sometimes used as the body of loops in cases where an empty body is needed. The null statement is the one case in which a statement may immediately follow another without an intervening newline. As such, semicolons may be used to place several statements on the same line, e.g.

```
a = 2; f(a, b); b = a + 1
```

5.3 Compound Statement

The compound statement permits several statements to be grouped together as one statement:

compound-statement:
  { *statement* [*statement*]... }

5.4 if Statement

The if statement is the basic conditional statement and permits a one-, two-, or multi-way branch on the result of an expression:

if-statement:
  if (*expression*) *statement*
  if (*expression*) *statement* **else** *statement*

In both forms, the expression is evaluated and if the result is non-zero the first substatement is executed. If the **else** is specified, a zero result causes execution of the second substatement.

The familiar 'dangling else' ambiguity in nested if statements is resolved by associating an **else** with the closest **if** that does not have an **else**. For example, in

```
if (a >= 0)
    if (a > b)
        max = a
    else
        max = b
```

the **else** is associated with the second **if**. A compound statement may be used to obtain alternate interpretations, e.g.

```
if (a >= 0) {
    if (a > b)
        max = a
    }
else
    max = b
```

Using an if statement as the substatement following an **else** is a general way of writing a multi-way decision and corresponds to a linear search. The general form is

  if *(expression)*
    statement
  else if *(expression)*
    statement
  else if *(expression)*
    statement
  else
    statement

The conditional expressions are executed in the order given and the first non-zero result causes the execution of the associated statement and termination of the search. If none of the expressions yields a non-zero result, the statement associated with the last **else** is executed. Note that this latter statement, which corresponds to a 'default' case, is optional.

5.5 switch Statement

The switch statement is similar to the if statement in that it permits a multi-way branch on the result of an expression. The important difference is that only *constants* may be compared with the resulting value to control flow. It is, therefore, a special case of the **if-else** chain described in the previous section.

switch-statement:
  switch (*expression*) {
    [**case** *case-label* [, *case-label*]... : [*statement*]...]...
    [**default** : [*statement*]...]
    }

case-label:
  [– | +] *integer-literal*
  [– | +] *integer-literal* .. [– | +] *integer-literal*

The expression is evaluated and the resulting value is compared to all of the cases. Execution continues with the statement sequence that follows the case containing the resulting value. Upon completion of that statement sequence, execution continues after the switch statement.

Note that this behavior is different than in C where control falls through to the next case.

If the value of the expression does not appear in any of the case lists, execution continues with the statement sequence that follows the case labeled **default**. The default case is optional; if it is omitted and the expression value does not appear in any case list, execution continues after the switch statement. The default case may appear anywhere within the switch statement, but only once.

5.6 while Statement

In the while statement

while-statement:
while (*expression*) *statement*

the expression is repeatedly evaluated until it yields a zero result. The substatement is executed after each evaluation that resulted in a non-zero value. Note that the result of the expression is tested *before* the substatement is executed. Thus, if the initial evaluation of the expression yields zero, the substatement is never executed.

5.7 repeat Statement

In the repeat statement

repeat-statement:
repeat *statement*
repeat *statement* **until** (*expression*)

The substatement is executed repeatedly, provided that after each execution, the expression yields a non-zero value. Note that the result of the expression is tested *after* the substatement is executed. Thus, the substatement is always executed at least once.

The **until** portion of the repeat statement is optional in which case the repeat statement is a non-terminating loop. In this case, the loop can be terminated by other means, e.g. via a break or return statement.

5.8 for Statement

The for statement

for-statement:
for ([*expression1*] ; [*expression2*] ; [*expression3*])
statement

is equivalent (in the absence of next statements) to

```
expression1
while ( expression2 ) {
    statement
    expression3
    }
```

In typical usage, *expression1* and *expression3* are assignments or procedure calls, and *expression2* is a conditional expression. For example,

```
sum = 0
for (i = 1; i <= 10; i = i + 1)
    sum = sum + a[i]
```

computes the sum of the elements of an array. The expressions in the for statement can, of course, be arbitrary expressions. For example,

```
for (c = getc(); c == ' ' | c == '\t'; c = getc())
    ;
```

reads the standard input until the first non-blank character, which is left in c. Note the use of the null statement as the loop body.

All of the expressions in the for statement are optional. If they are omitted, the meaning of the statement is identical to the corresponding expansion in terms of the while statement. Note that omitting all three expression yields

for (;;)
statement

which is a non-terminating loop. In this case, the loop can be terminated by other means, e.g. via a break or return statement.

5.9 break and next Statements

The following statements are used to alter the flow of control within loops:

break-statement:
break

next-statement:
next

The break statement causes immediate termination of the innermost loop (e.g. **while**, **repeat**, or **for**) in which it appears. Execution continues with the statement following the loop. Note that only the innermost loop is terminated, even if **break** appears in a nested for, repeat, or while statement.

The next statement causes immediate transfer to the 'next iteration' point of the innermost loop in which it appears. For a while statement, this point corresponds to the beginning of the conditional expression, i.e. to the 'top' of the loop. For a repeat statement, it corresponds to the beginning of the **until** portion of the statement, i.e. to the 'bottom' of the loop. For a repeat statement without an **until**, **next** causes a transfer to the beginning of the substatement. For a for statement, control is transferred to the beginning of its *expression3*.

5.10 return Statement

The return statement is used to transfer control from a procedure to its caller:

return-statement:
return
return (*expression*)

If an expression is given, it is evaluated and the result is transmitted to the caller of the procedure as the result of the procedure call. If necessary, the returned value is converted to the type of the procedure in which it appears. In the case of a bare **return**, the returned value is undefined. An implicit return statement is supplied at the end of each procedure so that flowing off the end of a procedure causes a return (with an undefined value).

6. Programming Examples

The following examples illustrate the use of Y. Most of them are taken from similar examples in Ratfor and C. It is assumed that the i/o routines described in *Software Tools* and in [han79a] are available. In addition, some of the examples use defined constants (e.g. EOF), which are handled by processing the Y source with **macro** prior to compilation (see Chap. 7 of *Software Tools*).

6.1 Word Counting

The following program counts the number of lines, words, and characters in its input. It is a simple version of the *wc* utility on UNIX [rit74] and is described in both [ker76] and [ker78].

```
# wc - count lines, words, and characters in input
module wc

    define(EOF,(-1))

    import printf, getc from "ylib.d"

    main()
        integer c, nw, nl, nc
        integer inword

        nl = nw = nc = 0
        inword = 0
        while ((c = getc()) ~= EOF) {
            nc = nc + 1
            if (c == '\n')
                nl = nl + 1
            if (c == ' ' | c == '\n' | c == '\t')
                inword = 0
            else if (inword == 0) {
                inword = 1
                nw = nw + 1
                }
            }
        printf("%d %d %d\n", nl, nw, nc)
    end
end
```

6.2 8 Queens

The 8-queens problem is commonly used (and over-used) as an example of backtracking (cf. [wir76], Sec. 3.5). The object is to determine all of the ways 8 queens can be placed on a chess board so that no queen can take any of the others. The following recursive solution prints all 92 solutions (although only 12 are unique).

```
module eightqueens

    import putc from "ylib.d"

    integer up[15]      # up-facing diagonals
    integer down[15]    # down-facing diagonals
    integer rows[8]     # rows
    integer x[8]        # holds solution

    main()
        integer i

        for (i = 1; i <= 15; i = i + 1)   # free the board
            up[i] = down[i] = 1
        for (i = 1; i <= 8; i = i + 1)
            rows[i] = 1
        queens(1)  # place 1st and subsequent queens
    end
    queens(c)
        integer r, c

        for (r = 1; r <= 8; r = r + 1)
            if (rows[r] & up[r-c+8] & down[r+c-1]) {
                rows[r] = up[r-c+8] = down[r+c-1] = 0
                x[c] = r# record solution so far
                if (c == 8)
                    print()
                else
                    queens(c + 1)
                rows[r] = up[r-c+8] = down[r+c-1] = 1
                }
    end
    print()
        integer k

        for (k = 1; k <= 8; k = k + 1) {
            putc(' ')
            putc('0' + x[k])
            }
        putc('\n')
    end
end
```

6.3 Pocket Calculator

The following program simulates a simple reverse Polish pocket calculator. It is similar to the program described in Sec. 4.4 of [ker78], but includes a facility for storing values and operates on real values. Input consists of numbers, single-letter variable names, and operators. This example also illustrates separate compilation and the use of modules for information hiding. In the stack module, only the procedures clear, dump, pop, and push are exported. The representation of the stack is hidden within its module.

```
# dc - reverse polish pocket calculator
module dc

    define(EOF,(-1))

    import getc, printf from "ylib.d"
    import push, pop, clear, dump from "stack.d"

    real variables[26]  # variable storage
    integer peek        # pushed back character

    main()
        integer c, i, ngetc(), getvar()
        real t, getnum()

        clear()
        peek = 0
        for (i = 1; i <= 26; i = i + 1)
            variables[i] = 0
        while ((c = ngetc()) ~= EOF)
            switch (c) {
                default:
                    printf("%c ?\n", c)
                case ' ', '\t', '\n':
                case 'a'..'z':
                    push(variables[c - 'a' + 1])
                case 'A'..'Z':
                    push(variables[c - 'A' + 1])
                case '0'..'9', '.':
                    ungetc(c)
                    push(getnum())
                case ':':
                    if (i = getvar())
                        variables[i] = pop()
                case '+':
                    push(pop() + pop())
                case '-':
                    t = pop()
                    push(pop() - t)
                case '*':
                    push(pop() * pop())
                case '/':
                    t = pop()
                    if (t ~= 0.0)
                        push(pop() / t)
                    else
                        printf("division by 0\n")
```

```
            case '=':
                printf("\t%f\n", push(pop()))
            case '?':
                printf("stack =")
                dump()
                printf("\n")
                for (i = 1; i <= 26; i = i + 1)
                    if (variables[i])
                        printf("%c = %f\n", i + 'a' - 1, variables[i])
            case ';':
                clear()
            case '!':
                pop()
            case '$':
                break
            }
end
# getnum - read and return number
real getnum()
    integer c, ngetc()
    real r, t

    r = 0.0
    for (c = ngetc(); c >= '0' & c <= '9'; c = ngetc())
        r = 10.0*r + c - '0'
    if (c == '.') {
        t = 1.0
        for (c = ngetc(); c >= '0' & c <= '9'; c = ngetc()) {
            r = 10.0*r + c - '0'
            t = 10.0*t
            }
        r = r/t
        }
    ungetc(c)
    return (r)
end
# getvar - get next variable name
integer getvar()
    integer c, ngetc()

    c = ngetc()
    while (c == ' ' | c == '\t')
        c = ngetc()
    if (c >= 'a' & c <= 'z')
        return (c - 'a' + 1)
    if (c >= 'A' & c <= 'Z')
        return (c - 'A' + 1)
    else {
        printf("%c ? variable name expected\n", c)
        ungetc(c)
        return (0)
        }
end
# ngetc - get next input character
integer ngetc()
    integer c

    if (peek)
        c = peek
    else
        c = getc()
    peek = 0
    return (c)
end
# ungetc - put a character back on input
ungetc(c)
    integer c

    peek = c
end
end
```

```
module stack

    define(MAXSTACK,20) # stack size

    import printf from "ylib.d"
    export clear, dump, pop, push to "stack.d"

    integer sp              # stack pointer
    real stack[MAXSTACK]    # stack
    # clear - clear stack
    clear()

        sp = 0
    end
    # dump - print contents of stack
    dump()
        integer i

        for (i = sp; i > 0; i = i - 1)
            printf("\t%f\n", stack[i])
    end
    # pop - pop top value from stack
    real pop()
        real x

        if (sp > 0) {
            x = stack[sp]
            sp = sp - 1
            }
        else {
            printf("? stack empty\n")
            x = 0
            }
        return (x)
    end
    # push - push x onto stack
    real push(x)
        real x

        if (sp >= MAXSTACK)
            printf("? stack full\n")
        else {
            sp = sp + 1
            stack[sp] = x
            }
        return (x)
    end
end
```

6.4 Word Frequencies

The following program computes the frequency of occurrence of the words in its input, treating upper- and lower-case letters as equivalent. It uses a binary tree to store the words and their associated counts. Note the use of recursion to locate and install words in the tree (lookup) and to print the tree (tprint).

```
# wf - print word frequencies
module wf

    import getc, putc, printf, exit from "ylib.d"

    # layout of tree nodes
    define(COUNT,0)   # number of times word appears
    define(LLINK,1)   # pointer to left subtree
    define(RLINK,2)       # pointer to right subtree
    define(WORD,3)    # pointer to word
    define(NODESIZE,4)

    define(MAXWORD,15)       # maximum word length
    define(TBUFSIZE,2000)    # size of node storage
    define(CBUFSIZE,2000)    # size of char storage
```

```
define(EOF,(-1))

integer tbuf[TBUFSIZE]     # holds trees
char cbuf[CBUFSIZE]        # holds chars
integer nexttbuf           # index of next free word in tbuf
integer nextcbuf           # index of next free char in cbuf
integer total              # total number of words

main()
    integer p, lookup(), getword()
    char word[MAXWORD]

    tbuf[1] = 0# root of the tree
    nexttbuf = 2
    nextcbuf = 1
    total = 0
    while (getword(word)) {
        p = lookup(word, 1)
        tbuf[p+COUNT] = tbuf[p+COUNT] + 1
        }
    tprint(tbuf[1])
end
# getword - get next input word into buf, return length
integer getword(buf)
    char buf[]
    integer i, c, isletter()

    while ((c = getc()) ~= EOF)
        if (isletter(c))
            break
    for (i = 1; c = isletter(c); c = getc())
        if (i < MAXWORD) {
            buf[i] = c
            i = i + 1
            }
    buf[i] = 0
    return (i - 1)
end
# isletter - return folded c if it is a letter, 0 otherwise
integer isletter(c)
    integer c

    if (c >= 'A' & c <= 'Z')
        c = c + 'a' - 'A'
    if (c >= 'a' & c <= 'z')
        return (c)
    else
        return (0)
end
# lookup - lookup word in tree; install if necessary
integer lookup(word, tree)
    char word[]
    integer tree, cond, p, strcmp(), strlen()

    if (p = tbuf[tree]) {
        cond = strcmp(word, 1, cbuf, tbuf[p+WORD])
        if (cond < 0)
            return (lookup(word, p + LLINK))
        else if (cond > 0)
            return (lookup(word, p + RLINK))
        else
            return (p)
        }
    else {  # new entry
        p = nexttbuf
        nexttbuf = nexttbuf + NODESIZE
        if (nexttbuf > TBUFSIZE) {
            printf("out of node storage\n")
            exit()
            }
        tbuf[p+COUNT] = 0
        tbuf[p+LLINK] = tbuf[p+RLINK] = 0
        tbuf[p+WORD] = nextcbuf
        nextcbuf = nextcbuf + strlen(word) + 1
        if (nextcbuf > CBUFSIZE) {
            printf("out of word storage\n")
            exit()
            }
        strcpy(word, 1, cbuf, tbuf[p+WORD])
        total = total + 1
        tbuf[tree] = p
        return (p)
        }
end
# tprint - print tree
tprint(tree)
    integer tree, count, i

    if (tree) {
        tprint(tbuf[tree+LLINK])
        count = tbuf[tree+COUNT]
        printf("%d\t%f\t", count, 100.0*count/total)
        for (i = tbuf[tree+WORD]; cbuf[i]; i = i + 1)
            putc(cbuf[i])
        putc('\n')
        tprint(tbuf[tree+RLINK])
        }
end
# strcmp - compare s1[i] and s2[j], return <0, 0, or >0
integer strcmp(s1, i, s2, j)
    char s1[], s2[]
    integer i, j

    while (s1[i] == s2[j]) {
        if (s1[i] == 0)
            return (0)
        i = i + 1
        j = j + 1
        }
    if (s1[i] == 0)
        return (-1)
    else if (s2[j] == 0)
        return (1)
    else
        return (s1[i] - s2[j])
end
# strcpy - copy string at s1[i] to s2[j]
strcpy(s1, i, s2, j)
    char s1[], s2[]
    integer i, j

    while (s2[j] = s1[i]) {
        i = i + 1
        j = j + 1
        }
end
# strlen - return length of s
integer strlen(s)
    char s[]
    integer i

    for (i = 1; s[i]; i = i + 1)
        ;
    return (i - 1)
end
end
```

Acknowledgments

The many helpful comments and suggestions from Chris Fraser and Jack Davidson are greatly appreciated.

References

[aho77]
Aho, A. V. and Ullman, J. D. *Principles of Compiler Design*. Addison-Wesley, Reading, MA, 1977, Sec. 5.5.

[ans77]
American National Standards Institute. *USA Standard Code for Information Interchange*. X3.4-1977, New York, 1977.

[dav80]
Davidson, J. W. and Fraser, C. W. A retargetable peephole optimizer and its application to code generation. *ACM Trans. on Prog. Languages and Systems 2*, 2 (Apr. 1980), 191-202.

[fra79]
Fraser, C. W. A compact, machine-independent peephole optimizer. *Conf. Rec. Sixth ACM Ann. Symp. on the Prin. of Prog. Languages*, San Antonio, Jan. 1979, 1-6.

[gri80]
Griswold, R. E. and Hanson, D. R. Reference Manual for the Icon Programming Language. Tech. Rep. TR 79-1a, Dept. of Computer Science, Univ. of Arizona, Tucson, Feb. 1980.

[han79a]
Hanson, D. R. Software Tools Programmer's Manual, Tech. Rep. TR 79-15, Dept. of Computer Science, Univ. of Arizona, Tucson, Aug. 1979.

[han79b]
Hanson, D. R. A simple technique for controlled communication among separately compiled modules. *Software—Practice and Experience 9*, 11 (Nov. 1979), 921-924.

[han80a]
Hanson, D. R. Is Block Structure Necessary? Tech. Rep. TR 80-3, Dept. of Computer Science, Univ. of Arizona, Tucson, Mar. 1980.

[han80b]
Hanson, D. R. Code Improvement via Lazy Evaluation. Tech. Rep. TR 80-8, Dept. of Computer Science, Univ. of Arizona, Tucson, Apr. 1980.

[ker75]
Kernighan, B. W. Ratfor—a preprocessor for a rational Fortran. *Software—Practice and Experience 5*, 4 (Dec. 1975), 396-406.

[ker76]
Kernighan B. W. and Plauger, P. J. *Software Tools*. Addison-Wesley, Reading, Mass., 1976.

[ker78]
Kernighan, B. W. and Ritchie, D. M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1978.

[rit74]
Ritchie, D. M. and Thompson, K. The UNIX timesharing system. *Comm. ACM 17*, 6 (Jul. 1974), 365-375.

[wir76]
Wirth, N. *Algorithms + Data Structures = Programs*. Prentice Hall, Englewood Cliffs, NJ, 1976.