The *W Programming Language

John Colagioia

199?

Introduction

The *W language should be based on the W language which, of course, does not exist. Instead it is based on an assortment of odds and ends which could be useful in languages, but never seem to have been implemented (and definitely shouldn't have been implemented in the same language), combined with some patching added to, firstly, make *W truly bizarre and, secondly, to make it as functionally complete a language as C and C++ (from which some concepts such as casting have been borrowed, as well as the name convention), for example. The data types should provide enough of a range that any structure may be built up, and includes arbitrary bitstrings, machine-independant pointers (useful for a pass-by-reference), name bindings of data (useful for a pass-by-name), allows for homogenous array-like compositions, and has a semi-structured composite data type. All arithmetic expressions are constructed in prefix to preserve continuity with subroutine calls, and include a fairly complete set of arithmetic and bitwise operators and inbuilt functions to execute any calculation.

The language is also quite robust in flow control, allowing for conditionals, iteration (bounded and unbounded), function calls, interrup-driven, and even random execution. To enforce structured programming, however, neither a "go to" or a "come from" statement has been implemented in *W.

To minimize readability, of course, *W is fully case insensitive, so that `Count`, `COUNT`, `count`, and `COunT` are all indistinct except under fairly confusing circumstances (which may or may not exist, depending on the implementation). Data instances must begin with an alphabetic character (`a-z` or `A-Z`), an underscore (_), or a hyphen (`-`). The remainder of the name may be made up of any alphanumeric characters and the underscore, hyphen, and, of course, the right bracket (`]`). Comments are also possible in *W (though not necessarily suggested as the language is fairly confusing without misspelled and incorrect descriptions of the program to botch things), and may be included in the text by placing a double pipe(`||`) at the beginning of the comment, terminated by the doubled end-of-statement marker (`!!`). Such comments may be placed anywhere in the program, and should be completely ignored by the compiler (just as the are by most programmers). This does not mean that comments are equivalent to whitespace. On the contrary, the compiler considers comments to simply not exist, essentially concatenating the strings on either side of the comment.

*W, like most modern languages, is entirely freeform, meaning that statements are not constrained to the dimensions of, say, a punch card, teletype, computer monitor, or three-dimensional, virtual reality programmers' editor. Program format, therefore, is entirely dependant on input device, host computer's character set, and lack of programmer style, though the compiler is permitted (actually somewhat encouraged) to mock poor format style.

*W Data Types

The *W data types are designed with maximum versatility in mind. With them, any other known (and several unknown) types may be built. In addition, several predefined instances of these types are provided to enhance the language.

`bits` A bitstring of arbitrary length.

`cplx` A complex number in the mathematical form $(\alpha + \beta j)$ where α and β are integers and j is the square foot of –1. Each component of a `cplx` is specified to have a minimum precision of –32768 ... 32767, but may have more, depending on the implementation.

`sack` A (semi)structured data type consisting of a collection of element which can be packed, unpacked, and checked with other data.

`dref` A reference to an instance of some data type.

`name` A name of another datum.

`chrs` A character string of arbitrary length.

`hole` A data type with no value. May pose as any type.

Predefined *W Instances

The following data instances are provided to the *W programming environment to facilitate programming certain concepts which would be nearly impossible otherwise.

`WORLD(bits)` The *W representation of the outside world. Assigning an expression to `WORLD` (see below) causes the character represented by the expression to be appended to the computer display. Likewise, using `WORLD` in an expresssion represents the value of the next character in the input buffer (if any).

`NOWHERE(hole)` `NOWHERE` is a place to discard things as well as a place to get nothing. Can also be used for comparison purposes.

`NL(chrs)` `NL` is a newline character.

`POCKET(sack)` and `RESULT(bits)` Data instances local to each routine. Both may be used to store any data, but `RESULT` will be available for the calling routine to read.

*W Program Parts

Each *W program is made of several parts. The functions part, which defines any user functions, the stuff part, which defines any instances of data for the program, and the text part, which contains the program instructions, themselves.

Functions

Subprograms which can be used from the Text, in the form.

```
@ name = Stuff Text
```

Stuff

Declarations of data to be used by the Text section of the program, with an optional constant initializer. The initial number allows multiple indexed instances to exist.

```
num/name IS type [const] !
num/name , num/name ... ARE ALL type [const] !
AUTOPACKED SACK name [, ... name] HAS type [, ... type] !
```

Text

A list of statements, appearing as:

```
TEXT: [statements] :ENDTEXT
```

*W Statement Types

```
statement % expr !
```

Runs statement with a probability of expr. If expr is less than 100, the statement is executed that percentage of time. If it is greater, the statement is executed expr/100 more times, each time decrementing the value of expr by 100, and, if expr ever falls before 100, is subject to the first rule. A negative value for expr works just like a positive value, except only under conditions where program execution runs backwards; otherwise it is treated as a zero.

```
statement UNLESS expr !
```

The statement is executed whenever encountered except in any cases when expr evaluates to non-zero.

```
statement WHEN expr !
```

The statement is not executed when encountered, but is instead executed after any statement where expr currently evaluates to non-zero.

```
lval < expr !
expr > lval !
```

Takes the value of expr and copies it into lval.

```
function (parameters) !
```

Calls a function with the appropriate parameters. The parameter list must correspond one-to-one with the Stuff list for that function.

The scoping rules in *W are much more simplified than they would have been in W, had it existed: A function may only access data instances declared within itself, including those implicitly defined.

```
-|- (expr) !
```

If expr is zero, jumps to the end of the current block (see below), otherwise, terminates the current expr blocks. If expr is greater than the current block nesting, it does nothing. If expr is negative, the program terminates.

```
& statement & statement & ... statement &&
```

A block mechanism for multiple statements.

*W Mathematical Operations

^ X	And the bits of X (yielding a single bit).
. X	Or the bits of X.
? X	Xor the bits of X.
* X	Butterfly the bits of C, i.e. 11001100 becomes 10100101.
- A B	If A and B are simple (bits, cplx, chrs, hole), identical types, subtracts B from A.
/ A B	If A and B are numeric (cplx), divides A by B.
# A B	If A and B are numeric (cplx), takes A to the B power.
$ A B	Mingles B with A.
~ A B	Selects B bits from A.
SIZE X	Returns the size of X, in full bytes (rounded up if X is a bitstring).

*W Sack Operations

`PACK sack data:`	Add data to sack.
`UNPACK sack data:`	Remove a data-like element from sack.
`UNPACK sack:`	Remove an element from sack.
`CHECK sack data:`	Examine sack for data.
`WEIGH sack:`	Return the weight of the sack (in bits).

Other *W Operations

`NAME name AFTER data:`	Assigns the name of data to name.
`WHOIS (data):`	Returns name suggested by data.
`REF (data):`	Returns a reference to data.
`DATA (dref):`	Returns the data referred to by ref.
`FCHRS (chrs):`	Returns the first character of the string.
`LCHRS (chrs):`	Returns the last character of the string.
`FBIT (bits):`	Returns the first (lowest) bit of the bits.
`LBIT (bits):`	Returns the last (highest) bit of the bits.

Sample *W Programs

Hello, World!

Prints “Hello, World!” to the screen, followed by a newline.

```
Functions:
|| No functions for this program !!
Stuff:
    1/Hello is chrs !
    1/Sz, 1/Total are all cplx !
Text:
|| Initialize the data !!
    Hello < "Hello, World!" !
    Size Hello > Sz !
    Total < 0 !
|| Take the string length and multiply by 100 !!
    - Size - 0 Total > Total % 10000 !
|| Print and delete a character that many times !!
    &   WORLD < FCHRS (Hello) !
    &   Hello < - Hello FCHRS (Hello) !
    &&  % Total !
|| Add a newline !!
    WORLD < nl !
:Endtext
```

Factorial

Accepts a positive integer (n) as input, then outputs the factorial of n (n!).

```
Functions:
    @ mult =
        Stuff:
            1/A, 1/B are all cplx !
        Text:
```

```
            cplx (RESULT) < 0 !
            cplx (RESULT) < - cplx (RESULT) - 0 B % 10000 !
            bits (B < RESULT !
            cplx (RESULT) < 0 !
            cplx (RESULT) < - cplx (RESULT) - 0 A % B !
        :Endtext
    @ fact =
        Stuff:
            1/n is cplx !
        Text:
            RESULT < bits (1) !
            RESULT < bits (mult (n, - n 1)) unless - 1 ? n !
        :Endtext
Stuff:
    1/Input, 1/Output are all chrs !
    1/SR is chrs "0" !
    1/Num, 1/Out, 1/Place, 1/Index, 1/Mod are all cplx 0 !
Text:
    WORLD > Input !
    Place < 1 !
    Index < cplx (mult (SIZE Input, 100)) !
    &   Num < - Num - 0 cplx (mult (Place, - LCHRS (Input) ZR)) !
    &   Input < - Input LCHRS (Input) !
    &   cplx (mult (10, Place)) > Place !
    &&  % Index !
    cplx (fact (Num)) > Out !
    &   - Out cplx (mult (/ Out 10, 10)) > Mod !
    &   + 100 / Out 10 > Out !
    &   + Output chrs (+ Mod Zr) > Output !
    &&  % Out !
    Size Output > Index !
    Index < cplx (mult (100, Index)) !
    &   WORLD < LCHRS (Hello) !
    &   Hello < - Hello LCHRS (Hello) !
    &&  % Index !
    WORLD < nl !
:Endtext
```