# Scriptify Documentation


## Phil Carmody


# 1. Introduction

This is not a script language, it's more like a 'macro' front end to PFGW SCRIPT files. A bit like the `#define BEGIN {` hacks to make C look more like Pascal. However, it's remarkably deceptive how C-like the scripts can now look!

Basically, you write a script in the pseudo-C syntax, and then feed it to scriptify.pl which produces a SCRIPT file that performs the same job. I.e. there's *nothing* that these can do that SCRIPT files can't. It's purely cosmetic.

This document is not intended to be normative. In places it may even be wrong. The definition of the behaviour of the script is the script itself.

This document applies to version 0.95 of Scriptify. 0.95 is expected to become the 1.0 release with no substantive changes.

# 2. Running Scriptify

Scriptify is a perl script, conventionally called scriptify.pl. To convert pseudo-C script test.pcs to SCRIPT file test.scr, from the command line do:

```
$ scriptify.pl < test.pcs > test.scr
```

Warnings and errors will be reported to stderr.

Note - as distributed, the script may have a version number in its name, such as `scriptify.0.8.pl`. However, the name of the script actually doesn't matter.

# 3. Syntax

As Scriptify works on a strictly line-by-line basis, the number of transformations it can perform is strictly limited. However, it contains all of the functionality of the original SCRIPT format and a little syntactic sugar to make it more convenient for C programmers. The language is mostly white-space insensitive, except for the fact that each command must be on a separate line.

The following conventions are assumed in the explanations of the syntax:

- The '<' and '>' characters surround the description of the syntactic element.

- Variable, function, and label names are alphanumeric, but may not begin with a digit. '_' is considered alphanumeric. Names prefixed "PC" are reserved for the Scriptify script. All names are *case insensitive*, but Scriptify will issue a warning if a variable is used with two names that don't match exactly.
- String literals are between double-quotes, and may contain literal quote characters using the escaped sequence '\"' and literal backslashes using the escaped sequence '\\'.
- Filenames do not contain semicolons, commas, spaces, quotes, or any other character commonly used as a parameter separator.
- Arithmetic expressions are expressions containing literal numbers and integer variables in well-formed expressions containing +, -, *, /, ^, round brackets ( and ), and the functions listed in the Arithmetic Operations section.

## 3.1. Metastatements

In order to aid comprehension and code reuse, Scriptify supports two types of comments and external file inclusion. For forward compatibility, it also provides facilities for injecting arbitary SCRIPT statements into the output.

### 3.1.1. Comments

Scriptify recognises both C `/* block comments */` and C++ `// end-of-line comments`.

C-style block comments are discarded entirely by the script, and therefore can be used for in-depth annotations, or alternatively for removing large chunks of unwanted code that one might still want to refer to. For example:

```
/* <anything> */
```

e.g.

```
/* The following variable is used to store the
   information on the state of the number
   whose state we wish to be aware of in both
   an epistemic and numeric sense */
int x;

/* Phil - if we need x's value later, reenable this
int old_x=x;
*/

/* this */ string /* is */ not /* readable */;
```

A C++-style end-of-line comment is propagated into the output SCRIPT file as a comment prefixed with a ':'. The comment will be injected into the SCRIPT file before any other SCRIPT statements created by the line.

```
// <anything>
```

e.g.

```
a+=2;  // increment a by two
```

Note - comment processing takes place before any other parsing of the file. This means that it's impossible to insert things that look like comments into literal strings. To overcome this, use escape characters so that the comment detecter does not recognise the construct as a comment:

```
string s1="/\* not a comment *\/";
string s2="/\/ not a comment ";
```

### 3.1.2. File Inclusion

Scriptify permits scripts to include external pseudo-C script files:

```
# include <filename>

where:
  <filename> is a literal string in double quotes.
```

Inclusion may be recursive.

### 3.1.3. SCRIPT Statement Injection

For forward comparability with future versions of the SCRIPT syntax, Scriptify supports arbitary SCRIPT statement injection:

```
# pragma __emit <anything>
```

e.g.

```
#pragma __emit DDOS 131.72.156.54,SYNFLOOD
```

Absolutely no processing or checking of the line is performed.

### 3.1.4. Processing Commentary

The scripts are permitted to commentate on their processing, for example if a line in the script is known to flag a usage warning, but the code is known to be correct, then the commentary can be used to reassure the user that the warning is expected.

```
# pragma warn <anything>
```

e.g.

```
#pragma warn Don't worry - file will have been opened in a helper function.
x=fread(myfile);
```

The warning is issued to the standard error stream, stderr, in the same way as compilation errors and warnings, but is marked `#Warning`, to distinguish it from Scriptify-generated warnings.

## 3.2. Variables

All variables must be declared lexically before they are first used.

There are three types of variable: integers, strings, and file handles. It is not permitted to declare a variable more than once. Variables may not have the same name as either labels or functions, either built-in or user-defined.

### 3.2.1. Declarations

The syntax for a variable declaration is as follows:

```
<type> <name> ;
```

e.g.

```
integer bignum;
string  description;
file    logfile;
```

### 3.2.2. Declaration with Assignment

Variables may be assigned to as part of the declaration:

```
<type> <name> = <value> ;
```

e.g.

```
integer bignum=k*2^n+1;
string  description="big \"Proth\" prime";
file    logfile=fopen("prothlog.out", "w");
```

The range of expressions for *<value>* is limited. The following is an exhaustive list of expression schemata that are permitted:

```
integer <name> = <arithmetic-expression> ;
string  <name> = <string-literal> ;
file    <name> = fopen ( <filename> , <mode> ) ;

where:
  <filename> is a literal string in double quotes or a string variable.
  <mode> is one of "r", "w", "a". See Section 3.3.8 below.
```

### 3.2.3. Assignment

The syntax for assignment is:

```
<name> = <value> ;

where:
  <name> is a previously declared variable name
  <value> is an expression compatible with the variable's type
```

e.g.

```
bignum=bignum-2;
description="Riesel equivalent";
infile=fopen("prothcandidates", "r");
myresult=myfunction(param1,param2);
```

The list of acceptable expressions is a superset of those for the simultaneous declaration and assign. The additional expressions, documented in Section 3.3 and Section 3.4, are as follows:

```
<integer-variable> =
  atoi( <string-variable> ) ;
  atoi( <string-literal> ) ;
  powmod ( <base> , <power> , <modulus> ) ;
  factorize ( <value> ) ;
  factorize ( <value> , <fmin> ) ;
  factorize ( <value> , <fmin> , <fmax> ) ;
  PRP ( <value> ) ;
  PRP ( <value> , <string-literal> ) ;
  PRP ( <value> , <string-variable> ) ;
  fread ( <file-handle> ) ;
  fread ( <file-handle> , & <string-variable> ) ;
  system ( <string-variable> ) ;

  <user-function> ( <parameters> ) ;
```

### 3.2.4. Modification Operators

Integer variables can be modified using the following syntax:

```
<name> <op>= <value>

where:
  <op> is +, -, *, /, %, <<, or >>.
  Note - there is no space between the operator and the '='.
```

This is exactly equivalent to:

```
<name> = <name> <op> ( <value> )
```

except in the case of "<<" and ">>", which are converted to "*2^" and "/2^" respectively.

### 3.2.5. String Construction

As well as a simple assignment from a literal string, it's possible to construct a string expression from a format string and other variables.

```
sprintf ( <string-variable> , <format-string> , <variable-list> ) ;

where:
  <format-string> is a literal string containing
    no unescaped semicolons or quote characters, and with
    %d used to interpolate an integer variable,
    %s used to interpolate a string variable, and
    %% to represent a literal quote character.
  <variable-list> is a comma-separated list of integer and string
    variables.
```

e.g.

```
sprintf(str, "\"Proth\" prime = %d*2^%d+1, %s", k, n, status);
```

### 3.2.6. Pseudovariables

There are five integer pseudovariables which are implicitly used to store some built-in functions return types. They must not be declared explicitly. These variables are:

#### Pseudovariables

ERRORLEVEL

: stores the result of the most recent system call.

FACTORFOUND

: stores the result of the most recent factorize call.

ISPRIME

: stores the result of the most recent PRP call.

MAXF, MINF

: store the factorisation limits used for the most recent factorize call.

## 3.3. Built-in Functions

The pseudo-C script language understands the following functions:

- atoi: for evaluating a string.
- powmod: for performing a modular exponentiation.
- factorize, PRP: for trial dividing a number, and performing a PRP test.
- puts, print, printf: for writing to the PFGW log file and the screen.
- fopen, fclose: for opening and closing files.
- fread, write, fprintf: for reading/writing to (open) files.
- system: for executing an external command/program.

### 3.3.1. String Evaluation

It is possible to get PFGW to evaluate a string, and return the value as an integer using the atoi function as follows:

```
<integer-variable> = atoi ( <string-variable> ) ;
<integer-variable> = atoi ( <literal-string> ) ;
```

e.g.

```
int k=17;
int n=12345;
int p;
string expr="k*2^n+1";
p=atoi(expr);
```

### 3.3.2. powmod

powmod is a helper function that quickly evaluates a modular exponentiation:

```
<integer-variable> = powmod ( <arithmetic-expression> , <arithmetic-expression> , <arithmetic-exp
```

e.g.

```
b4096 = powmod(base, 4096, k);
```

### 3.3.3. factorize

This function's purpose is to test if a number has small factors, using trial division. Optionally, the bounds for the trial division can be set by calling the function with two parameters to set the lower bound, or three parameters to set both bounds:

```
factorize ( <integer-variable> ) ;
factorize ( <integer-variable> , <arithmetic-expression> ) ;
factorize ( <integer-variable> , <arithmetic-expression> , <arithmetic-expression> ) ;
```

e.g.

```
factorize(b4096, 4097);
```

If upper or lower bounds are not specified, then whatever was used for the previous factorisation is used again.

The FACTORFOUND pseudovariable is always assigned the return value of this function, which is the small factor found, or zero if no small factor is found. However, the return value may optionally be assigned to an ordinary integer variable as well.

This function also sets the ISPRIME pseudovariable to zero if a factor is found.

### 3.3.4. PRP

The PRP function can be called two ways:

```
PRP ( <arithmetic-expression> ) ;
PRP ( <arithmetic-expression> , <string-expression> ) ;
```

The function always sets the ISPRIME psuedovariable, but may optionally have the return value assigned to an ordinary integer variable.

The return value is 0 if the expression evaluates to a composite number, and 1 if it evaluates to a (pseudo-)prime.

Note - PFGW honours the -f (and -e, -s, etc.) flags while in script mode; therefore the PRP function will also perform trial division first if so requested.

### 3.3.5. print

The print function takes a single integer variable and writes it to both the screen and the PFGW log file, appending a new line.

```
print ( <integer-variable> ) ;
```

e.g.

```
print(b4096);
```

There is no return value from the print function.

### 3.3.6. puts

The puts function takes a single string variable and writes it to both the screen and the PFGW log file, appending a new line.

```
puts ( <string-variable> ) ;
```

e.g.

```
puts(expr);
```

There is no return value from the puts function.

### 3.3.7. printf

The printf function writes a formatted string to both the screen and the PFGW log file, appending a new line.

```
printf ( <format-string> , <variable-list> ) ;
```

The parameters are the same as the corresponding ones in the sprintf functions. E.g.

```
printf("%s => %d", expr, b4096);
```

There is no return value from the print function.

### 3.3.8. fopen

The fopen function takes a filename, and a file mode, and returns a file handle which can be used to read from, or write to, the file. See Section 3.3.9 and Section 3.3.10.

```
<file-handle> = fopen ( <filename> , <mode> ) ;

where:
  <filename> is a literal string in double quotes or a string variable.
  <mode> is "r" for files opened for reading only
            "w" for files opened for over-writing
    "a" for files opened for appending
```

The return value must be assigned to a file variable.

### 3.3.9. fread

The fread function takes a file handle for a file that was opened for reading (mode "r"), and an optional second parameter which is a reference to a string that is to be set to the literal string read from the file. The function reads a single line from the file, and treats it as a single arithmetic expression.

```
<integer-variable> = fread ( <file-handle> ) ;
<integer-variable> = fread ( <file-handle> , & <string-variable> ) ;
```

The return value is the integer value of the arithmetic expression read.

### 3.3.10. write

The write function takes a file handle for a file which was opened for either writing or appending (modes "w" and "a"). It also takes a single integer or string variable which is to be written to the file. A newline is also written to the file after the variable.

```
write ( <file-handle> , <integer-variable> | <string-variable> ) ;
```

There is no return value from the write function.

### 3.3.11. fprintf

The fprintf function writes a formatted string to the specified file and appends a new line:

```
fprintf ( <file-handle> , <format-string> , <variable-list> ) ;
```

e.g.

```
fprintf(blog, "%s => %d", expr, b4096);
```

The parameters are the same as the corresponding ones in the sprintf and printf function.

There is no return value from the fprint function.

### 3.3.12. system

The system function is used to execute an arbitrary string using the underlying command interpreter. By definition this means that it is OS and shell dependent.

```
system ( <string-variable> ) ;
```

e.g.

```
sprintf(gpstr, "echo 'polyrootsmod(polcyclo(%d)-1,%d)' | gp -q > gplog", nn, pp);
system(gpstr);
```

The return value of the command is stored in the pseudovariable ERRORLEVEL, and may also optionally be assiged to an integer variable.

## 3.4. User-defined Functions

Scriptify supports rudimentary user-defined functions. Functions may have their parameter types and return declared in advance of actual definition:

```
<type> <name> ( <parameter-list> ) ;

where:
  <parameter-list> is a comma-separated list of variable declarations.
```

To define the function's implementation, use the following syntax:

```
<type> <name> ( <parameter-list> ) {
   <other-statements>
}
```

The return type, parameter types, and *parameter names* must be identical in the function's definition and in any forward declarations.

Note that the function name must be unique, and not used elsewhere as a label or variable name. It is advised that parameter names not be the same as any global variables. However, local variables are scoped, so they may be reused in different functions. Effectively, all variables are global variables. There's little shame in using global variables instead of parameters, for reasons of efficiency.

Any number of the statements between the bounding curly brackets may be return statements. These permit a value to be passed back to the caller of the function.

## 3.5. Flow Control

There are 3 simple flow control techniques. Firstly, the pseudo-C script understands labels and goto statements for unconditional flow control. Secondly, it understands simple conditional execution of statements using "if" statements. Thirdly, it understands simple "while", "do", and "for" loops. These statements are the most white-space sensitive of the statements in the language; it is vital, for example, to ensure that 'K&R' brace style is used. Sorry.

In addition to these, a script can be terminated by calling exit at any point.

### 3.5.1. Labels and goto Statements

A label is defined by the following syntax:

```
<label> :
```

All labels must be unique, and must also be distinct from any variable names. Labels beginning 'PC' are reserved for use by the script translater.

Flow is transferred to a point specified in the script by a label by using a goto statement with the following syntax:

```
goto <label> ;
```

### 3.5.2. if Statements

Arithmetic expressions can be used as a boolean predicate to alter the program flow. A value of 0 is taken as false, and any other value is taken as true. The syntax of a simple if statement is:

```
if ( <arithmetic-expression> ) {
   <other-statements>
}
```

Note - the K&R-style opening brace is mandatory. The other statements are only processed if the predicate is true.

An else clause may also be used:

```
if ( <arithmetic-expression> ) {
   <other-statements-1>
} else {
   <other-statements-2>
}
```

In this instance, the statements `<other-statements-1>` will be processed if the predicate is true, and the statements `<other-statements-2>` will be processed only if the predicate is false.

### 3.5.3. do, while, and for Loops

Rudimentary loops can be set up using one of three constructs, each of which has the same general semantics as the C equivalent, viz. "do" loops, "while" loops, and "for" loops.

Explicitly, "do" loops:

```
do {
  <other-statements>
} while ( <arithmetic-expressions> ) ;
```

"while" loops:

```
while ( <arithmetic-expression> ) {
  <other-statements>
}
```

or "for" loops:

```
for ( <assignment-expression> ; <arithmetic-expression> ; <assignment-expression> ) {
  <other-statements>
}
```

The arithmetic expression is treated as a predicate in the same way that it is for if statements. The constructs have the same semantics as they do in C. Note, however, that the K&R-style brace is mandatory.

For "for" loops, the assignment expressions are single assignments to integer variables with the same syntax as an assignment or modify expression as documented in Section 3.2.3 and Section 3.2.4. As in C, the first one is executed once before the loop begins, and the latter one is executed at the end of the loop, just before the condition predicate is evaluated.

Loops may be broken out of early using the break statement. The break statements' syntax, unlike that of C, may contain a parameter which indicates how many levels of nested loops out from which it should break. Its syntax is simply:

```
break ;
break ( <constant> ) ;

where:
  <constant> is a small decimal integer.
```

Typical use of break might be as in the following:

```
for(mainloop=100; mainloop<200; mainloop+=4) {
  while(condition1) {
    do {
      // statements elided ...
      if(condition2) {
```

```
        break(2);
      }
    } while(condition3);
  }
  // the break would jump here
}
```

This particular example shows the break statement breaking out of two loops, the do and the while, when condition2 is satisfied. The next statements to be executed would be the `mainloop+=4` from the for loop, and then the condition `mainloop<200` would be checked.

### 3.5.4. Terminating a Script

A script terminates with the exit function, whose syntax is:

```
exit ( ) ;
```

I.e. it is a funtion which takes no parameters and returns no value.

### 3.5.5. return Statements

A return statement not within a user-defined function acts identically to an exit() statement. Otherwise, it causes the optional returned value to be returned to the caller of the function:

```
return ;
return <expression> ;
```

The optional expression must be compatible with the return type of the function. For integers, it must be an integer variable or an arithmetic expression, and for strings it must be a string variable or string literal.

## 3.6. Arithmetic Operations

Arithmetic expressions are compatible with PFGW's expression syntax, and, thus, with SCRIPT files' expression syntax. No attempt is made to parse the expressions; they are passed to PFGW uninterpreted. Therefore, this documentation may be incomplete. Please see other PFGW documentation for more information on expression syntax.

I.e. as well as the usual arithmetic operations, +, -, *, /, %, <, <=, >, >=, but *not* << or >>, the pseudo-C script language accepts the following functions:

```
C, F, GCD, IF, L, LEN, LINEAR, LUCASV, LUCASU, P, PHI, PRIMU,
PRIMV, R, S, SM, SMR, U, V, W
```

These functions are documented elsewhere in PFGW's documentation.

## 3.7. Alii

The management reserve the right to swap the usage of these alii without notice. Sorry, but we're in sub v1.0 territory, and we can't quite work out what's the best name to use yet. Feel free to let the management know which of the alii you prefer.

int can be used as an alias for integer, the variable type.

eval can be used as an alias for the function atoi. However, don't expect to be able to mail your scripts to other people just in case anti-malware scripts rewrite them ( http://groups.google.com/groups?selm=CMM.0.90.4.987039292.risko%40chiron.csl.sri.com ).

factorise can be used as an alias for function factorize.

read can be used as an alias for function fread.

# 4. Transformations

This list is most useful for either finding how the pseudo-C script commands are implemented in terms of SCRIPT commands, or finding where in this manual one can find a description of the pseudo-C alternatives for a given SCRIPT command.

Variable declaration (Section 3.2.1, Section 3.2.2)

```
integer                   <-> DIM
string                    <-> DIMS
file (just declaration)   <-> nothing
file (with assignment)    <-> OPENFILE(IN|OUT|APP)
```

Variable assignment (Section 3.2.3, Section 3.2.4, Section 3.2.5)

```
integer =, <op>=          <-> SET
string =                  <-> SETS
sprintf                   <-> SETS
file = fopen()            <-> OPENFILE(IN|OUT|APP)
```

Built-in functions (Section 3.3)

```
atoi                      <-> STRTOINT
powmod                    <-> POWMOD
factorize                 <-> FACTORIZE
PRP                       <-> PRP
print, puts               <-> PRINT
```

```
fopen                    <-> OPENFILE(IN|OUT|APP)
fclose                   <-> CLOSEFILE
fread                    <-> GETNEXT
write                    <-> WRITE
fprintf                  <-> SETS and WRITE
system                   <-> SHELL
user-defined functions   <-> SET, SETS, and flow control
```

Flow control (Section 3.5)

```
labels                   <-> LABEL
goto                     <-> GOTO
if, else                 <-> IF, ELSE
do, while, for, break    <-> a synthesis of the above 4
exit, return             <-> END
```

# 5. Errors and Warnings

Note - Just because Scriptify translates a pseudo-C script to a SCRIPT script does not mean that either the pseudo-C or the SCRIPT script is well-formed. Syntax checking is quite rudimentary.

Errors and warnings are reported with the following schema:

```
<line-number> <type> <explanation>

where:
  <type> is "Error", "Warning", or "#Warning".
```

If an error is detected within a #included file, the line number reported is that of the #include line.

Errors are fatal; processing cannot continue. Warnings, however, whilst they may indicate real errors in the script, are not fatal because the translater cannot know all the run-time properties of the script.

Warnings that appear with a type of "#Warning" are not errors detected by Scriptify, but are ones output at the request of the script itself as documented in Section 3.1.4.

## 5.1. Errors

### 5.1.1. Processing Errors

```
Can't include file <filename>
```

### 5.1.2. Name or Type Errors

All variables must be declared lexically before, i.e. earlier in the script file than, any usage of that variable. Some illegal usage is detected, and will result in the following error messages:

```
Can't <operation> undeclared <type> <variable>.
Can't declare <variable> as <type>, already a <variable> declared at <line-number>.
Trying to <operation> <type> <variable> from line <line-number>, should be type <type>.
Function <name>'s return type (<type>) doesn't match (<type>) in forward declaration on line <l
Function <name>'s protoype (<parameter-list>) doesn't match (<parameter-list>) in forward decla
```

### 5.1.3. Structural Errors

The braces for if's, do's, and while's must be in sensibly nested pairs. If they aren't, then the following error messages might be seen:

```
} with no context.
} else { with no context.
} while with no context.
Can't break requested number of loops.
```

User-defined functions may not be nested, and attempts to do so will yield an error:

```
Cannot nest function <name> within function <name>.
```

If a function has a forward declaration, but no definition, then the following error will be reported: function "<*name*>" not found.

A goto may precede the label to which it jumps. If such labels were missing, then the follow error will be generated:

```
goto target "<name>" not found.
```

If the number of closed braces is in deficit, then the following will be seen, which gives some, admittedly cryptic, indication of the constructs which have not been terminated correctly:

```
Context=<list>.
```

The follwing errors will hopefully never be seen:

```
Breakage=<number>.
```

### 5.1.4. Other Errors

Sometimes the translater just has to give up, as it doesn't know what is going on. If that is the case, you'll see the following:

```
Can't parse line.
```

## 5.2. Warnings

These warnings are indicative that almost certainly something is wrong with the script, and should be investigated.

Some warnings are simply complaining about the use of non-identical names for the same variable, function, or label, as, to PFGW, names are case-insensitive, but it's considered naughty to be inconsistent.

```
Treating <type> <name> as the same as the <type> <name> from line <line-number>.
```

Similarly, the use of a function's local variable with the same name as a global variable flags a warning:

```
Parameter <type> <name> shadows global <type> <name>.
```

Finally in this class is the situation where repeated forward declarations are found. This isn't really a problem, but having redundant statements in scripts is, quite frankly, redundant.

```
Forward declaration of <function-name>, while valid, duplicates that from line <line-number>.
```

Another class of warning pertains to the potential use of (correctly declared) files when the script has not detected the correct sequence of operations (open before read/write and close):

```
possible use of already open file <file-handle>.
possible read of unopened file <file-handle>.
possible read of write/append file <file-handle>.
possible write to unopened file <file-handle>.
possible write/append to file opened for reading <file-handle>.
possible close of unopened file <file-handle>.
```

However, the following indicates that an almost certainly illegal ';' character was found in a format string for sprintf, printf, or fprintf:

```
literal ';' found in format string. Uglifying.
```

The 'uglification' that takes place is the propagation of a '£' (Sterling) symbol in place of the errant semicolon(s).

Some minimal checking is performed on the arithmetic expressions used in calls to functions that accept such expressions. In particular, calls to user-defined functions will *not* be understood by the PFGW parser, and therefore these is almost certainly a problem with the script:

```
Possible use of user-function <function-name> in arithmetic expresssion.
```

The work-around for this limitation is simply to assign the return value of the user-defined function to a variable, and then use that variable in the expression instead.

Finally, the following warning may be seen. If it is, the chances are that an actual error will follow anyway, but if the error received is "`Can't parse line.`" this warning might indicate the simplest remedy:

```
Almost certainly an error. Missing :, {, }, or ;
```

At the present time, *all* valid lines end with one of those four symbols.

# 6. Issues

There are no known 'issues', as in bugs, in the Scriptify script itself. However, there are some issues with the underlying SCRIPT syntax. For example, it appears that labels cannot have digits in them, else they get interpreted differently (as line offsets).

# 7. Future Enhancements

Simply brainstorming...

- string concatenation

  *<name> .= <string-literal>*

  e.g.

```
string str;
integer val;
val = read(inp, &str);
PRP(val+2);
if(ISPRIME) {
  str .= " and twin"
}
PRP(val*2-1);
if(ISPRIME) {
  str .= " and CC"
}
```

  Maybe use C++'s "+=" rather than Perl's ".=" ?

  Difficulty - easy

- Brace style flexibility

  Purely cosmetic. It requires adding more state to the parser, which means more complexity.

  Difficulty - unknown, let’s call it medium

- Persuade Noah Webster to ensure that agent nouns end "-er".
- Thank Jim, Anna, houseofshred.com, and Saku Olletehas