The

Cowgol v2.0

programming language

v1.3

compiled by Ladislau Szilagyi, 2024-2025

Introduction

Cowgol is a programming language for very small systems (6502, Z80, etc).

The current version of the Cowgol language is v2.0.

Cowgol language was invented by David Given in the late 2010’s; he wrote also a Cowgol compiler (see https://github.com/davidgiven/cowgol).

The basic set of Cowgol features is:

- strongly typed --- no implicit casting (not even between integers of different widths of signedness)
- records, pointers etc
- subroutines with multiple input and output arguments
- arbitrarily nested subroutines, with access to variables defined in an outer subroutine
- no recursion and limited stack use
- byte, word and quad arithmetic for efficient implementation on small systems
- simple type inference of variables if they're assigned during a declaration
- separate compilation with global analysis

For Z80 computers, there is a complete Cowgol development environment, hosted on CP/M (see https://github.com/Laci1953/Cowgol_on_CP_M), enabling the user to mix Cowgol, C and assembler source files to build an executable.

A short introduction will be followed by a detailed description of the syntax and semantics of the Cowgol language.

Chapter 1 - A Cowgol language tutorial

We begin with a quick introduction in Cowgol, to show the essential elements of the language in real programs, but without getting bogged down in details, rules, and exceptions.

The first program to write is the same for all languages: Print the words “hello, world”.

In Cowgol, the source code to do this is:

```
include "cowgol.coh";
print("hello, world");
```

After compiling-it (without showing here all the compilation details…)

```
A>cowgol hello.cow
```

, we can run-it:

```
A>hello
hello, world
```

Now, some explanations about the program itself.

A Cowgol program, consists of *statements*, *subroutines* and *variables*.

A subroutine (somehow equivalent to the C language *function*) contains statements that specify the computing operations to be done, and variables store values used during the computation.

In our example, we have only statements (not every program must have a subroutine).

The first line:

```
include "cowgol.coh";
```

tells the compiler to include another file, named cowgol.coh; this file contains some useful, basic subroutine definitions (e.g. print).

1.1 Variables and Arithmetic Expressions

The second program uses the formula $^{o}C=(5/9)(^{o}F-32)$ to print the table of Fahrenheit temperatures and their centigrade or Celsius equivalents.

This program introduces several new ideas, including *comments*, *declarations*, *variables*, *arithmetic expressions*, *loops* , and *subroutines*:

```
include "cowgol.coh";

record buffer is
	bytes: uint8[5];
```

```
end record;
var buf: buffer;

sub itoa(i: int16): (pbuf: [uint8]) is
	var sign: uint8;

	pbuf := &buf.bytes[4];  # points to terminating zero
	[pbuf] := 0;

	if (i >= 0) then
		sign := 0;
	else
		i := -i; sign := 1;
	end if;

	loop
		pbuf := pbuf - 1;
		[pbuf] := '0' + ((i % 10) as uint8);
		i := i / 10;
		if i == 0 then break; end if;
	end loop;

	if (sign == 1) then
		pbuf := pbuf - 1; [pbuf] := '-';
	end if;
end sub;

sub convertF() is
	const TAB := 9;
	var fahr: int16;
	var celsius: int16;
	var lower: int16;
	var upper: int16;
	var step: int16;

	lower := 0;
	upper := 300;
	step := 20;
	fahr := lower;

	while fahr <= upper loop
		celsius := 5 * (fahr - 32) / 9;
		print(itoa(fahr)); print_char(TAB); print(itoa(celsius)); print_nl();
		fahr := fahr + step;
	end loop;
end sub;

convertF();
```

```
A>cowgol convert.cow
A>convert
```

```
0      -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

The fragment “# points to terminating zero” is a *comment*; any characters following # in the current line are ignored by the compiler.

In Cowgol, all *variables* must be declared before they are used, usually at the beginning of the subroutine before any executable statements.

A *declaration* announces the properties of variables:

```
var fahr: int32;
```

declares a signed 32 bit variable named “fahr”.

1.2 Subroutines

There are two *subroutines* in this source file: “itoa” and “convertF”.

The first one, “itoa”, converts a signed 16 bit integer to its decimal ASCII representation as a string. The second one, “convertF”, does the temperature conversion computations.

The method of communicating data between subroutines is for the caller to provide a list of values, called arguments, to the subroutine it calls. The parentheses after the function name surround the argument list.

Computation in the temperature conversion begins with the *assignment statements*:

```
lower := 0;
upper := 300;
step := 20;
```

which set the variables to their initial values.

Individual statements are terminated by semicolons. It is possible to include more than one statement in a single program source line:

```
print(itoa(fahr)); print_char(TAB); print(itoa(celsius)); print_nl();
```

The print(itoa(fahr)); statement means that first the subroutine itoa is called, then the returned pointer will be used to print the integer ASCII representation of the value contained in the fahr variable.

Each line of the table is computed the same way, so we use a loop that repeats once per output line; this is the purpose of the while loop:

```
while fahr <= upper loop
        …
end loop;
```

The *while loop* operates as follows: the condition following “while” is tested. If it is true (fahr is less than or equal to upper), the body of the loop (the statements until “end loop”) is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false (fahr exceeds upper) the loop ends, and execution continues at the statement that follows the loop.

Most of the work gets done in the body of the loop. The Celsius temperature is computed and assigned to the variable “celsius” by the statement :

```
celsius := 5 * (fahr-32) / 9;
```

The reason for multiplying by 5 and dividing by 9 instead of just multiplying by 5/9 is that in Cowgol, as in many other languages, integer division truncates: any fractional part is discarded. Since 5 and 9 are integers. 5/9 would be truncated to zero and so all the Celsius temperatures would be reported as zero.

Chapter 2 – Types, Operators and expressions

2.1 Variable names

There are some restrictions on the names of variables and symbolic constants. Names are made up of letters and digits; the first character must be a letter. The underscore ``_'' counts as a letter; it is sometimes useful for improving the readability of long variable names. Upper and lower case letters are distinct.

2.2 Data types

Cowgol provides the following set of scalar data types:

uint8	unsigned 8 bits integer
int8	signed 8 bits integer
uint16	unsigned 16 bits integer
int16	signed 16 bits integer
intptr	alias to uint16
uint32	unsigned 32 bits integer
int32	signed 32 bits integers

No floating point data type is provided.

No implicit casting is done. If you want to use mixed types, you must explicitly convert, using the *as* keyword.

Example:

```
var i: uint8;

var j: uint16;

j := j + i;            # wrong, the compiler reports a syntax error here!

j := j + (i as uint16); # right!

j := j + 1;             # as a special exception, numeric constants work anywhere
```

It is possible to define aliases for these data types, using *typedef* :

```
typedef byte is uint8;
```

Also, typedef can be used to define new data types having a specific value range:

```
typedef nibble is int(0,15); # 0 <= nibble <= 15
```

2.3 Constants

In Cowgol, you can also define constants:

```
const ZERO := 0;
```

The values used in Cowgol can be hexadecimal (0x12AB), decimal=default (123 or 0d123), octal (0o17), binary (0b101) ; _ characters are ignored in numbers (12_345 is a valid number).

A character constant is an integer, written as one character within single quotes, such as 'x'.

Certain characters can be represented in character and string constants by escape sequences : \n (newline), \r (carriage return), \t (tab), \e (ESC), \\ (\), \’ (‘), \” (“), \0 (0 value); these sequences look like two characters, but represent only one.

A string constant, or string literal, is a sequence of zero or more characters (up to 128) surrounded by double quotes, as in:

```
"I am a string"
or
"" # the empty string
```

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; \" represents the double-quote character.

2.4 Declarations

Variables are declared using the var keyword:

```
var n: uint8;
```

Variables are not initialised to anything, so if you don't zero arrays and structures yourself before use, they'll be full of garbage.

When declaring a variable, you can initialize its value:

```
var i: uint8 := 4;    # variable declaration with initialiser
```

You can omit the type of a variable if, at declaration, it is initialized with a value contained in another variable:

```
var x: int16 := 1;

var y := x;                 #y is declared as int16
```

2.5 Arithmetic operators

For computations, the following operators may be used: + - * / % & | ^ ~

In addition, there are the << and >> operators.
These are special; the second argument must always be a uint8.

Therefore, this is correct:

```
var a: uint8;
var b: uint16;
b := b >> a;
```

But the following one is not allowed (the compiler will issue an error message):

```
var a: uint16;
var b: uint16;
b := b >> a;
```

2.6 Relational and logical operations

The relational operators are
`>  >=  <  <=`
They all have the same precedence. Just below them in precedence are the equality operators:
`==  !=`
Relational operators have lower precedence than arithmetic operators, so an expression like i < lim-1 is taken as i < (lim-1), as would be expected.

More interesting are the logical operators *not*, *and* and *or.* Expressions connected by not and ,or or are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.

2.7 Arrays

Cowgol supports single-dimensional arrays.

```
var array: uint8[42];
array[1] := 9;              #second element (indexes start at 0)
```

The index of an array is either a uint8 or a uint16 based on the size of the array. Using the wrong type will cause a compilation error.

```
var array: uint8[42];
var i: uint16 := 9;
print_i8(array[i]);          # wrong!
print_i8(array[i as uint8]); # right!
```

It is possible to automatically determine the type of an array index using special syntax.

```
var array: uint8[42];
var i: @indexof array := 9;  # automatically picks a uint8 or a uint16
print_i8(array[i]);       # always works
```

That way you can resize your array later without having to rewrite lots of code.

Arrays also support the @*sizeof* modifier to return the number of elements in the array:

```
var array: uint8[42];
var i: @indexof array := 0;
while i != @sizeof array loop
```

```
  array[i] := 9;
  i := i + 1;
end loop;
```

2.8 Records

Cowgol supports structured records.

```
record ComplexNumber is
  i: int32;
  r: int32;
end record;

var c: ComplexNumber;
c.i := 4;
c.r := 9;
```

Records may inherit from other records:

```
record EvenMoreComplexNumber: ComplexNumber is
  q: int32;
end record;

var c: EvenMoreComplexNumber;
c.i := 4;
c.r := 9;
c.q := -7;
```

An inherited record gets all the parameters of its base class, in the same place; so it's legal to cast a pointer to one to a pointer to another and have those fields still be accessible. Implicit downcasts are not done.

You may use @*at()* to specify the actual offset of a member. This is useful for interoperation with hardware, and also to create unions.

```
record HardwareRegister is
  datareg @at(0): uint8;
  statusreg @at(1): uint8;
end record;
```

```
record UnionRecord is
  option1 @at(0): OptionOne;
  option2 @at(0): OptionTwo;
  option3 @at(0): OptionThree;
  non_union_member: uint8;  # goes after the three option members
end record
```

2.9 Static initialisers

You may define array and record variables statically. These work the way you would expect and allow arrays inside records, arrays of records, and strings:

```
var array1: uint8[3] := {1, 2, 3};    # the number of elements must match the size
var array2: uint8[] := {4, 3, 2, 1};  # or the compiler can figure it out


record Record is
        a: uint8;
        b: uint8;
        c: uint8;
        d: uint8[3];
        a_string: [uint8];
end record;
var a_record: Record := { 1, 2, 3, { 4, 5, 6 }, "foo" };
```

Variables declared in this way should be considered static --- they're initialised once on program startup and then never again. So, if you define one inside a subroutine then any changes will persist across multiple calls to the subroutine.

There's limited support for array initialisers. These only work for one-dimensional arrays of scalars (so far). They work by embedding the data in the executable, so they generate no code; but the data is intrinsically static (if you use one inside a subroutine, be careful).

```
var array: uint8[42] = {1, 2, 3, 4};  # remaining items initialised to zero
var hugearray: uint32[1024] = {};     # your executable just went up in size by 4kB
```

2.10 Pointer types

Cowgol has pointers.

```
record Structure is
        i: uint8;
end record;
```

```
var v: value := { 1 }; # member of record
var i: uint8 := 1;     # scalar type
var p: [uint8];        # pointer type
p := &v.i;             # allowed: taking the address of a member
p := &i;               # disallowed: taking the address of a scalar variable
[p] := [p] + 1;        # dereference pointer
```

The keyword ‘nil’ has the same meaning as ‘NULL’ in the C language.
Example of using ‘nil’:

```
        if p == nil then
                Exit();
        end if;
```

As in the C language, p := nil is equivalent with p := 0 as [uint8].
Another advantage of ‘nil’ : it is convertible to any type of pointer, therefore if we have another pointer to a 16bit value declared as var p16: [int16]; , we can use the same code:

```
        if p16 == nil then
                Exit();
        end if;
```

, without being constrained to specify ‘as [int16]’ as in all other cases (remember, Cowgol does not allow automated casting, therefore p := p16 as [uint8] is mandatory!).

You may not take the address of a scalar variable: that is, a simple variable containing an integer or pointer.
You can take the address of a record, or a scalar member of a record.
If you know what you're doing you can bypass this restriction with p := @alias &i.
This doesn't make it safe, it just stops the compiler generating an error.
Pointers are not indexable! But you can do pointer arithmetic on them.

Pointer arithmetic always works in bytes.

```
var p: [uint32] := &v.alignedint; # p is correctly aligned
var badp := p + 1;              # now p is misaligned
var goodp := @next p;           # @next advances a pointer to the next item
goodp := @prev goodp;           # ...and back to the original item
```

You may have pointers to pointers, but remember that you can't take the address of a scalar variable, so they're of limited use.

```
record Structure is
        i: uint8;
        p: [uint8];
        pp: [[uint8]];
end record;

var v: Structure;
v.p := &v.i;
v.pp := &v.p;
[[v.pp]] := 7;
```

2.11 Special type tricks

You can define type aliases.

```
typedef ObjId is int(0, 127);      # a custom integer type (actually uint8)
typedef MyArray is ObjId[42];
```

These just define a new name for the type. Type aliases to the same type are compatible.

You can use @bytesof to return the size of any type or variable.

```
var array: MyArray;
MemZero(&block as [uint8], @bytesof MyArray);
MemZero(&block as [uint8], @bytesof array);        # this works too

var elementsize: intptr := @bytesof(@sizeof array); # use parentheses
```

2.12 Interfaces and implementations

There's an analogue of function pointers. It's stricter than in C; you may only take the address of subroutines which have been explicitly declared to be a member of a particular interface.

```
interface Comparator(o1: [Object], o2: [Object]): (result: int8);

sub PointerComparator implements Comparator is
    result := 0;
        if o1 >= o2 then
        result := 1;
        end if;
end sub;

sub StringComparator implements Comparator is
    result := StrCmp(o1 as [uint8], o2 as [uint8]);
end sub;

var someComparator: Comparator := PointerComparator;
if isString() != 0 then
    someComparator := StringComparator;
end if;
var r := someComparator(&object1, &object2);
```

Implementation references (e.g. PointerComparator or StringComparator above) are opaque pointer-sized objects, but not actually pointers. As with @impl, they use the parameter list of the interface. They behave exactly like normal subroutines and are constant values which can be used in initialiser lists.
The language syntax makes it possible but slightly hard to return a subroutine reference to an outer scope. Don't do this, because as the outer scope exits its variables will be reused and the subroutine reference's upvalues will become garbage. On the other hand, you can use them to call into a scope:

```
interface FileObserver(filename: [uint8]);
@decl sub ScanDirectory(path: [uint8], callback: FileObserver);

...

sub ShowDirectoryContents() is
    sub Callback implements FileObserver is
            print(filename);
                print_nl();
        end sub;

        ScanDirectory(".", Callback);
end sub
```

2.13 Inline assembly

Fragments of code written in assembly language can be inserted anywhere:

```
@asm "ld a,1";           # emitted literally

var regs_pair: uint16;
@asm "ld hl,(", regs_pair, “)”;# you may have references to simple variables

@asm "ld a,", p.i;         # not allowed!
```

Chapter 3 Control flow

3.1 Statements

An expression such as x = 0 or printf(...) becomes a statement when it is followed by a semicolon, as in

```
x = 0;
```

3.2 If Then Else

If-then-else statements can be used:

```
if x > 4 then
        y := 1;
elseif x == 4 then
        y := 2;
else
        y := 3;
end if;
```

Of course, if-then-else statements nesting is allowed.

3.3 Loops

Loops are also available, in two variants: the while and the loop.

```
while i != 0 loop
        x := x + 1;
end loop;
```

The break and continue work as in the C language:

```
loop
        if x < 0 then
                break;
        end if;
        if y > 5 then
                continue;
        end if;
        y := y + 1;
end loop;
```

The following conditional expressions may only be used in if and while statements:

```
and
or
not
==
!=
<
<=
>
>=
```

3.4 Case

The case statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly.

```
case v is   # p is a pointer to a string
            when 0: p := "   "; #Empty space
            when 1: p := " * "; #Star
            when 2: p := ">!<"; #Federation starbase
            when 3: p := "+K+"; #Klingon battlecruiser
            when else: p := "<*>"; #Your starship's position
end case;
```

Each ‘when’ ends with an implicit ‘break’, therefore the following code fragment, written in the ideea that for both 1 and 2, do_something will be executed, is wrong:

```
case n is
            when 1: # When n is 1, nothingh is executed, because the implicit ‘break’!
            when 2: do_something();
end case;
```

Chapter 4 – Subroutines

Subroutines are declared as:

```
sub name(input parameters): (output parameters) is
        <statements>
end sub;
```

Cowgol accepts subroutines with multiple input and output parameters:

```
# subroutine with one input parameter
sub ThisIsASubroutine(i: uint8) is
  # subroutine with no input or output parameters
  sub ThisIsANestedSubroutine() is
    print("nested subroutines can access upvalues!");
    print_i8(i);
  end sub;

  # subroutine with multiple output parameters
  sub swap(in1: uint8, in2: uint8): (out1: uint8, out2: uint8) is
    out1 := in2;
    out2 := in1;
  end sub;
  # calling a subroutine with multiple output parameters
  (i, j) := swap(i, j);
  return;  # does not take any parameters
end sub;
```

The compiler is strictly single pass, so if you want to use a subroutine before it's been defined you need to split the declaration and the implementation. It works like this.

```
sub CombinedDeclarationAndImplementation(i: uint8) is
  DoSomethingWith(i);
end sub;

@decl sub SplitDeclarationAndImplementation(i: uint8);

...arbitrary code here...
```

```
@impl sub SplitDeclarationAndImplementation is
  DoSomethingWith(i);
end sub;
```

Note that in the implementation, the parameters are the ones used in the declaration.

If you're using separate compilation, you can mark a subroutine as being external, with a link name, and the linker will resolve these.

In file1.cow:

```
sub DefiningAnExternal(a: uint8): (ret: uint8) @extern("routine1") is
  ret := a + 1;
end sub;
```

In file2.cow:

```
@decl sub ImportingAnExternal(i: uint8): (ret: uint8) @extern("routine1");
var x: uint8;
var y: uint8;
y := ImportingAnExternal(4);
```

Externals may only be defined at the top level.

It is important to mention that the parameters of the subroutines are loaded into registers, when possible.

E.g.: for the Z80 implementation of the Cowgol compiler, if the subroutine has only one parameter, the caller will load the register A, or HL, or HL and HL' , depending on the size of the parameter type. Any supplementary parameters will be pushed on the stack.

Also, it is important to know that the subroutine's code is responsible to "pop" from the stack these values.

The subroutine's code will store each parameter into a statically allocated space, owned by the subroutine.

This is why re-entrancy is impossible; therefore, no recursive subroutines are allowed, and also "circular" calls (e.g. Subroutine X calling subroutine Y, that calls back the subroutine X) are forbidden (the Cowlink linker will fail to build the executable…).

But, because it is possible to call C routines and assembler code from Cowgol programs, the re-entracy issue can be fixed easily.

There are some advantages of storing statically the values of subroutine parameters.

The executables are smaller, compared with the same programs written in C.

As an example:

- The Startrek game, written in C, results in a 27KB executable; ported to Cowgol, the executable has only 20KB.

The Cowgol executables are also faster, compared with their C implementation.

As an example:

- For a 272KB file, dumpx (written in C) takes 27 minutes to finish, while hexdump (written in Cowgol, with a similar output) takes only 7 minutes.

Chapter 5 Include files

Use the *include* keyword:

```
include "myfile.coh";
```

It is possible to use nested include files (myfile.coh may start with another include keyword...).

Chapter 6 Porting a C program to Cowgol

1. Beware, variables in Cowgol are not initialized to zero, as in the case of C global variables!

2. Cowgol has no floating point.

In certain cases, fixed point will be enough (see https://github.com/Laci1953/Cowgol_on_CP_M/tree/main/GAMES/Bowling).

3. Which C flow control structures must be changed:

There is no 'for' flow control structure in Cowgol, but 'while ... loop' may be used instead.
Example:

The C source fragment:

```
int n;

for (n = 0; n < 10; n++)
        ...
```

may be implemented in Cowgol as:

```
var n: uint16;
n := 0;
while n < 10 loop
        ...
        n := n + 1;
end loop;
```

Also, there is no 'do ... while' flow control in Cowgol, but 'while ... loop' may help.
Example:

The C source fragment:

```
int n;
n = 0;
do
{
        this();
        n++;
}
while (n < 10);
```

may be implemented in Cowgol as:

```
var n: uint8;
n := 0;
```

```
this(); n := n + 1;
while n < 10 loop
        this();
        n := n + 1;
end loop;
```

The 'switch ... case' flow control structure from C can be found in Cowgol as the 'case ... when'.
But, in Cowgol, each ‘when’ ends with an implicit ‘break’.

So, if you must port from C something similar with:

```
switch (v)
{
        case 0:
        case 1:
        case 2:
                do_this();
                break;
        case 3:
                do_that();
                break;
        default:
                err();
                break;
}
```

then you must use the following approach in Cowgol:

```
var v: uint16;
var tmp: uint16;

tmp := v;

if tmp == 1 or tmp == 2 then tmp := 0; end if;
case tmp is
        when 0: do_this();
        when 3: do_that();
        when else: err();
end case;
```

4. Cowgol has only one-dimensional arrays

What if our C source use a two-dimensional array?

A possible solution is to use two additional routines ('get value' and 'put value').
Example (from
https://github.com/Laci1953/Cowgol_on_CP_M/blob/main/GAMES/Startrek/starmain.cow):

```
#char quad[8][8];
# we have no two dimensional arrays, therefore...
var quad: int8[64];
```

```
sub get_quad(x: uint8, y: uint8): (ret: int8) is
            ret := quad[(y*8)+x];
end sub;

sub put_quad(x: uint8, y: uint8, val: int8) is
            quad[(y*8)+x] := val;
end sub;
```

5. Cowgol has no 'printf'

You can instead use a combination of some existing library functions.
Example:

```
var v: int16;

#printf("value is %d", v);
print("value is ");
print(itoa(v));
```

6. Cowgol has a limited set of 'string escape chars'

Only \r (CR), \n (LF), \t (TAB) are working in Cowgol.

7. Pointer arithmetic in Cowgol

In Cowgol, pointer arithmetic always works in bytes. But, you can use the @next and @prev operators.

8. No recursive subroutines in Cowgol.

But, you can combine such a recursive C subroutine with a Cowgol main program (see https://github.com/Laci1953/Cowgol_on_CP_M/tree/main/GAMES/Queens).