Dot Language

Full Syntax Guide

Version: *Draft 1.3 (Canonical)*

Philosophy:

Dot is a symbolic, pointer-oriented language that emphasizes manual control, clean scoping, and minimal syntax, designed to be transpiled into clean C++.

Core Symbols:

' = *non-ephemeral* pointer
" = dereference or array index access of non-ephemeral
\ = deallocation (e.g. 'x\)
@ = *pseudo-ephemeral* reference in function signature or body (left-hand side)— survives until }
@ = pseudo-ephemeral dereference in function body (right-hand side)
. = const reference in function signature or passed as argument
; = (re)assignment to non-ephemeral
: = const value assignment

Variables and Pointers:

```
i_ 'x; // Declare an int non-ephemeral pointer
i_ x = 3, // Declare + assign ephemeral pointer
i~ 'x; // Declare heap non-ephemeral
~x\ // Free heap memory and associated pointer
x" // Dereference and print value at address of non-ephemeral
     pointer
'x // Print non-ephemeral pointer address itself
```

Arrays:

```
i_5 'arr; // Declare static array of 5 ints
arr"2 = 7; // Assign index 2
arr"2 // Print value at index 2
i_6 arr;
arr2 = 4; // arr2 expires by; (outside functions) or } (inside
functions) (non-ephemerals only survive within function {}, it’s
cold outside)
```

Function Definitions:

```
if(i_ @x, i_ @y.) { // x is mutable, y is const so use . both
                    symbols are pseudo-ephemeral, const type always
                    at end
x@ = x@ + y@; // modifies a non-ephemeral outside of function, so
             ends in ;
} // pseudos die at closing }
```

```
y(i_ *e) { //function declaration without definition-yet- but must
exist somewhere in src/
```

Rules:
- Use ',' when no non-ephemeral memory is being reassigned to
- Use ; to end lines in which non-ephemeral memory is reassigned
- Use '.' after each const parameter, always at the end of the parameter list
- Single const parameters must still end with .

Function Calls:

```
f(‘a, ‘b.); // pass non-ephemerals by reference. Const
non-ephemerals take .
// function calls that modify non-ephemeral memory must end with ;
```

Rules:
- Use ',' for writable args
- Use '.' for const args (: at assignment to const)
- All args must be used inside the function or error
- Argument order matters: f('x, 'y:) != f('y: 'x)
- Terminate call with ; if non-ephemeral memory is modified by the function

Ephemerals:

```
i_ x = 5, // Ephemeral int, expires at next ; or \ (outside
function) or } (inside function)
```

Rules:
- Ephemerals are auto-deallocated at end of block (inside function), or next ‘ (outside function)
- Must not use ; after non-ephemeral declaration alone as only symbol is reserved (no assignment yet)
- Heap allocations cannot be ephemeral, but can be referenced by pseudos inside function signatures and bodies
- Ephemerals must not be dereferenced later (automatically die at ; \ or })

Sets (Namespaces):

```
set_i math { // set_ can allow any number of types, omni-type=set_
    add(i_ @a, i_ @b,) {
        a@ = a@ + b@; // modifies non-ephemeral pointer value
    } // death of pseudos by closing }
}

math.add('x, 'y); // non-ephemeral memory modified by function? End
call with ;
```

Struct:

```
struct_vec(i_ ‘x, i_ ‘y); // creating struct type ‘vec_’
i_ x” = 2;
i_ y” = 3;
vec_ point(x”, y”);
‘x\ ‘y\ // ‘x ‘y no longer used
point.x” // prints 2
```

Sets may restrict access to struct type:

```
set_vec linear_algebra{ // declared

i_ x”=2; i_ y”=3;
struct~heap_vec(i_ ‘x, i_ ‘y); // struct~ is heap, struct_ is stack
heap_vec~ myheapvec(x”,y”);
‘x\ ‘y\
```

Control Flow:

```
for (i_ i = 0, i < 5, i++) { arr"i = 1; } // assuming access to
‘arr

while (cond) { body }

if (cond) { body }

elif (cond) { body }

else { body }

except { handler } // like catch or SIGINT handling
```

Rules:

- Loop/branch args are ephemeral by default
- Follow ; rules where non-ephemeral memory is used or pseudo-ephemerals are passed

Threading:

```
$_ f” = set.function(i_ ‘var)... // blocking thread. Pointer ‘f
points to thread, evaluates as bool (false = still running, true on
completion)
$_ f” = set.function(i_ ‘var) //non-blocking thread
while(f” == false){} // blocks until thread complete
```

Memory and Deallocation:

- Use \ to free pointers manually:
```
'x\
```
- Heap allocations must be freed:
```
i~ 'buf; => ~buf\
```
- If not freed, compiler adds training wheels with --graceful flag
print

- Again, heap allocation is never ephemeral

Print

```
f"3 + x" // Print result of expression
'x // Print pointer address
x" // Print value pointed to by x
```

Rules

- Never use ; after print lines

Summary of ; Usage:

Use ; only when:
- Assigning to a non-ephemeral
- After non-ephemeral-modifying function call ();
- Never after }, \ or prints

Function Parameters and Memory Responsibility:

In Dot, function parameters are ephemeral in scope, but not in memory responsibility (*pseudo-ephemerals*).

- The symbols exist only within the function `{}`, but they point to memory that must be terminated.

Example:

```
add(i_ @x, i_ @y) {
x@ = y@ + 1; //pseudos modify non-ephemeral memory, terminate with
                 ;
     }
```

Comparison Table:

Type	Lifetime Must Free?	Terminator
Ephemeral	Never	Until ; or }
Non-ephemeral ‘ ‟	Always (straight after final use)	\
Pseudo-ephemeral *lhs rhs*	Always (straight after final } in function definition	\

This preserves **Dot's promise:** scope autonomy = scope responsibility.

The Central Dogma of *Pointer Oriented Programming* (POP):

D. A. U. T.

Declare -> Assign -> Use -> Terminate

String Arrays and s_ Types:
s_ types are treated like indexed character arrays and follow the same access pattern as numeric arrays.

Example:
```
s_ str” = "hey";
str"2 = "g";
str" // prints "heg"
```

Rules:
- s_ behaves like i_ with character data
- Can be indexed and reassigned using "n syntax
- Like all Dot types, must be terminated when no longer in use