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* reference
" = dereference or array index access
\ = deallocation (e.g. 'x\)
, = *ephemeral* or *pseudo-ephemeral* reference in function definition
. = const reference in function definition
; = writable argument in function call or full assignment of pointer
: = const argument in function call or const value assignment

Variables and Pointers:

```
i_ 'x; // Declare an int pointer
i_ x = 3, // Declare + assign ephemeral within function definition
i~ 'x; // Declare heap pointer
~x\ // Free heap memory
x" // Dereference and print value at address
'x // Print the 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
```

Ephemeral indexing uses same syntax, expires after ; (outside functions) or } (inside functions) (non-ephemerals only survive within function {}, it’s cold outside)

Function Definitions:

```
f(i_ 'x, i_ 'y.) { // x is mutable, y is const. both symbols are ephemeral, // so , . notation used x" = x" + y"; // modifies a non-ephemeral outside of function, // use ',' to show continuation } 'x\ 'y\ // must always terminate reference symbols (pseudo-ephemerals)

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

Rules:

- Use ',' between mutable parameters

- Use '.' between const parameters
- Single parameters must still end with , or .

Function Calls:
```
f('a; 'b:); // pass non-ephemerals by reference
// function calls that modify non-ephemeral memory must end with ;
```

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

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

Rules:
- Ephemerals are auto-deallocated at end of block, or next ; or \ outside function
- Must not use ; after declaration as only symbol is reserved,
- Heap allocations cannot be ephemeral
- Must not be dereferenced later (automatically die at ; \ or })

Sets (Namespaces):
```
set math {
    add(i_ 'a, i_ 'b,) {
        a" = a" + b"; // modifies non-ephemeral pointer value
    } 'a\ 'b\
}

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

Control Flow:
```
for (i_ i = 0, i < 5, i++) { arr"i = 1; } // terminate 'arr after
} if no longer used

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

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 a non-ephemeral
- Separating args in function calls (not defs)
- After function call ();
- Never after }, \ or print lines

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; // modifies non-ephemeral memory, terminate with ;
     } 'x\ 'y\
```

Comparison Table:

Type	Lifetime Must Free?	Terminator
Ephemeral	Never	Until ; or }

Non-ephemeral and heap	Always (straight after final use)	\
Pseudo-ephemeral	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 if declared as a pointer or passed to functions