Gosilang MVP — Grammar, Macros, and C Skeleton (bind/unbind, vec/span, NIL/NULL)

This captures a *minimal but working* formal layer for Gosilang that you can evolve inside the RIFT toolchain. It defines:

- Tokens & grammar (EBNF) for `!`-invocation, `#def` macros, `#bind/#unbind`, `vec`, `span`, `range`.
- `NIL` vs `NULL` semantics (lattice-aware sentinel vs outside-control-space).
- Parallel **lazy diff** execution model for `#bind(EVERYTHING, UNIVERSE)`.
- Vector & unit lattice primitives (mag, norm, dot, cross; span in [-1,1]).
- A compact C MVP skeleton: lexer → macro registry → evaluator → parallel bind.

> *Scope: homogenous vectors first; heterogeneous can be layered by tagged tuples.*

---

1) Lexical & Tokens

Delimiters: `( ) < > [ ] { } , : ;`

Operators: `!` (invoke), `#` (compiler directive), `:=` (bind-to-name), `=` (assign literal), `->` (macro transform)

Keywords: `#def`, `#bind`, `#unbind`, `span`, `range`, `vec`, `nil`, `null`

Identifiers: `[A-Za-z_][A-Za-z0-9_]*`

Numbers: decimal integers; optional `.` for floats; scientific later.

---

2) Core Types & Sentinels

- **Scalar**: `int`, `float`.
- **Vector**: `vec<N>` homogeneous numeric.
- **Span**: normalized axes in **[-1, 1]**; used for lattice/complex boundary checks.
- **Range**: interval `[a..b]` in original units; can normalize to span.
- **Unit lattice**: tags carried as metadata; unit math is multiplicative (unit×unit=unit²); conversions as explicit macros.
- **NULL**: *outside control space* — no memory, no physics.
- **NIL**: *inside lattice but unbound/intentional no-allocation* sentinel; safe to carry in vectors; meaningful in span checks.

---

3) EBNF (Minimal)

```
Program      := { Stmt } ;
Stmt         := DefStmt | BindStmt | UnbindStmt | AssignStmt | ExprStmt ;
DefStmt      := "#def" "[" MacroSig "->" MacroExpr "]" ;
MacroSig     := Ident "(" [ ParamList ] ")" ;
ParamList    := Ident { "," Ident } ;
BindStmt     := "#bind" "(" Expr "," Expr ")" ;
UnbindStmt   := "#unbind" "(" Ident ")" ;
AssignStmt   := Ident ":=" Expr ;
ExprStmt     := Expr ;
Expr         := Invoke | Vector | Span | Range | Ident | Number ;
Invoke       := "!" ( Ident | "<" TagList ">" ) "(" [ ArgList ] ")" ;
TagList      := Ident { "," Ident } ;
ArgList      := Expr { "," Expr } ;
Vector       := Ident "<" DimList ">" "(" ArgList ")"   // e.g., vec<3>(1,2,3)
DimList      := Number { "," Number } ;
Span         := "span" "[" Expr ".." Expr "]" ;
Range        := "range" "[" Expr ".." Expr "]" ;
```

Notes:

- `!vec<...>(...)` is sugar for a vector constructor with normalization capability (via macro).
- `!<x,y,z>(a,b,c)` allows axis-tagged construction without naming a type; the compiler infers `vec<3>`.

---

4) Macro System (`#def[...]`)

Design: `#def` introduces hygienic macros that transform call AST → expression AST.

Examples:

```
#def[ mag(v) -> sqrt(sum(v[i]*v[i] for i in 0..len(v)-1)) ]
#def[ norm(v) -> v / mag(v) ]
#def[ vec(args...) -> norm(vec_construct(args...)) ]
#def[ dot(a,b) -> sum(a[i]*b[i] for i in 0..len(a)-1) ]
#def[ cross(a,b) -> vec(
  a[1]*b[2]-a[2]*b[1],
  a[2]*b[0]-a[0]*b[2],
  a[0]*b[1]-a[1]*b[0]
) ]  // defined only when len(a)=len(b)=3
```

Axis-tagged vector:

```
#def[ <x,y,z>(ax,ay,az) -> vec(ax,ay,az) ]
```

Unit helpers (sketch):

```
#def[ yards_to_miles(y) -> y / 1760.0 ]
#def[ mph_to_mps(v_mph) -> v_mph * 0.44704 ]
#def[ F(m,a) -> m*a ]  // force; m has mass units, a has accel units
```

> *Heterogeneous vectors use tagged tuples in a later phase; MVP keeps vectors numeric while attaching unit tags in metadata.*

5) `!` Invocation Semantics

- `!vec<3>(1,2,3)` → constructs `vec<3>` then normalizes via macro `vec(...)` → `norm(vec_construct(...))`.
- `!<x,y,z>(a,b,c)` → tag-driven sugar → `vec(a,b,c)`.
- Overlong/short argument lists are compile errors.

Complex boundary: when computing in span space, any operation yielding < 0 under square-root lifts into complex domain `(re, im)`; domain tag recorded.

6) Span & Range

- `span[s..t]` normalizes into [-1,1] by affine map; used for lattice navigation and NIL-placement logic.
- `range[a..b]` keeps native units and can be mapped to `span` by `to_span(range)`.

NIL placement: `NIL` may encode "present but out-of-real-sector"; math ops treat `NIL` as *skip* for reductions, or projectable sentinel if an explicit macro requests a projection.

7) Bind/Unbind — Lazy Parallel Diff

Intent: `#bind(EVERYTHING, UNIVERSE)` expresses *lazy, non-cloning* parallel map

```
Δ[i] := EVERYTHING - UNIVERSE[i]
```

- No data cloning; `UNIVERSE` remains read-only during the bind window.
- `NIL` elements yield `NIL` deltas unless an explicit projection macro is applied.
- Execution model: chunked parallelism over isolated items; no shared mutable state.

Decoherence/Collapse:

- Optional attribute: `@cohere(ms)`; when elapsed > ms, the lazy computation collapses to concrete values and the bind is released.
- `#unbind(EVERYTHING)` explicitly tears down the bind relation before timeout.

Errors:

- Different lengths → compile error.
- Type mismatch (scalar vs vector) → compile error.

8) Worked Examples

A) Vector construction

```
let V := !vec<3>(24,6,4)    // normalized vector
let M := mag(V)             // = 1 by construction
```

B) Axis-tagged sugar

```
let P := !<x,y,z>(1,1,1)    // same as vec(1,1,1) then normalized
```

C) Units & magnitude (sketch)

```
let yards := 16750
let miles := yards_to_miles(yards)  // 9.517...
let R := range[0..miles]
let S := to_span(R)                 // normalized [-1,1]
```

D) Bind diff (the 42 − universe example)

```
let EVERYTHING := 42
let UNIVERSE := vec(23,45,67,2,5)
#bind(EVERYTHING, UNIVERSE)   // lazy Δ
// Evaluate Δ concretely → [19,-3,-25,40,37]
#unbind(EVERYTHING)
```

---

9) C MVP Skeleton (single file demo)

> *Purpose: prove the semantics — not a full compiler. It lexes just enough to demo `vec`, `mag`, `norm`, and the `bind` parallel diff with `NIL` handling.*

```
// gosilang_mvp.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <stdatomic.h>

// —— Sentinels ——
#define NIL_PTR ((void*)-1)

typedef struct { double *data; size_t n; } vec_t;

typedef struct { double re, im; } cnum_t; // for future complex work

static double mag(const vec_t *v) {
    double s = 0.0; for (size_t i=0;i<v->n;i++) s += v->data[i]*v->data[i];
    return sqrt(s);
}

static vec_t norm(vec_t v) {
    double m = mag(&v); if (m==0.0) return v;
    for (size_t i=0;i<v.n;i++) v.data[i] /= m; return v;
}

static vec_t vec_make(size_t n, double *src) {
    vec_t v = { .data = malloc(sizeof(double)*n), .n = n };
    for (size_t i=0;i<n;i++) v.data[i] = src[i];
    return v;
}

// —— Bind diff ——
typedef struct { const double *universe; double everything; double *out; size_t from, to; } shard_t;

static void* shard_run(void *p) {
    shard_t *s = (shard_t*)p;
    for (size_t i=s->from; i<s->to; i++) {
        // NIL support: here we interpret NaN as NIL for numeric MVP
        double u = s->universe[i];
        if (isnan(u)) s->out[i] = NAN; else s->out[i] = s->everything - u;
    }
    return NULL;
}

static void parallel_diff(double everything, const double *universe, size_t n, double *out) {
    const size_t T = 4; // fixed threads for MVP
    pthread_t th[T]; shard_t shards[T]; size_t step = (n+T-1)/T;
    for (size_t t=0;t<T;t++) {
        size_t a = t*step, b = (a+step<n)?(a+step):n;
        shards[t] = (shard_t){ .universe=universe, .everything=everything, .out=out, .from=a, .to=b };
        pthread_create(&th[t],NULL,shard_run,&shards[t]);
    }
    for (size_t t=0;t<T;t++) pthread_join(th[t],NULL);
}

int main(void){
    // vec / norm demo
    double raw[3] = {24,6,4};
    vec_t v = vec_make(3, raw); v = norm(v);
    printf("mag(v) = %.6f\n", mag(&v));

    // bind diff demo: 42 - [23,45,67,2,5]
    double uni[5] = {23,45,67,2,5};
    double out[5];
    parallel_diff(42.0, uni, 5, out);
    for(size_t i=0;i<5;i++) printf("%s%.0f", i?", ":"[", out[i]);
    printf("]\n");

    free(v.data);
    return 0;
}
```

NIL in numeric MVP: we encode `NIL` as `NaN` for arrays. In pointer-bearing structures, use `NIL_PTR`.

10) Infrared Mapping (next module hook)

Goal: convert RGBA → wavelength-space histogram (including IR beyond visible). Keep it as a separate GOSI module:

```
#def[ rgba_to_lambda(r,g,b,a) -> /* calibrated mapping */ ]
#def[ ir_project(img) -> histogram(lambda in [700nm..1100nm]) ]
```

- Keep the core language agnostic of color; expose this via library macros so the compiler stays small.

11) Compliance Notes

- No cloning: `#bind` forbids deep copies; all ops are computed-on-read, chunked in parallel.
- Isolation: per-element processing; no shared writeable state.
- Determinism: functions are pure; wall-clock only in `@cohere(ms)` scheduling.

12) Next Steps

1. Add a tiny macro expander (string/AST) so `#def` examples are executable in the MVP.
2. Swap `NaN` → explicit tagged value to distinguish `NIL` from numeric NaN.
3. Add complex-domain lift for span sqrt(<0) cases (`cnum_t`).
4. Wire a `.gs` front-end that generates the MVP IR; RIFT can later own the full pipeline.