# Core Lexer Architecture with PhenoMemory

```c
// gosilang_pheno_lexer.h
#include <ctype.h>  // For character classification
#include <stdint.h>

// Phenomenological token structure
typedef enum {
    PHENO_ACTOR_STATE,      // Actor isolation states
    PHENO_MEMORY_STATIC,    // Static memory allocation
    PHENO_MEMORY_DYNAMIC,   // Dynamic memory allocation
    PHENO_IOTA_DEFAULT,     // Default placeholder (from Go's iota)
    PHENO_ISOLATED,         // Hardware isolation marker
    PHENO_MESSAGE_CHANNEL   // Inter-actor communication
} PhenoTokenType;

typedef struct PhenoTokenValue {
    union {
        int32_t     i32_val;    // For `var count: i32`
        bool        bool_val;    // For `isolated: true`
        uint32_t    iota_val;    // For enumerated constants
        void*       actor_ref;   // Actor reference
    } data;

    PhenoTokenType type;
    size_t memory_weight;        // For AVL balancing
} PhenoTokenValue;

// AVL-Trie hybrid node for O(log n) lookup
typedef struct PhenoMemoryNode {
    // AVL properties
    int height;
    int balance_factor;

    // Trie properties
    char* prefix;
    struct PhenoMemoryNode* children[256];  // For byte-indexed trie

    // Phenomenological data
    PhenoTokenValue token;

    // Hash table link for O(1) auxiliary lookup
    struct PhenoMemoryNode* hash_next;
} PhenoMemoryNode;
```

# Dynamic Lookup Table with Space/Time Optimization

```
// Optimized lookup table with O(log n) search, O(1) auxiliary space
typedef struct PhenoLookupTable {
    // Primary AVL-Trie for ordered access
    PhenoMemoryNode* avl_root;

    // Auxiliary hash table for O(1) average case
    PhenoMemoryNode* hash_buckets[1024];  // Fixed size = O(1) auxiliary space

    // Space/Time metrics
    size_t total_nodes;
    double space_time_ratio;  // log(n) / 1 for optimization tracking
} PhenoLookupTable;

// Lookup algorithm achieving log(n) time with O(1) aux space
PhenoTokenValue* pheno_lookup(PhenoLookupTable* table, const char* key) {
    // First try O(1) hash lookup
    uint32_t hash = hash_function(key) % 1024;
    PhenoMemoryNode* node = table->hash_buckets[hash];

    while (node) {
        if (strcmp(node->prefix, key) == 0) {
            return &node->token;  // O(1) hit
        }
        node = node->hash_next;
    }

    // Fallback to O(log n) AVL-Trie search
    return avl_trie_search(table->avl_root, key);
}
```

# Actor Type System with Iota

```
// Actor type definitions using iota pattern
typedef enum {
    ACTOR_ISOLATED = 0,  // iota starts at 0
    ACTOR_SHARED,        // = 1
    ACTOR_HYBRID,        // = 2
    ACTOR_MEDICAL,       // = 3 (for medical_monitor.gs)
    ACTOR_INTERSTELLAR   // = 4 (for interstellar.gs)
} ActorIsolationState;

// Canonical form for actor definitions
typedef struct GosilangActor {
    char* name;
    ActorIsolationState isolation;  // Uses iota enumeration

    // Key-value pairs for properties
    struct {
        char* key;
        PhenoTokenValue value;
    } properties[MAX_PROPERTIES];

    // Message channel for AVL rotation-based communication
    struct MessageChannel* channel;
} GosilangActor;
```

# Message Channel with AVL Rotation

```
// Message channel using AVL rotations for load balancing
typedef struct MessageChannel {
    PhenoMemoryNode* message_queue;  // AVL tree of messages

    // Rotation-based load balancing
    void (*rotate_left)(PhenoMemoryNode**);
    void (*rotate_right)(PhenoMemoryNode**);

    // O(log n) insertion with automatic balancing
    void (*send_message)(struct MessageChannel*, PhenoTokenValue*);
} MessageChannel;

// AVL rotation for message priority balancing
void avl_rotate_for_balance(PhenoMemoryNode** root) {
    if ((*root)->balance_factor > 1) {
        if ((*root)->children[0]->balance_factor < 0) {
            // LR rotation
            rotate_left(&(*root)->children[0]);
        }
        rotate_right(root);  // LL rotation
    }
    // Similar for right-heavy case
}
```

# Heterogeneous vs Homogeneous Type Handling

```
// Type classification for Gosilang compilation
typedef enum {
    TYPE_HOMOGENEOUS,   // Same type collection (e.g., all i32)
    TYPE_HETEROGENEOUS  // Mixed types (actors with different properties)
} TypeClassification;

// Canonical form resolver
PhenoTokenValue canonicalize_type(const char* type_str) {
    PhenoTokenValue result;

    if (strncmp(type_str, "i32", 3) == 0) {
        result.type = PHENO_MEMORY_STATIC;
        result.data.i32_val = 0;  // Default
    } else if (strcmp(type_str, "isolated") == 0) {
        result.type = PHENO_IOTA_DEFAULT;
        result.data.iota_val = ACTOR_ISOLATED;
    } else if (strcmp(type_str, "bool") == 0 ||
               strcmp(type_str, "true") == 0 ||
               strcmp(type_str, "false") == 0) {
        result.type = PHENO_MEMORY_STATIC;
        result.data.bool_val = (strcmp(type_str, "true") == 0);
    }

    return result;
}
```

# Integration with ctype.h for Token Classification

```
// Enhanced tokenizer using ctype.h
TokenType classify_token_with_ctype(const char* str) {
    if (isalpha(str[0]) || str[0] == '_') {
        // Check if it's an actor keyword
        if (strcmp(str, "actor") == 0) return TOKEN_ACTOR;
        if (strcmp(str, "isolated") == 0) return TOKEN_ISOLATED;

        // Check for type identifiers
        for (int i = 0; str[i]; i++) {
            if (!isalnum(str[i]) && str[i] != '_') break;
        }
        return TOKEN_IDENTIFIER;
    }

    if (isdigit(str[0])) {
        return TOKEN_NUMBER;
    }

    // Use ispunct() for operators
    if (ispunct(str[0])) {
        return TOKEN_OPERATOR;
    }

    return TOKEN_UNKNOWN;
}
```

# Implementation Notes

1. **Space/Time Optimization**: The hybrid approach uses:
    - Primary AVL-Trie: O(log n) guaranteed worst-case
    - Auxiliary hash table: O(1) average case with fixed space
    - Total auxiliary space: O(1) as hash table size is constant
2. **Iota Pattern**: Following Go's convention:

```
// Automatic enumeration like Go's iota
#define IOTA_BEGIN(name) typedef enum name {
#define IOTA_END(name) } name##_t;

IOTA_BEGIN(ActorState)
    ISOLATED,  // = 0
    SHARED,    // = 1
    HYBRID     // = 2
IOTA_END(ActorState)
```

3. **Memory Layout**: Actors use isolated memory regions:

```
// Each actor gets its own memory segment
void* allocate_isolated_actor_memory(size_t size) {
    // Allocate with hardware isolation if available
    return mmap(NULL, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
}
```

This architecture ensures your lexer maintains the O(log n) lookup performance while keeping auxiliary space at O(1), perfect for the thread-safe, phenomenologically-aware Gosilang compilation pipeline.