An Informal Introduction To MIGR (Multilayered Intermediate Graph Representation) For Creolynator

~ By Aliqyan Abid

Index

Prologue: The Problem

One day I forgot where I kept my notes...

This is not dramatic. This is ordinary. A *.md* file I made in vim, scattered across a filesystem, divorced from context, orphaned. I could search for it. I could find it eventually. But the file itself contained no memory of its connections to other files, to ideas, to the network of thought from which it emerged.

This is the fundamental condition of most note-taking systems: they preserve content but destroy context. They are drawers, not libraries. Containers, not ecosystems.

And then, I discovered **Obsidian**. It solved the problem elegantly. Backlinks, tags, a visual graph of connections. Notes could reference other notes. The container became a network.

But Obsidian was not *mine*. It was polished, capable, closed. I wanted to understand how it worked. More importantly, I wanted to build something that embodied a principle: that tools should be as transparent as they are powerful. That the user should see into the machinery, not be insulated from it.

This is where **Creolynator** began...

Chapter 1: Why Creole, Why Parse, Why Trees

1.1 The Markdown Problem

When I decided to parse markdown, I encountered vagueness, and many edge cases.

Markdown is intentionally loose. A philosophy of simplicity. But this looseness becomes ambiguity at scale. Two parsers may produce different interpretations of the same markdown like **github** markdown is different and **slack** markdown is different, both are subsets of markdown, so u can use or make your subset in markdown, but if u parse whole markdown, then there are many edge cases so u will have to make your own rules anyway. The specification is more guideline than law. This is not a bug, it is the design. Markdown was written to be human readable ASCII, not to be machined.

But when you want to build a system on top of markup, ambiguity is costly. Every edge case requires a decision. Every decision compounds.

1.2 Why Creole

Creole is different. It has rules. Not many, but firm. A heading is `= text =`. A link is `[[target]]`. A tag is `#tag`, currently in our case [[#tag]]. Most importantly; for me as the eventual writer and parser of it; the syntax is constrained. The grammar is unambiguous. Which is great.

This was not accident. Creole was designed by a consortium to solve exactly this problem: create markup that is both human readable *and* formally parseable.

When I switched to Creole, I gained something: a grammar I could trust.

1.3 The Tree (AST)

If I Parse Creole it will produce a tree. This is natural. Parsing any language produces a tree. The tree mirrors the document's structure: headings contain paragraphs, paragraphs contain text and links, links contain URLs.

This tree is an Abstract Syntax Tree (AST). It abstracts away the syntax (the `[[` and `]]`) and preserves structure and content.

An AST is powerful. You can traverse it, transform it, generate output from it. Every compiler, every language processor, uses ASTs.

But...for note-taking, the AST was not enough.

Chapter 2: The Insufficiency of Trees

2.1 What Trees Cannot Express

A tree is a hierarchy. Parent-child relationships. Containment. Order.

But a note-taking system must express relationships that are not hierarchical:

- A note tagged `#project-x` is grouped with other notes tagged `#project-x`. This is not containment. The tag does not contain the notes. Rather, the notes share a label.

- A note that links to another note creates a connection that is not structural. The linking note does not contain the target. They are siblings in a graph, not parent-child in a tree.

- A note's "backlinks" i.e. the set of notes that link to it, form a pattern of reference. This pattern is not expressed in the tree. It must be reconstructed by querying all other notes.

A tree cannot natively express these relationships because a tree has a single root, a single path from root to any node. But in a semantic network, a node can be reached from many paths. It can belong to many groups. It can be a reference and be referenced.

2.2 The Graph

Tool like *Obsidian* solved this by adding a layer. The AST remains (it still parses the document into a tree). But overlaid on top is a semantic graph: nodes for tags, nodes for references, edges connecting them.

This is not new. Knowledge graphs, semantic webs, RDF triples, all express this idea: meaning is not just hierarchy, but relationship.

But most implementations make a choice: either you have the hierarchy (the tree) or you have the semantic network (the graph). They are separate systems. The tree tells you what is *in* the document. The graph tells you what the document *means*.

They speak different languages. Synchronizing them is manual labor.

2.3 The Insight: Layers

Here is where I saw something. Not something new in absolute terms, others have thought this way, but something *right* for my problem:

What if the tree and the graph were not separate systems, but layers of the same representation?
What if a single node could participate in both hierarchies and networks simultaneously?
What if you could traverse either layer independently, or query across both?

This is not revolutionary. Graph databases and hypergraphs have existed for decades. But it is uncommon in document processing. Most IRs choose a side: either AST-based (trees) or semantic-based (graphs). Few integrate both.

MIGR chooses integration.

Chapter 3: The Birth of MIGR

3.1 What MIGR Is (Conceptually)

MIGR is **Multilayered Intermediate Graph Representation**. Let's break this down:

- **Intermediate:** It is not the source markup. It is not the output format. It is the representation between input and output. The IR.
- **Graph:** It is fundamentally a graph. Nodes and edges. Not a tree.
- **Multilayered:** But it is not a single graph. It is multiple graphs, layered, each with its own semantics, yet interconnected.
- **Representation:** It represents a document. Not just its content, but its structure, its meaning, its connections.

3.2 The Layers

Structural Layer: This is the tree. Headings, paragraphs, lists, images. The document's outline. What you see when you collapse all formatting and look at the skeleton.

This layer answers: What is the document *about* structurally? What is the hierarchy?

Inline Layer: Nested within the structural layer, this is formatting. Bold, italic, links, code. The fine grain. The texture.

This layer answers: How is content *formatted* within blocks?

Semantic Layer: This is the graph. References, tags, backlinks, cross-references. The web of meaning.

This layer answers: What does the document *mean*? What does it reference? What tags does it belong to? What other notes reference it?

3.3 Why Layers?

Why not one unified graph?

Because concerns are different. Structuring a document, organizing it into sections is not the same as tagging it. Formatting text is not the same as linking to another document.

By separating layers, each layer can have its own model, its own query logic, its own traversal. They do not interfere with each other. They can be updated independently.

This is the mechanical clock principle I have. It states that, from outside, a clock is simple: you read the time. Inside, hundreds of gears mesh in precise order. Each gear has a function. Each layer of gearing has a purpose.

Change one layer, and the others are not affected, unless they are explicitly connected, then ofc there will be changes, but coordinated.

3.4 The Node Abstraction

Every node, across every layer, is the same object. It is a shared interface containing:

- An identity (unique ID)
- A type (what kind of thing it is)
- Content (what it contains)
- Metadata (information *about* it)
- A version (how many times it has changed)
- A hash (what its content is, distilled)

And it participates in two kinds of edges:

- Structural edges (parent-child, only in structural and inline layers)
- Semantic edges (cross-references, only in semantic layer, but can reference any node)

This unified abstraction means: you can write generic code that works on any layer. A visitor pattern, a traversal algorithm, a serialization routine; these do not need to know if they are working on structure or semantics. Because they just work on nodes and edges which are same for all layers!

Chapter 4: How MIGR Is Different

4.1 Versus Traditional ASTs

AST: Parse → Tree → Traverse tree → Generate output.

Linear pipeline. One representation.

MIGR: Parse → Build structural tree → Extract semantic graph → Query across layers → Generate output.

Multi-stage, multi-representation pipeline. But unified node model.

Consequence: ASTs are focused on *structure*. They answer "what is the document?" MIGR answers "what is the document? what does it mean? what is it related to?"

4.2 Versus Semantic Webs and RDF

RDF: Everything is a triple. Subject-Predicate-Object. Uniform, but abstract.

```
note1 references note2
note1 hasTag project-x
note2 hasTag project-x
```

Very flexible. But disconnected from the document's structure. An RDF database does not know that note1 is a heading followed by a paragraph. That information is lost or stored separately.

MIGR: Structure and semantics coexist. A HEADING node is both structural (child of DOCUMENT_ROOT) and semantic (can be tagged, referenced, linked).

4.3 Versus Obsidian's Model

Obsidian (as I understand it) has a document tree and a semantic graph as separate systems. They are connected, you can visualize both, but they are implemented separately. The document is parsed into a tree. Then, separately, links and tags are extracted and stored in a graph.

MIGR: The node model unifies them. Both tree edges and semantic edges attach to the same nodes. Querying is uniform. The separation is logical (different layers) not physical (different data structures).

Consequence: MIGR makes it easy to ask questions like "what are all the headings tagged #important?" Because headings are real nodes that can be tagged, not proxies in a separate system.

4.4 Versus Hypergraphs

A hypergraph allows edges to connect multiple nodes at once. `edge(node1, node2, node3)` is valid.

MIGR for now uses only simple edges (one-to-one). This is less expressive than a full hypergraph, but also simpler to implement and reason about.

Trade-off: Some semantic relationships would be more naturally expressed as hyperedges (e.g., "these three notes are related to project X as a unit"). But for note-taking, pairwise relationships are usually sufficient. A note can be tagged multiple times. Multiple tags can be searched. And thus the same effect emerges before your beautiful eyes.

Chapter 5: Philosophical Vision

5.1 The Mechanical Clock

A mechanical clock is: "a complex machine that is easy to use but inside something very complex is going on."

This is my aesthetic...

A mechanical clock has no electronics. No software. But it is sophisticated. Gears mesh with precision. Springs store energy. Escapements regulate release. Everything is visible. If you open the case, you can see how it works. Yet most people just need only read the time. They do not need to understand the mechanism.

This is what I want MIGR to be.

For the user: A tool that lets you write notes, link them, tag them, search them. Simple interface. Natural workflow.

For the craftsperson: A representation that is transparent. You can see the layers. You can understand how a document is structured, what nodes exist, what relationships are drawn. You can build on it! Extend it! Modify it!

For the scholar: A formal model. Clear semantics. Predictable behavior. No magic.

Most software today is not like this. It is opaque. Black box. The user interacts with the surface. The internals are hidden. You cannot understand how the system works because it is not designed to be understood. It is designed to be...just used.

MIGR *is designed differently.*

5.2 Simplicity Outside, Complexity Inside

One of my favourites, **Mr. Richard Stallman Sir** has argued something on the lines of "software freedom matters because users are not passengers. They should be able to understand and modify their tools."

MIGR is written in this spirit. Not perfectly ofc, is any system perfect?, but in spirit.

The code is readable. The model is learnable. A person can study MIGR and understand what it does. They are not left guessing.

This in my opinion is quite rare. Most systems are designed for scale and performance, not for comprehension, so if we achieve this, then best for us.

5.3 Nature as Metaphor

A humble thought: "on outside all looks so simple, but from inside, it is all so complex all the small things."

Nature works this way. A cell appears as a unit. Simple. But inside, organelles and proteins and DNA perform countless operations. An organism appears as a whole. But it is composed of trillions of cells, each doing its work.

The beauty is that this complexity is *organized*. It is not chaos. It is layered. Systems within systems. Each layer has a function. Each function serves the whole.

MIGR aspires to this. Not natural complexity, not randomly created or evolved. But organized complexity. Layers that each do one thing well.

Chapter 6: The Problem Space MIGR Addresses

6.1 Traditional Note-Taking

Most note-taking systems, even good ones, optimize for capture, not connection.

You write a note. It is stored. You can search for it later. But the relationships between notes are secondary. Tags are metadata. Links are just text. Backlinks are computed on demand.

This works for small scale. A few hundred notes, a few dozen tags. But the relationships matter more as the system grows. With a thousand notes, structure becomes crucial.

6.2 Obsidian's Contribution

Obsidian (and **Roam Research**, and others) realized: the relationships *are* the system. The graph of connections is not a feature. It is the core.

This was a philosophical shift. Note-taking is not filing. It is knowledge construction. Knowledge is a web. And these tool are eventually doing knowledge management.

6.3 MIGR's Position

MIGR takes this further, but also steps back.

Further: By unifying the structural and semantic layers, MIGR makes it natural to query across both. "Show me all headings with tag X" is not a special case. It is a natural question to ask.

Step back: MIGR does not claim to be the final representation. It is an IR, an intermediate step. From Creole source to MIGR to output (HTML, PDF, EPUB or custom formats). Each step transforms the representation.

This is humble. MIGR is a just an IR for a specific job: parsing structured markup and making its meaning queryable.
And Creolynator is something that will use this IR. The most important part of our 'clock' will be MIGR.

Chapter 7: What MIGR Learns From Others

7.1 From Compiler Design

Compilers invented IRs. The insight: you do not transform source code directly to machine code. You transform source to IR, then IR to code. The IR is simpler than either end point.

MIGR borrows this. Parse Creole to MIGR. Then MIGR to HTML, PDF, database, visualization.

7.2 From Database Design (idea courtesy of my work colleague *Rishik Ram*)

Databases learned: separate schema from query language. Define the data structure once. Then query it in many ways.

MIGR does this with layers. Define the structure (what nodes exist, what edges exist). Then query using any traversal strategy.

7.3 From Graph Theory

Graphs are old. But their application to knowledge representation is relatively new. I learned that graphs are more natural for representing the meaning. Not trees.

7.4 From Unix Philosophy

Keep tools small and composable. Each tool does one thing well. Combine tools with pipes.

MIGR aspires to this. Each layer is independent. Each query operation is a single purpose. Combine them to answer complex questions.

Chapter 8: What MIGR Is Not

8.1 Not a Database

MIGR is an in-memory representation. It is not persistent by default. It is not optimized for billions of nodes. It is not a database engine.

This is intentional. MIGR is for document-scale problems. A single document, or a few documents, with links between them. Not enterprise data.

8.2 Not a Semantic Web Engine

MIGR is not RDF, not OWL, not a semantic web standard.

It is simpler. More specific. Designed for notes, not for arbitrary knowledge representation.

8.3 Not a Replacement for Obsidian

MIGR is an **IR**. Obsidian is a UI, a sync service, a community, an ecosystem.

MIGR could power a note-taking system. *Obsidian could use* MIGR as an IR. But they are different layers of abstraction.

8.4 Most Importantly - Not Finished

Right now it is what it is, maybe one day later, I will add code or refactor and make it better, and two days later I will make it better still, it works on this philosophy.

And this is honest. MIGR is early. The traversal layer is not yet written. The builder API is not yet written. Deserialization does not work. Optimization has not been attempted.

It is a foundation. A good foundation surely, but not a house.

Chapter 9: Why Pragmatism Matters

I built MIGR incrementally. I did not spend a year in theory. I built, and I learned and the design emerged.

According to me, this is not a failure of planning. This is **pragmatism**.

Theory without building is speculation. Building without theory is flailing. The right approach is: understand the space, sketch the design, build to learn, refactor when patterns emerge.

I did this.

The result: a system that is not theoretically perfect, but practically sensible. The layers are clean, but not over-engineered. The node model is flexible, and not baroque. The code is readable.

Chapter 10: Open Problems and Future Directions

10.1 Persistence

MIGR exists in memory. Serialization is incomplete (deserialization is not implemented). This must be fixed.

When it is, MIGR can be stored to disk, loaded back, modified, saved. A true document format.

10.2 Incremental Updates

Currently, parsing rebuilds everything from scratch. Errr... For real-time editing (as you type), this is wasteful.

Future: Parse only changed regions, reuse unchanged nodes, update semantic graph incrementally.

10.3 Extensibility

Currently, node types and edge types are enums. i.e., Fixed. To add a new semantic type, you modify the code.

Future: Allow user-defined node and edge types. Let systems built on MIGR extend the model.

10.4 Performance

MIGR is not optimized. Queries are O(n). ID generation is not thread-safe. Hashing is weak for now.

This is fine for now. But I do realise that as documents grow larger, optimization becomes necessary. MIGR is in safe hands.

10.5 Formal Semantics (Literature)

So, MIGR for now has an informal description (this document, the code). It lacks formal semantics.

Formal semantics would let us prove properties: "traversing structural edges and following backlinks produces the same result as...?" These proofs would give confidence.

Chapter 11: On Building for Others

Initial idea was: "I wanna do something of my own."

This is important. MIGR is taking birth because I was frustrated. I wanted something better.

But building for myself and building for others are different.

For myself, I can accept quirks. I know the system. I know when to regenerate caches. I know what metadata fields to set.

For others, every quirk is a bug. Every assumption is a surprise.

As MIGR matures, this will matter. The traversal layer, the builder API—these are not features. They are clarity. They are ways of saying: here is how to use this system.

And documentation. Tests. Examples.

These seem tedious compared to coding. But they are how you communicate with the future. Your future self. Other people. And I do need other people for this surely.

Chapter 12: Conclusion

MIGR is a response to a real problem: notes scattered, connections lost, structure and meaning separated.

The solution is not revolutionary. It combines known ideas: trees from compilers, graphs from databases, layering from software architecture.

But the combination seems right for the problem.

Just like a mechanical clock does not invent new physics. It just combines gears and springs and escapements in particular ways that makes it tell time accurately.

MIGR combines trees and graphs and layers in ways that let you understand your notes.

This is not trivial. Most systems do not even try.

The future work is refinement. Making it persistent. Making it incremental. Making it extensible. Making it fast.

But the foundation is sound. The vision is clear.

Epilogue: On Incremental Growth

Note that the current architecture is "what it is" and we must incrementally make it better and better as time passes.

I think this way of development is wise. Not from laziness, but from humility.

As I do not know yet what refinements will be necessary. I do not know yet what scale this will reach. I do not know yet what use cases will emerge, what I do know is that, what I have to do next, and while I do that I will know what I have to do after that and so on...

Build. Learn. Refactor.

This is how good systems grow, right?

The alternative is to over-design from the start. Anticipate and predict every problem. Build infrastructure for scales you may never reach. This leads to complexity without payoff.

My approach I believe is better: solve the problem at hand. Make the code readable. Accept that tomorrow I will know more than today.

For me it is the only honest way to build.

Thank you for reading.