1. Architectural Overview: From Source Text to Semantic Graph

The construction of the Zero-Copy Cognitive Graph is a sophisticated pipeline designed to translate raw source code into a rich, multi-layered, and performance-optimized binary representation. This process is orchestrated by the Minotaur parser and a series of subsequent enrichment passes. The core principle is to defer semantic decisions and disambiguation, first capturing all syntactic possibilities and then layering semantic understanding on top.
The pipeline consists of three primary stages:

1. **Syntactic Parsing to a Shared Packed Parse Forest (SPPF):** The Minotaur GLR parser consumes the source text and, guided by the .grammar file, produces a temporary, in-memory object graph. This graph is an SPPF, which efficiently represents every possible valid parse tree for the input, crucial for handling ambiguity.[1] During this stage, metadata from the grammar, such as projections and semantic trigger identifiers, are attached to the corresponding nodes.
2. **Serialization to the Zero-Copy Buffer:** A dedicated GraphBuilder component traverses the in-memory SPPF object graph and serializes it into the final, zero-copy Memory<byte> buffer. This process uses a post-order traversal (leaf-to-root) to resolve memory offsets, ensuring the final structure is pointer-free and instantly usable without deserialization.[4]
3. **Semantic Enrichment (CPG Overlay):** After the syntactic skeleton is serialized, dedicated analysis passes enrich the graph with semantic edges, transforming it into a full Code Property Graph (CPG). This includes adding control flow, data flow, and symbol resolution edges. This step leverages whole-project context to build a complete semantic picture.

This phased approach cleanly separates the concerns of syntactic parsing, high-performance data layout, and deep semantic analysis, resulting in a robust and efficient construction process.

2. Phase 1: Parsing to an In-Memory Shared Packed Parse Forest (SPPF)

The Minotaur parser's first responsibility is to translate the source text into a structure that faithfully represents all syntactic possibilities defined by the .grammar file. Given that the grammar can be context-sensitive and ambiguous, a simple Abstract Syntax Tree (AST) is insufficient as it represents only a single parse.[6] Therefore, the parser builds an SPPF.

2.1. The SPPF Object Model

The SPPF is constructed in memory using a temporary set of standard C# classes. This object-oriented representation is optimized for ease of construction during the dynamic parsing process, before being serialized into the high-performance zero-copy format.

- **TempSymbolNode:** Represents a grammar symbol (terminal or non-terminal) spanning a specific region of the source text. It contains:
    - SymbolID: An integer identifier mapping to the symbol in the .grammar file.
    - SourceStart, SourceEnd: The start and end character offsets in the input.
    - PackedNodes: A list of TempPackedNode children. If this list contains more than one child, the node is ambiguous.[3]
    - ProjectionID, TriggerID: Metadata identifying any associated projection or semantic trigger from the grammar.
- **TempPackedNode:** Represents a single derivation for a parent TempSymbolNode, corresponding to one specific grammar rule application. It contains:
    - RuleID: An integer identifier for the grammar rule used.
    - Children: A list of TempSymbolNode children that form the right-hand side of the rule.

This structure allows the GLR parser to efficiently represent a potentially exponential number of parse trees in a compact, polynomial-sized graph.[1]

2.2. Parser-Grammar Interaction

As the Minotaur GLR parser consumes tokens and reduces rules defined in the .grammar file, it performs the following actions:

1. **Node Creation:** When a grammar rule is successfully applied to a segment of the input, the parser creates a TempPackedNode for that rule application and a parent TempSymbolNode for the rule's non-terminal.
2. **Metadata Association:** The parser inspects the grammar rule for any associated projection or semantic trigger annotations. The unique identifiers for these constructs are stored in the ProjectionID and TriggerID fields of the newly created TempSymbolNode.
3. **Handling Ambiguity:** If the parser finds that a different rule can also produce the same non-terminal over the exact same text span, it does not create a new TempSymbolNode. Instead, it creates a new TempPackedNode for the alternative derivation and adds it to the PackedNodes list of the existing TempSymbolNode.[3] This is the core mechanism for capturing ambiguity without data duplication.

At the end of this phase, the entire source file is represented as a single, root TempSymbolNode which is the entry point to the complete parse forest.

3. Phase 2: Serialization into the Zero-Copy Buffer

This phase converts the flexible, heap-allocated SPPF object model into the rigid, high-performance zero-copy CognitiveGraph buffer. This is a pure serialization step, focused on creating the optimal binary layout defined by the schema.

3.1. The GraphBuilder Component

A dedicated GraphBuilder class manages the serialization. It uses a resizable memory buffer (backed by System.Buffers.ArrayPool<T> to minimize allocations) and follows a post-order traversal strategy inspired by serializers like FlatBuffers.[5]

3.2. The Serialization Process

The GraphBuilder recursively traverses the TempSymbolNode graph from the root, writing data from the leaves up to ensure all offsets are known before they are referenced.

1. **String and Property Serialization:** The builder first iterates through all nodes to collect unique property strings (like variable names) and writes them to a dedicated string table area in the buffer. This deduplication reduces the final buffer size.
2. **Recursive Node Writing:** The builder's main function is a recursive WriteNode method:
   - It is first called on the root TempSymbolNode.
   - The method recursively calls itself on all of the node's children (via its packed nodes).
   - Once all children have been written to the buffer and their final offsets are known, the method writes the current node's data to the buffer. This includes its SymbolID, source span, and the list of offsets pointing to its child PackedNode structures.
   - The method returns the offset of the newly written node.
3. **Packed Node and CPG Edge Serialization:** The process for TempPackedNode is similar. It writes its RuleID and the list of offsets to its child SymbolNodes. Crucially, this is also where the list of semantic CpgEdges associated with a specific parse is written (see Phase 3).
4. **Header Finalization:** After the root node is written, its final offset is known. The GraphBuilder then writes the GraphHeader at the beginning of the buffer, filling in the RootNodeOffset and other metadata like total node counts and the location of the source text copy.

The result of this phase is a single, contiguous byte or Memory<byte> that contains the complete, self-contained, and pointer-free CognitiveGraph.

4. Phase 3: Semantic Enrichment with CPG Edges

The graph produced by Phase 2 is syntactically complete but semantically sparse. The final phase adds the rich semantic edges that define a Code Property Graph, enabling deep, whole-project analysis.

4.1. The Semantic Analysis Pass

This pass operates on the fully constructed, whole-project CognitiveGraph. Because Minotaur's semantic triggers require whole-project context, this analysis is performed after all files have been parsed and their individual graphs have been linked.
For C# code, this pass can leverage the Roslyn API for its powerful semantic analysis capabilities.[7]

1. **Control Flow Graph (CFG) Edges:**
   - The analyzer identifies all SymbolNodes representing method bodies.
   - For each method, it uses the Roslyn SemanticModel to request a ControlFlowGraph.[8] This API requires enabling the "flow-analysis" feature when creating the Roslyn compilation.[8]
   - The analyzer iterates through the BasicBlocks of the Roslyn CFG. For each block, it finds the corresponding SymbolNode(s) in the CognitiveGraph and creates CONTROL_FLOW edges between them, mirroring the predecessor/successor relationships from the Roslyn CFG.
2. **Data Flow Graph (DFG) Edges:**
   - To establish data dependencies, the analyzer uses the SemanticModel.AnalyzeDataFlow method. This method can be applied to any statement or expression to determine which variables are read (ReadInside) or written (WrittenInside).
   - The analyzer traverses the graph, and for each variable usage, it performs a data flow analysis to find its definition site(s). It then creates a DATA_FLOW edge from the definition node to the usage node.

4.2. Updating the Graph

Adding these semantic edges requires modifying the CognitiveGraph buffer. To maintain the integrity and performance benefits of the zero-copy structure, this is done using a **copy-on-write** approach.

- The semantic analyzer creates a new, larger buffer.
- It performs a traversal of the original graph, block-copying the existing syntactic data

into the new buffer.
- As it calculates new semantic edges, it writes these CpgEdge structures into the new buffer and updates the CpgEdgesOffset fields in the relevant PackedNode structures to point to these new lists.

This process results in a final, semantically rich CognitiveGraph that contains a complete syntactic and semantic representation of the source code, ready for high-performance analysis by Project Golem.

Works cited

1. Part VI : Parse Forests - dei.unipd, accessed September 12, 2025, http://www.dei.unipd.it/~satta/tutorial/esslli13/part_6.pdf
2. An Efficient Context-free Parsing Algorithm For Natural Languages 1 Masaru Tomita Computer Science Department Carnegie-Mellon Un - IJCAI, accessed September 12, 2025, https://www.ijcai.org/Proceedings/85-2/Papers/014.pdf
3. Shared Packed Parse Forest (SPPF), accessed September 12, 2025, https://lark-parser.readthedocs.io/en/latest/_static/sppf/sppf.html
4. Cap'n Proto: Introduction, accessed September 12, 2025, https://capnproto.org/
5. Tutorial - FlatBuffers Docs, accessed September 4, 2025, https://flatbuffers.dev/tutorial/
6. Abstract syntax tree - Wikipedia, accessed September 5, 2025, https://en.wikipedia.org/wiki/Abstract_syntax_tree
7. Introduction to Roslyn and its use in program development - PVS-Studio, accessed September 4, 2025, https://pvs-studio.com/en/blog/posts/csharp/0399/
8. Create control flow graph for c# code using the .Net compiler Roslyn - Stack Overflow, accessed September 12, 2025, https://stackoverflow.com/questions/54466794/create-control-flow-graph-for-c-sharp-code-using-the-net-compiler-roslyn
9. ControlFlowGraph Class (Microsoft.CodeAnalysis.FlowAnalysis), accessed September 12, 2025, https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.flowanalysis.controlflowgraph?view=roslyn-dotnet-4.13.0