

Deterministic ACE Parsing to Knowledge Graph Toolchain

Completing the ACE-to-Knowledge Graph Toolchain

To build a **deterministic pipeline from ACE text to a traversable knowledge graph**, you can leverage the existing Attempto Controlled English (ACE) parser and then convert its output into a graph structure for use with GraphRAG. The general steps are:

- 1. Parse ACE Sentences with APE (ACE Parsing Engine):** Use the Attempto Parsing Engine to analyze your ACE text. APE deterministically translates ACE input into a formal representation (a **Discourse Representation Structure**, or DRS) without ambiguity ¹. The APE can directly output this representation in several formats, including standard first-order logic or OWL (Web Ontology Language) ontologies ². For example, APE supports outputs like `fo1` (first-order logic), `drsxm1` (DRS in XML), or `owlxm1` (an OWL/SWRL ontology in XML) ³. You can run APE via its command-line or web service API to get the desired output format.
- 2. Convert the Parser Output to a Graph Model:** Once you have the formal output, the next step is to **map it into a knowledge graph structure**. There are two common approaches:
 - 3. Using OWL/RDF:** If you configured APE to output an OWL ontology (for example, in RDF/XML or OWL Functional syntax), you essentially have a set of triples representing classes, individuals, and relationships. These can be loaded into an RDF-based triple store or converted for a property graph. For instance, a sentence like *"Every woman is a human."* would translate to an OWL class inclusion ($\text{Woman} \subseteq \text{Human}$) ⁴, and *"Socrates is a man."* would become an individual assertion ($\text{Socrates} \in \text{Man}$). You could load such triples into a graph database or use a tool (e.g. Neo4j's neosemantics plugin) to import RDF data into a property graph. This approach takes advantage of ACE's built-in OWL translation ¹ to produce a ready-made knowledge graph format.
 - 4. Using DRS/FOL:** Alternatively, you can parse the DRS or first-order logic output yourself and programmatically construct the graph. The ACE DRS encodes entities and predicates in a Prolog-like notation ⁵. For example, after parsing *"Some frogs are green. Kermit is a frog."*, the DRS might contain conditions like an object of type `frog` and a property `green`, plus an entity named "Kermit" of type `frog` ⁵. You could interpret each **entity** as a node in the graph (e.g. a node for "Kermit" and perhaps a generic node for "a frog" instance) and each **predicate/relation** as an edge (e.g. an edge "is-a" from Kermit to Frog, or a property edge indicating "hasColor: green"). This custom conversion gives you flexibility to shape the property graph exactly as you want (for example, using node properties for attributes like colors, or creating separate nodes for classes vs. instances).
- 5. Choose a Graph Storage Solution:** With triples or node-edge data in hand, decide on a graph database or structure to store and query the knowledge:

6. **Property Graph DB (e.g. Neo4j):** This is a modern and developer-friendly choice for knowledge graphs. Neo4j stores nodes and relationships with arbitrary properties, and you can query it using Cypher. Your converted ACE facts can be ingested as nodes (for entities/concepts) and relationships (for ACE relations). For example, you might create nodes like `Person("Socrates")` and `Category("Man")`, then a relationship `(:Person {name:"Socrates"})-[:IS_A]->(:Category {label:"Man"})`. Neo4j and similar property graph databases excel at graph traversal queries and have good integration with applications. Using a property graph can make it straightforward to implement the GraphRAG retrieval step – for instance, finding the neighbors of a node representing an entity in question.
7. **RDF Triple Store:** If you stick with the OWL/RDF output, an RDF triplestore (like GraphDB, Apache Jena/Fuseki, or Blazegraph) might be convenient. These stores use SPARQL for queries. The advantage is that your ACE-to-OWL translation drops directly into an RDF model ¹, preserving formal semantics (useful if you plan to do logical reasoning on the data). However, SPARQL queries can be verbose, and setting up a triplestore can be heavier than a property graph for certain applications.
8. **In-Memory or Custom Graph (e.g. in Rust):** Since you mentioned possibly building your own solution in Rust, you could use an in-memory graph representation. Rust has libraries like **petgraph** or **indradb** that can manage graph data. You could write a converter that takes the APE output and populates your Rust data structures (nodes and edges with types/properties). This gives you full control and potentially high performance with zero overhead from external databases. The trade-off is that you'll need to implement query logic for traversing or searching the graph. If your knowledge base isn't enormous and you prefer not to introduce an external DB, this approach can work well – just ensure you design clear APIs to retrieve subgraphs or perform graph queries (e.g. find all facts related to a given entity) for your LLM to use.
9. **Integrate with GraphRAG for LLM Queries:** With the knowledge graph constructed, the final part of the toolchain is querying it to assist the LLM. **GraphRAG** (Graph-based Retrieval-Augmented Generation) typically uses a knowledge graph to supply relevant context to the language model. In Microsoft's GraphRAG, an LLM is normally used to *build* a graph from unstructured text ⁶, and then that graph is queried for relevant nodes/relations at answer time. In your case, you've pre-built the graph from ACE, saving the heavy lifting. To use it:
10. When a user query comes in, identify key entities or terms in the query that correspond to nodes in your graph (you might do a simple lookup or use the LLM itself to pick out entities).
11. Retrieve the subgraph around those entities. For example, if the question is *"What does the Attempto parser output?"*, you'd find the node for "Attempto Parsing Engine" or "APE" and pull its relationships (e.g. *outputs -> DRS*, *outputs -> OWL*, etc., if those were encoded from ACE facts). In Neo4j, this could be a Cypher query fetching neighbors up to a certain depth; in SPARQL, a set of triple patterns; in a custom store, a graph traversal function.
12. Convert the retrieved graph information into a form the LLM can understand as context. This might be a set of natural language sentences (you can even use ACE paraphrases or just a templated English description of the triples), or a structured format that the LLM is trained to interpret. The idea is to supply the relevant facts from your knowledge graph to the LLM's prompt so it can ground its answer in those facts.

By following these steps, you *complete the toolchain* from ACE input to a knowledge graph that a LLM can traverse. Crucially, this approach offloads work from the LLM: instead of the model having to read raw text and extract knowledge (which is resource-intensive), you rely on ACE's precise semantics to build the graph. This **makes knowledge ingestion more efficient**. As Microsoft's documentation notes, GraphRAG usually relies on an LLM to generate the knowledge graph from text ⁶, but in your

setup the graph is constructed deterministically from ACE – yielding the same benefit (a structured graph) with far less runtime cost.

Should You “Modernize” the ACE Language?

ACE (Attempto Controlled English) is a well-established controlled natural language that has been around since the 1990s ⁷. It's designed to look like simplified English but with a strict formal grammar and semantics. ACE's big advantage is that any sentence you write in it has a single unambiguous interpretation, enabling automatic translation into logic or OWL formats. In practice, ACE has been successfully used for knowledge representation in areas like ontology authoring, software specifications, and rule-based reasoning ⁸. Given this, do you need to “modernize” ACE or switch to something else? Here are some considerations:

- **ACE's Language and Grammar:** The ACE language itself is stable and expressive enough for many knowledge modeling needs (it supports basic and composite sentences, quantifiers like *every/no/some*, conditionals, anaphora, etc.). Its last major version (6.7) was released in 2013 ⁷, but despite the age, it remains a unique tool – it's *already a formal language hidden in natural language clothing*. In terms of capabilities, ACE can express first-order logic statements, which is quite powerful. There isn't a fundamentally “more modern” controlled English that outclasses ACE in general usage. For example, **ClearTalk** is another controlled English variant aiming at machine-readable knowledge ⁹, but it's not clearly more capable or popular than ACE. Unless you have very domain-specific requirements that ACE's syntax cannot cover, ACE's design is still sound. In other words, the **language itself doesn't really need modernization** – it was ahead of its time in providing a natural yet formal way to encode knowledge.
- **Tooling and Integration:** Where you might consider modernization is in the **toolchain around ACE**. The official APE parser is implemented in Prolog (with a Java wrapper). This implies a dependency on older technology (SWI-Prolog) and maybe less convenient integration with modern systems. If your goal is to have a long-term, maintainable system, you have a few options:
 - Continue using APE as is, but wrap it with modern interfaces. For instance, you could run APE as a microservice (it even has an HTTP server mode ¹⁰) and have your Rust or other backend call it. The performance of APE is usually quite good for reasonable-sized texts (parsing is on the order of milliseconds to tenths of a second for typical sentences ¹¹). As long as that meets your needs, this is a stable solution. Many projects have successfully integrated ACE by treating the parser as an external service or using the provided APIs.
 - If you are feeling ambitious, **reimplement or extend the parser in Rust**. This would be a true modernization – you'd essentially port the ACE grammar and interpretation rules into a Rust parsing library. Doing so might improve runtime efficiency and remove the Prolog dependency. However, be aware this is a non-trivial project: ACE's grammar is quite rich and the interpretation (like resolving anaphora, handling plurals, etc.) is complex. The existing APE took years of development and is robust; reimplementing it means you'd need to carefully mirror all those rules. Unless you require deep integration at the parser level or need to modify ACE's syntax for your purposes, rewriting may not be worth the effort. APE is open-source (LGPL) ¹², so an alternative strategy could be to *modernize incrementally* – for example, by writing glue code in Rust to invoke the Prolog engine or by using FFI if a Prolog library can be embedded. This way, you benefit from ACE's determinism now, and you always have the option to replace the parser later if it becomes a bottleneck or if you decide to evolve the language.

- **Extending or Evolving the Language:** If by “modernize” you mean **adding new linguistic features or syntax to ACE**, think carefully. ACE’s defined subset of English is intentionally restrictive to keep it unambiguous. For future resilience, you actually want to **preserve that determinism and clarity**. Extending the language could introduce ambiguity or complexity. Instead of changing ACE itself, a safer path is to define patterns or a style guide for your knowledge authors to follow within ACE’s limits. You can introduce new *vocabulary* (domain terms) via ACE’s *user lexicon* feature easily, which is a form of modernization (keeping the lexicon up-to-date with your domain). But you probably should not loosen ACE’s grammar rules – that could undermine the very benefit of using ACE. ACE is already “future-proof” in the sense that it’s a formal specification language; any sentence can be converted to logic and thus is compatible with future reasoners or graph systems. In fact, translating ACE into OWL or other standards means your knowledge can interoperate with modern Semantic Web tools ¹.
- **Community and Maintenance:** One downside of ACE being older is that it’s not a huge active community project today. However, the flip side is that ACE is a mature, well-documented system rather than a moving target. Your base is *sane and well-tested*. To prepare for future challenges, concentrate on building your knowledge graph in standard formats (OWL/RDF or a clear node-edge model) so that if needed, you could feed the data into any other system later. The controlled language front-end (ACE) can be one of multiple inputs – for instance, in the future if you needed to ingest unconstrained natural language, you might use an LLM to parse that into your knowledge graph as well. But having ACE in place now gives you a high-precision, unambiguous knowledge source to start with. It’s a strong foundation, so long as the people authoring the knowledge are comfortable writing in ACE or using an authoring tool for it.

Bottom line: You likely do **not need to fundamentally modernize the ACE language** itself. ACE remains a powerful choice for knowledge representation due to its clarity and determinism, even if it’s a bit “old school.” Instead, focus on modernizing how you use it – for example, modern storage (property graphs or RDF as discussed), modern integration (wrappers in Rust, web services, etc.), and perhaps building user-friendly interfaces or editors for ACE. By doing so, you create a resilient pipeline: human-readable yet formal knowledge in ACE gets parsed by a deterministic engine and lands in a queryable graph. This approach positions you well for the future. You’ll have a knowledge graph grounded in logic (making it easier to maintain consistency and even perform reasoning), and you’ll avoid the heavy costs of letting an LLM hallucinate or misinterpret knowledge. In summary, **ACE gives you a sane base to build on** – stick with it unless you have a clear reason to invent a new controlled language, and complete the toolchain by bridging ACE’s parser output into the graph technology that best suits your needs. This balanced strategy will make your system robust to future challenges while leveraging the best of both worlds (precise symbolic knowledge and powerful LLMs).

Sources:

- Fuchs et al., *Attempto Controlled English Meets the Challenges of Knowledge Representation...* – ACE is a controlled natural language used as a formal knowledge representation and query language ¹³. The ACE Parsing Engine translates ACE sentences into formal logic (DRS and even OWL) unambiguously ¹.
- *Attempto Parsing Engine Documentation* – Details on using APE and its output formats (e.g., OWL functional syntax, RDF/XML, first-order logic) ³. The APE web service/API can return parses in machine-readable forms for integration.
- Larson & Truitt (Microsoft Research Blog 2024), *GraphRAG: Unlocking LLM discovery on private data* – Describes how GraphRAG usually relies on an LLM to build a knowledge graph and then uses that graph for retrieval at query time ⁶. By using ACE, you pre-build the knowledge graph deterministically, saving resources while achieving a similar goal.

- Wikipedia – *Attempto Controlled English* overview (retrieved Oct 2024) ⁷ ⁸ and “See also” mentioning **ClearTalk** as another machine-readable English variant ⁹. This context confirms ACE’s longevity and applications, helping to judge if changes are needed.

¹ ² ⁴ ⁷ ⁸ ⁹ ¹³ Attempto Controlled English - Wikipedia

https://en.wikipedia.org/wiki/Attempto_Controlled_English

³ ⁵ ¹⁰ ¹¹ APE Webservice

https://attempto.ifi.uzh.ch/site/docs/ape_webservice.html

⁶ GraphRAG: Unlocking LLM discovery on narrative private data - Microsoft Research

<https://www.microsoft.com/en-us/research/blog/graphrag-unlocking-llm-discovery-on-narrative-private-data/>

¹² GitHub - Attempto/APE: Parser for Attempto Controlled English (ACE)

<https://github.com/Attempto/APE>