Ciência de Dados e I.A.
Escola de Matemática Aplicada
Fundação Getúlio Vargas

Engenharia de Requisitos

Implementação da AST

LLM para Engenharia de Requisitos

Aluno: Isabela Yabe
Orientador: Rafael de Pinho André
Escola de Matemática Aplicada, FGV/EMAp
Rio de Janeiro - RJ.

Rio de Janeiro, 2025

Sumário

1 Implementação da Ferramenta de Análise Estática

1.1 Visão geral da solução

Para viabilizar o uso do núcleo de análise estática em diferentes experimentos, foi desenvolvida uma API que encapsula todo o fluxo de construção da AST do Python, aplicação dos *passes* e exportação dos nós enriquecidos em um formato serializável. Essa API expõe uma função de alto nível, `run_ast_analysis`, que recebe um diretório ou arquivo como entrada, executa a análise e retorna uma estrutura `AnalysisResult` contendo, para cada arquivo, o contexto de análise e a lista de `TNode`s gerados.

A API é construída sobre duas abstrações centrais: o contexto de análise (`Ctx`) e o nó sintático enriquecido (`TNode`). O contexto mantém informações globais do arquivo em processamento (pilhas de classes e funções, linhas de código, comentários por linha, caminhos de arquivo e um espaço de *scratch* compartilhado entre passes). Já o `TNode` encapsula o nó original da AST do Python (`ast.AST`) e agrega campos derivados por diferentes passes, como informações de caminho (*path info*), nomes e visibilidade, convenções de nomenclatura, tipo de método, assinatura de entrada/saída, tipo de classe, comentários e docstrings, além de métricas de grafo de chamadas (fan-in, fan-out, chamadores e chamados).

Com essa API, a construção de uma AST enriquecida deixa de ser um procedimento ad hoc e passa a ser uma operação reprodutível e configurável: basta apontar a função de alto nível para o repositório de código alvo e selecionar a estratégia de travessia e o conjunto de plugins desejados. Isso facilita a integração da análise sintática com os experimentos de extração de casos de uso, geração de embeddings e demais tarefas de engenharia de software apoiadas por aprendizado de máquina.

1.2 Organização dos módulos da ferramenta

A implementação foi estruturada como um pequeno *microkernel* de análise estática, separando a infraestrutura genérica da lógica específica de cada *pass*. O núcleo está concentrado no pacote `astcore`, enquanto os passes concretos são fornecidos por um pacote de plugins (`pass_plugins`), que pode ser estendido sem modificar o núcleo.

O pacote `astcore` agrupa os componentes centrais da arquitetura:

- **`astcore.model`**: contém as definições de `Ctx` e `TNode`, que formam o modelo intermediário, além de tipos auxiliares utilizados para representar o resultado da análise.
- **`astcore.pass_registry`**: implementa o mecanismo de registro e orquestração dos passes. Esse módulo define a estrutura `PassSpec`, o registrador central (`PassRegistry`) e o decorador `@register_pass`, utilizado pelos plugins para declarar nome, fase, ordem, dependências e tipos de nó atendidos por cada pass.

- **astcore.phase**: define o enumerado `Phase`, que organiza a execução dos passes em fases lógicas (`PRE`, `ENRICH`, `POST`). Essa enumeração é usada pelo registrador de passes para controlar a ordem global de execução.

- **astcore.traversal**: define as estratégias de travessia da AST, como pré-ordem recursiva, pós-ordem, versões iterativas e busca em largura. Cada estratégia implementa uma interface comum, permitindo escolher o padrão de visita mais adequado sem alterar os passes.

- **astcore.strategy_factory**: fornece uma *factory* que instancia a estratégia de travessia a partir de um identificador simbólico (por exemplo, `"preorder"` ou `"bfs"`). Isso desacopla o código cliente da implementação concreta de cada estratégia.

- **astcore.walker**: implementa o *walker* genérico (`walk_module`) responsável por percorrer a AST, criar um `TNode` para cada nó visitado e invocar, em cada fase, os passes registrados e aplicáveis àquele tipo sintático. Esse módulo conecta, na prática, a travessia, o modelo intermediário e o registrador de passes.

Sobre esse núcleo, a camada de plugins é organizada no pacote `pass_plugins`. O módulo `pass_plugins.loader` implementa o carregamento dinâmico de plugins: dada uma lista de nomes de módulos, ele realiza as *imports* correspondentes e invoca, em cada um, uma função de inicialização responsável por registrar os passes junto ao `PassRegistry`. Isso permite estender a ferramenta apenas adicionando novos módulos de passes, sem alterar o código do núcleo.

O conjunto de passes padrão é fornecido pelo subpacote `pass_plugins.builtin`, que agrupa os arquivos responsáveis por preencher os diferentes blocos do `TNode`. Entre eles, destacam-se:

- **path_info**: preenche os campos de localização do nó no projeto (`file_path`, `rel_path`, `dir_path`, `package`, `module`, `depth`, `ext`).

- **names_visibility**: identifica se o nó representa classes ou métodos, extrai o nome simples e o nome qualificado. Os campos preenchidos incluem `is_class`, `is_method`, `name`, `qname`, `decorators` e `visibility`, com base em convenções de nomenclatura (prefixos `_` e `__`).

- **naming**: decompõe identificadores em *tokens* léxicos e classifica o estilo de nomenclatura, preenchendo o bloco *naming* do `TNode`.

- **method_kind**: classifica métodos em instância, estáticos, de classe ou propriedades, com base em decoradores e no contexto de classe.

- **io_signature**: extrai a assinatura de entrada e saída de funções e métodos, incluindo parâmetros, anotação de retorno, uso de geradores, exceções levantadas e expressões de retorno.

- **class_kind**: analisa bases, metaclasse e decoradores de classes para identificar @dataclass, @final, enums e para classificá-las em *concrete*, *abstract* ou *protocol*.
- **docs_comments**: integra informações da AST com o índice de comentários mantido no Ctx, preenchendo docstrings, blocos de comentários de cabeçalho e comentários internos, além de alimentar o mapa global docstrings_by_qname.
- **core_nodes**: marca, por meio dos campos is_core e core_kind, os nós considerados centrais na representação reduzida da AST (escopos, atribuições, chamadas, retornos e lançamentos de exceção).
- **call_graph**: constrói o grafo de chamadas entre funções e métodos em duas etapas: primeiro coleta as chamadas locais e acumula arestas e nós chamáveis no scratch; depois, em uma fase posterior, resolve chamadores e chamados e preenche as métricas de *fan-in* e *fan-out* de cada TNode.

Essa organização modular separa claramente as responsabilidades entre o núcleo genérico de análise e os passes especializados. Novas dimensões de enriquecimento podem ser adicionadas simplesmente criando módulos adicionais em pass_plugins e registrando seus passes, preservando a arquitetura do microkernel e a reprodutibilidade dos experimentos.

1.2.1 Núcleo astcore

O núcleo astcore implementa a infraestrutura central do microkernel de análise estática. Sua função é fornecer os componentes genéricos e reutilizáveis responsáveis por modelar o contexto de análise e os nós enriquecidos (Ctx e TNode - astcore.model), organizar a execução dos passes por meio das fases (Phase - astcore.phase), registrar e ordenar passes com base em dependências declarativas (PassRegistry - astcore.pass_registry), definir múltiplas estratégias de travessia da AST (traversal - astcore.traversal) e expor uma fábrica para seleção dessas estratégias (strategy_factory - astcore.strategy_factory). Esses elementos são integrados pelo módulo walker, responsável por percorrer a AST, criar instâncias de TNode e aplicar, em cada fase, os passes registrados (astcore.walker).

astcore.model.

astcore.phase.

astcore.pass_registry. O módulo astcore.pass_registry implementa o microkernel de orquestração dos *passes* de análise. Cada *pass* é descrito pela estrutura PassSpec, que associa a função de processamento (fn) a um conjunto de metadados: index ordenado (sort_index), nome (name), fase de execução (phase), prioridade relativa dentro da fase (order), dependências declaradas (requires), tipos de nó da AST aos quais se aplica (node_types) e a lista de campos do TNode que ele garante preencher (provides). Um predicado opcional (when) permite restringir dinamicamente os nós para os quais o *pass* deve ser executado.

Além de armazenar metadados, o `PassSpec` executa um conjunto de validações estruturais diretamente no método `__post_init__`, o que impede que o pipeline seja inicializado com configurações inválidas.

A estrutura `PassRegistry` mantém o catálogo global de *passes*, organizado por fase de execução. Internamente, o registrador mantém (i) um índice por nome, que permite consultar rapidamente um `PassSpec`, e (ii) uma lista por fase (`PRE`, `ENRICH`, `POST`) com todos os *passes* declarados para aquela etapa. Quando um novo *pass* é registrado, o registrador verifica se o nome já foi utilizado, adiciona o `PassSpec` na fase apropriada e mantém a lista ordenada pelo par (`order`, `name`), garantindo uma ordenação.

Para compor a ordem final de execução, o `PassRegistry` oferece um método de ordenção topológica (`topological`) que recebe o conjunto de `PassSpecs` de uma fase, valida se todas as dependências (`requires`) estão presentes e, em seguida, constrói um grafo de precedência entre *passes*. A ordenação é calculada por uma variante do algoritmo de Kahn, combinada com uma fila de prioridade: a cada passo, são selecionados apenas *passes* cujas dependências já foram satisfeitas, priorizando aqueles de menor `order` e, em caso de empate, de menor `name`. Se, ao final do processo, ainda existirem *passes* com dependências não resolvidas, o registrador sinaliza um ciclo de dependência entre *passes*, impedindo a execução do pipeline em um estado inconsistente.

Por fim, o módulo expõe um decorador de conveniência, `@register_pass`, utilizado pelos plugins para declarar novos *passes*. Esse decorador recebe, como argumentos, os metadados do *pass* (nome, fase, ordem, dependências, tipos de nó e campos `provides`) e registra automaticamente a função alvo no `PassRegistry` global. Dessa forma, basta importar o módulo de plugins para que seus *passes* sejam visíveis ao núcleo de análise, sem necessidade de código adicional de configuração.

`astcore.traversal`.

`astcore.strategy_factory`.

`astcore.walker`.

1.2.2 Camada de plugins `pass_plugins`

`pass_plugins.loader`.

Passes padrão em `pass_plugins.builtin`.

1.3 Modelo de contexto e nós enriquecidos

1.3.1 Contexto de análise `Ctx`

1.3.2 Nó sintático enriquecido `TNode`

Campos estruturais básicos.

Bloco *path_info*.

Bloco *names_visibility*.

Bloco *naming*.

Bloco *method_kind*.

Bloco *io_signature*.

Bloco *class_kind*.

Bloco *docs_comments*.

Bloco de grafo de chamadas.

1.4 Execução dos passes e pipeline de análise

1.4.1 Fases de execução e registro de passes

Cada *pass* é implementado como uma função com assinatura padronizada, recebendo o `TNode` corrente, o nó original da AST (`ast.AST`) e o contexto de análise (`Ctx`) e enriquecendo apenas o `TNode` correspondente. O decorador `@register_pass` permite que essa função seja registrada declarativamente junto ao `PassRegistry`, informando: (i) o nome do *pass* (`name`); (ii) a fase em que deve ser executado (`phase`); (iii) uma prioridade relativa (`order`); (iv) as dependências de outros *passes* (`requires`); (v) os tipos de nó aos quais se aplica (`node_types`); e (vi) os campos do `TNode` que serão preenchidos (`provides`).

Durante a inicialização da ferramenta, os módulos de plugins são carregados e seus *passes* são registrados no `PassRegistry`. Para cada fase (`PRE`, `ENRICH`, `POST`), o núcleo consulta o registrador, obtém a lista de `PassSpec`s e aplica a ordenação topológica, garantindo que todas as dependências sejam respeitadas. Assim, por exemplo, um *pass* responsável por calcular métricas de grafo de chamadas pode declarar que depende de outro *pass* que constrói as arestas do grafo; o `PassRegistry` assegura que o primeiro só será executado depois que o segundo tiver produzido os dados necessários.

Esse mecanismo de registro e ordenação por fases permite separar a lógica de enriquecimento do código de infraestrutura de execução. O código cliente precisa apenas escolher quais plugins carregar; a definição da ordem correta de execução é responsabilidade do microkernel de *passes*. Como consequência, novos *passes* podem ser introduzidos ou removidos sem alterar o núcleo, preservando a extensibilidade e a reprodutibilidade dos experimentos.

1.4.2 Estratégias de travessia e `walker`

1.5 Passes padrão de enriquecimento

`path_info`.

`names_visibility`.

`naming`.

`method_kind`.

`io_signature`.

`class_kind`.

`docs_comments`.

`core_nodes`.

`call_graph`.

1.6 API de alto nível para análise de repositórios

1.7 Resumo da arquitetura de análise estática