TermIA – Terminal Inteligente

Projeto Final – ECOI26 Compiladores

Argéu V. S. Rodrigues Henrique T. Silva

Universidade Federal de Itajubá
Campus Itabira

November 29, 2025

Roteiro

Visão Geral do Projeto

O que é o TermIA?

- Shell interativo em Python que integra:
  - conceitos de **Compiladores** (analisador léxico, sintático e AST);
  - comandos reais de **sistema operacional**;
  - comandos de **Inteligência Artificial** para apoio ao usuário.
- Desenvolvido como projeto final da disciplina ECOI26 – Compiladores.
- Foco em ser um experimento didático de construção de uma linguagem de comandos e seu interpretador de ponta a ponta.

Motivação

- **Contexto**
  - Terminais tradicionais têm comandos fixos e pouca inteligência.
  - Há pouca integração entre ferramentas de linha de comando e recursos modernos de IA.
  - Alunos precisam de prática concreta em análise léxica e sintática, e não apenas exemplos teóricos isolados.
- **Do ponto de vista da disciplina**
  - Exercitar, em um projeto único, os principais tópicos vistos em sala:
    - definição de gramática formal;
    - implementação de **lexer** e **parser** com PLY;
    - construção de **árvore sintática abstrata** (AST);
    - mapeamento da AST para ações concretas (executor).
- **Do ponto de vista de usabilidade**
  - Criar um terminal mais amigável, com:
    - mensagens de erro claras;
    - histórico e **autocomplete**;
    - comandos de IA para explicar código, resumir textos e traduzir.
- **Objetivo central**
  - Criar um **terminal customizável** que também converse com IA, aproximando conceitos de Compiladores do uso real de linha de comando.
  - Entregar um artefato executável, testado e bem documentado para uso em aula e em futuros trabalhos.

Objetivos do Projeto

Objetivos Gerais

- Implementar um **terminal inteligente** com:
  - subconjunto de comandos de sistema operacional;
  - comandos específicos de IA;
  - comandos de controle do próprio shell.
- Aplicar conceitos da disciplina na prática:
  - análise léxica e sintática;
  - desenho de gramática e AST;
  - tratamento de erros.
- Seguir um processo de engenharia de software:
  - seguir requisitos funcionais e não funcionais;
  - arquitetura modular;
  - testes automatizados (pytest);
  - documentação em Markdown contida no README.md.

Arquitetura e Tecnologias

Arquitetura Geral

Pipeline de execução de comandos

`Input (linha de comando)`

⇓

Lexer (tokens)

⇓

Parser (AST)

⇓

Executor (interpretação da AST)

⇓

`Output (resultado para o usuário)`

Componentes Principais

- **Lexer**
  - Implementado com `ply.lex`.
  - Responsável por identificar palavras-chave, flags, caminhos, strings etc.
- **Parser**
  - Implementado com `ply.yacc`.
  - Constrói a **AST** a partir da gramática dos comandos.
- **Executor**
  - Recebe a AST e executa ações concretas:
  - chama funções de sistema de arquivos ou integra com IA.
- **Camada de IA**
  - Centraliza chamadas para o provedor de IA.
  - Converte resultados em texto amigável no terminal.

Tecnologias Utilizadas

- **Linguagem e ambiente**
  - Python 3.8+ com ambiente virtual (`venv`) e gerenciamento de pacotes via `pip`.
- **Ferramentas de compiladores**
  - PLY (`ply.lex` e `ply.yacc`) para implementação do analisador léxico e sintático.
  - Gramática formal em BNF e conjunto de tokens documentados em `README.md`.
- **Linha de comando e usabilidade**
  - `prompt_toolkit` para loop interativo, histórico e autocomplete do shell.
  - `pygments` e `colorama` para realce de sintaxe e cores no terminal.
- **Integração com IA e configuração**
  - `requests` para comunicação HTTP com a API de IA.
  - `pyyaml` para leitura de arquivos de configuração (ex.: chaves de API e parâmetros).
- **Modelo de IA (fornecido pelo professor)**
  - Endpoint: `https://api.ninja-apps.work/v1/chat/completions`
  - Modelo: `gpt-oss:20b` *(com guardrails)*
- **Testes e qualidade**
  - `pytest` para testes automatizados de lexer, parser, executor e comandos de IA.
- **Controle de versão e documentação**
  - Git + GitHub (`github.com/enriqTS/TermIA`) para versionamento do código.
  - Documentação em `README.md` e `gramatica.md`.

Funcionalidades Implementadas

Comandos de Sistema Operacional

Subconjunto suportado

- **`ls [opções] [caminho]`** – lista o conteúdo de um diretório
  - Opções suportadas: `-a` (arquivos ocultos), `-l` (formato longo), `-h` (tamanhos legíveis).
  - Permite combinações como `-al`, `-lh`, `-alh`.
- **`cd [caminho]`** – muda o diretório de trabalho atual
  - Suporta caminhos absolutos, relativos, `..` e atalho `~`.
  - Sem argumentos, retorna para o diretório “home” do usuário.
- **`mkdir [-p] <nome_diretorio>`** – cria novos diretórios
  - `-p`: cria diretórios pais automaticamente, se necessário.
- **`pwd`** – mostra o diretório de trabalho atual
  - Exibe o caminho completo do diretório atual.
- **`cat <arquivo>`** – exibe o conteúdo de arquivos texto
  - Leitura simples de arquivos, útil para inspecionar scripts, logs e textos.

Comandos de Inteligência Artificial

Interface de IA

- Todos os comandos de IA utilizam o prefixo **`ia`** seguido do subcomando.

- **`ia ask "<pergunta>"`**
  - Faz perguntas gerais em linguagem natural à IA.
  - Retorna respostas explicativas ou descritivas sobre o tema solicitado.
- **`ia summarize "<texto>" [-length short|medium|long]`**
  - Gera um resumo do texto fornecido.
  - `-length short`: resumo curto (padrão).
  - `-length medium`: resumo intermediário.
  - `-length long`: resumo mais detalhado.
- **`ia codeexplain <arquivo>`**
  - Lê o conteúdo do arquivo e produz uma explicação em linguagem natural.
  - Útil para entender trechos de código (ex.: `main.py`, `lexer.c`).
- **`ia translate "<texto>" -to <idioma>`**
  - Traduz o texto para o idioma especificado pela opção `-to`.
  - Idiomas suportados (documentados): `pt`, `en`, `es`, `fr`, `de`, `it`.

Comandos de Controle e Usabilidade

- **`history [n]`**: exibe histórico recente de comandos.
- **`help [cmd]`**: ajuda contextual para cada comando.
- **`clear`**: limpa a tela.
- **`exit`**: encerra o TermIA.

Recursos adicionais

- Histórico persistente entre execuções.
- Autocomplete inteligente de comandos e caminhos.
- **Syntax highlighting** básico para melhorar leitura.
- Tratamento amigável de erros de sintaxe e execução.

Requisitos Atendidos

Requisitos funcionais

- Interpretação dos comandos de sistema operacional especificados.
- Suporte aos comandos de `ia` conforme documentação.
- Histórico, ajuda e comandos de controle do shell.

Requisitos não funcionais

- Implementação modular (lexer, parser, executor, IA).
- Scripts de teste automatizados com `pytest`.
- Mensagens de erro claras e comportamento robusto frente a entradas inválidas.

Gramática, Lexer e Parser

Gramática Formal – Comandos de SO

BNF dos comandos suportados

```
<command>         ::= <os-command>
                    | <ia-command>
                    | <control-command>

<os-command>      ::= <ls-cmd>
                    | <cd-cmd>
                    | <mkdir-cmd>
                    | <pwd-cmd>
                    | <cat-cmd>

<ls-cmd>          ::= "ls" [<ls-options>] [<path>]
<ls-options>      ::= "-" <ls-flags>
<ls-flags>        ::= "a" | "l" | "h"
                    | "al" | "ah" | "lh" | "alh"

<cd-cmd>          ::= "cd" [<path>]

<mkdir-cmd>       ::= "mkdir" ["-p"] <path>

<pwd-cmd>         ::= "pwd"

<cat-cmd>         ::= "cat" <path>
```

- Cobre todos os comandos de sistema implementados: `ls`, `cd`, `mkdir`, `pwd`, `cat`.
- As combinações válidas de flags de `ls` são explicitadas em `<ls-flags>`.

Gramática Formal – IA e Controle

BNF para comandos de IA e de controle

```
<ia-command>     ::= "ia" <ia-subcommand>
<ia-subcommand>  ::= <ia-ask> | <ia-summarize> | <ia-codeexplain> | <ia-translate>

<ia-ask>         ::= "ask" <quoted-string>

<ia-summarize>   ::= "summarize" <quoted-string> [<length-option>]
<length-option>  ::= "--length" ("short" | "medium" | "long")

<ia-codeexplain> ::= "codeexplain" <path>

<ia-translate>   ::= "translate" <quoted-string> "--to" <language>
<language>       ::= "pt" | "en" | "es" | "fr" | "de" | "it"

<control-command>::= <history-cmd>
                   | <clear-cmd>
                   | <help-cmd>
                   | <exit-cmd>

<history-cmd>    ::= "history" [<number>]
<clear-cmd>      ::= "clear"
<help-cmd>       ::= "help" [<command-name>]
<exit-cmd>       ::= "exit"

<path>           ::= <identifier> | <path> "/" <identifier> | "." | ".." | "~"

<quoted-string>  ::= '"' <string-content> '"'
<number>         ::= <digit>+
<command-name>   ::= <identifier>
```

Lexer e Tokens

- **Implementação com `ply.lex`**
  - Regras léxicas em Python que transformam o texto digitado em uma sequência de tokens.
  - Cada palavra-chave, literal e símbolo descrito na gramática possui um token correspondente.
- **Categorias de tokens (documentadas em `README.md`)**
  - **Palavras-chave** (*keywords*): `LS`, `CD`, `MKDIR`, `PWD`, `CAT`, `IA`, `ASK`, `SUMMARIZE`, `CODEEXPLAIN`, `TRANSLATE`, `HISTORY`, `CLEAR`, `HELP`, `EXIT`.
  - **Operadores e símbolos**: `DOT` (`.`), `DOTDOT` (`..`), `TILDE` (`~`).
  - **Literais**: `STRING` (texto entre aspas), `NUMBER` (sequência de dígitos), `IDENTIFIER` (nomes de arquivos, diretórios e comandos), `PATH` (caminhos de arquivos/diretórios).
  - **Opções de linha de comando**: `OPTION_SHORT` (ex.: `-a`, `-l`, `-p`, `-la`, `-lah`), `LONG_OPTION` (ex.: `-length`, `-to`).
- **Tratamento da entrada**
  - Espaços em branco e quebras de linha são tratados de forma a separar comandos sem interferir na gramática.
  - Caracteres inesperados são detectados e reportados como erros léxicos para o usuário.

Parser e AST

- **Parser com `ply.yacc` (`TermIAParser`)**
  - Classe `TermIAParser` encapsula o lexer (`TermIALexer`) e o parser do PLY.
  - Método `parse(text)` recebe a linha digitada e retorna o nó raiz da AST.
  - Regra inicial `command` delega para `os_command`, `ia_command` ou `control_command`.
  - Cada produção é implementada como um método `p_<nome_regra>` (ex. `p_ls_command_full`, `p_ia_summarize_with_length`), que instancia diretamente o nó de AST correspondente.
- **Organização da AST (`ast_nodes.py`)**
  - Classe abstrata `ASTNode` define `__repr__()` e `to_dict()` para todos os nós.
  - Três famílias principais:
    - `OSCommand`: `LSCommand`, `CDCommand`, `MkdirCommand`, `PwdCommand`, `CatCommand`;
    - `IACommand`: `IAAskCommand`, `IASummarizeCommand`, `IACodeExplainCommand`, `IATranslateCommand`;
    - `ControlCommand`: `HistoryCommand`, `ClearCommand`, `HelpCommand`, `ExitCommand`.
  - Cada nó guarda apenas os dados relevantes (ex.: `options` e `path` em `LSCommand`), deixando a AST enxuta e fácil de serializar e debugar.
- **Erros e depuração**
  - Função `p_error` trata erros sintáticos, com mensagens específicas para casos comuns (como `mkdir` ou `cat` sem argumentos obrigatórios) e recuperação via `parser.errok()`.
  - Métodos auxiliares `parse_and_print` e função `print_ast` permitem visualizar a AST em formato textual e em `dict`, apoiando o desenvolvimento e os testes.

Testes e Qualidade

Estratégia de Testes

Estrutura em `tests/`

- **`test_ai_api.py`** – teste padrão da API de IA, fornecido pelo professor.
- **`test_lexer.py`** – testes do analisador léxico (tokens e erros de entrada).
- **`test_parser.py`** – testes do analisador sintático (aplicação da gramática e construção da AST).
- **`test_executor.py`** – testes do executor dos comandos de SO (`ls`, `cd`, `mkdir`, `pwd`, `cat`).
- **`test_ia_commands.py`** – testes dos comandos de IA (`ia ask`, `ia summarize`, `ia codeexplain`, `ia translate`).
- **`test_enhanced_features.py`** – testes das **features** adicionais (histórico, autocomplete, mensagens de erro e demais melhorias).

Execução com `pytest`

- Execução do conjunto completo: `pytest tests/`.
- Execução de arquivos específicos: por exemplo, `pytest tests/test_lexer.py -v`.

Resultados dos Testes

- A suíte de testes em `tests/` (arquivos `test_ai_api.py`, `test_lexer.py`, `test_parser.py`, `test_executor.py`, `test_ia_commands.py` e `test_enhanced_features.py`) foi utilizada como **principal mecanismo de validação** do projeto.
- Em conjunto, esses testes cobrem:
  - reconhecimento léxico e tratamento de erros de entrada;
  - aplicação da gramática e construção correta da AST;
  - execução dos comandos de SO e de controle do shell;
  - comportamento dos comandos de IA (`ask`, `summarize`, `codeexplain`, `translate`);
  - funcionamento das **features** adicionais (histórico, autocomplete, mensagens de erro mais amigáveis).
- Na entrega final do projeto:
  - todos os testes automatizados passaram com sucesso;
  - a suíte foi usada de forma recorrente para detectar regressões e garantir que o código e as funcionalidades implementadas se mantivessem corretos após cada alteração.

Conclusões

Conclusões

- **Objetivos do projeto atingidos**
  - Implementação de um terminal inteligente que combina comandos de SO, comandos de IA e comandos de controle em uma mesma linguagem de comandos.
  - Cumprimento do escopo definido em PROJETO_TermIA: lexer, parser, AST, executor, módulo de IA e **features** adicionais.
  - Arquitetura modular, documentada em `README.md`, com testes automatizados em `pytest`.
- **Resultados técnicos**
  - Gramática formal clara e alinhada com a implementação real.
  - Integração consistente entre `TermIALexer`, `TermIAParser`, nós de AST e executor dos comandos.
  - Suíte de testes utilizada como base para validar e evoluir o código com segurança.
- **Aprendizados na disciplina**
  - Consolidação dos conceitos de análise léxica, análise sintática e AST em um projeto completo.
  - Vivência prática de engenharia de software: testes, documentação, controle de versão e integração com serviços externos (IA).
  - Aproximação da teoria de Compiladores com ferramentas e fluxos de trabalho usados no dia a dia de desenvolvimento.

Trabalhos Futuros

Como ideias para funcionalidades futuras, pensamos em:

- Expandir o conjunto de comandos de sistema (ex.: `rm`, `cp`, `mv`).
- Suporte a **scripts** de TermIA (arquivos com sequência de comandos).
- Experimentar outros modelos/serviços de IA e ajustes de prompt.
- Empacotar o TermIA como ferramenta instalável (`pip` / `pipx`).

Demonstração / Perguntas

Dúvidas?