CSCD01 LangChain Issue Analysis

Team: Natural Dumbness

Issue

Add support for a variety of languages to LanguageParser #11229
https://github.com/langchain-ai/langchain/issues/11229

In LangChain, `LanguageParser` is a parser for Document Loaders that, given source code, splits each top-level function or class into separate documents. As stated in its documentation:

> "This approach can potentially improve the accuracy of QA models over source code. Currently, the supported languages for code parsing are Python and JavaScript."

We would like to add support for additional languages, such as C, C++, Rust, Ruby, Perl, and so on. This improvement would allow LangChain users to conveniently load codebases written in these languages into many small "documents". As the `LanguageParser` documentation suggests, this can improve the efficiency and accuracy of LLM-based code QA.

Our implementation will be based on the tree-sitter parser framework. There are tree-sitter parsers for a wide variety of languages, but crucially, each presents a common interface, through the py-tree-sitter API (which wraps the C API). We can thus coordinate the parsing for each language using a single, common algorithm, which is parameterized for each language only where necessary.

Community discussions

In our GitHub issue thread, we discussed some high-level topics with interested community members. In particular, some members have been keen to implement code splitting for the languages they personally use.

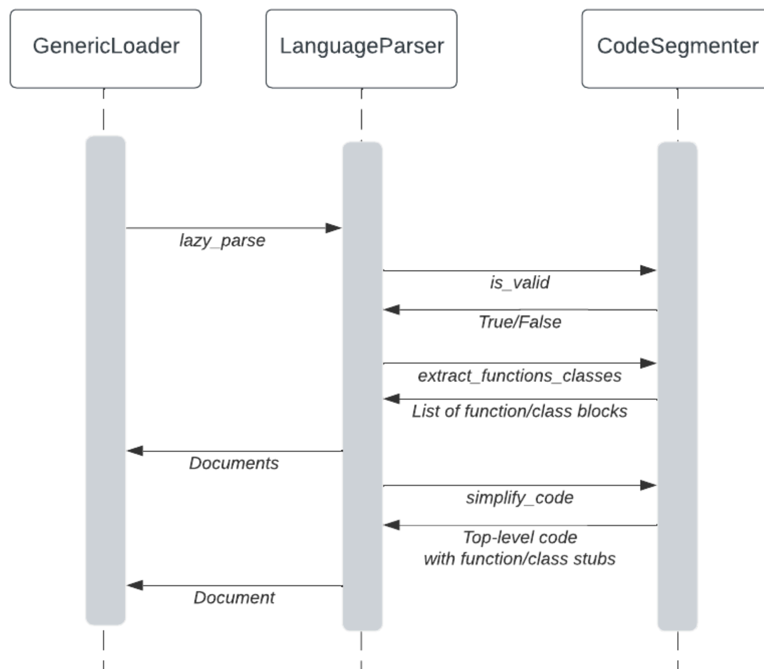
If our generic implementation based on tree-sitter is merged, we anticipate this task will be much simpler going forward. We have communicated as such, while also stressing that community members are still free to **not** use tree-sitter for their parsers, if they so choose.

Outline of changes

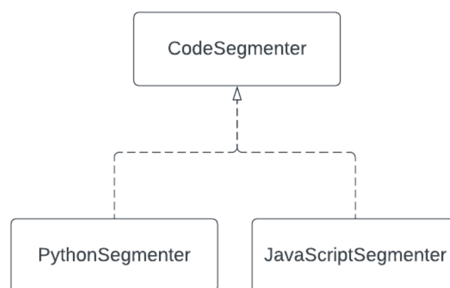
Primer on existing architecture

`LanguageParser` is an existing class that orchestrates the parsing process. To do so, it takes as input a case of the `Language` enum, or infers one from the filename extension. This `Language` is then mapped to a *segmentation* [*strategy*](), which is a class that extends `CodeSegmenter` (which is abstract). There is one segmentation strategy for each supported language. The static dict `LANGUAGE_SEGMENTERS` maps each `Language` case to a segmentation strategy.

Sequence diagram



Segmentation strategy class diagram



Files to modify

- text_splitter.py
  - `Language` enum
    - we may add new cases
- language_parser.py
  - `LANGUAGE_EXTENSIONS` and `LANGUAGE_SEGMENTERS` static dicts
    - we will register new segmentation strategies here

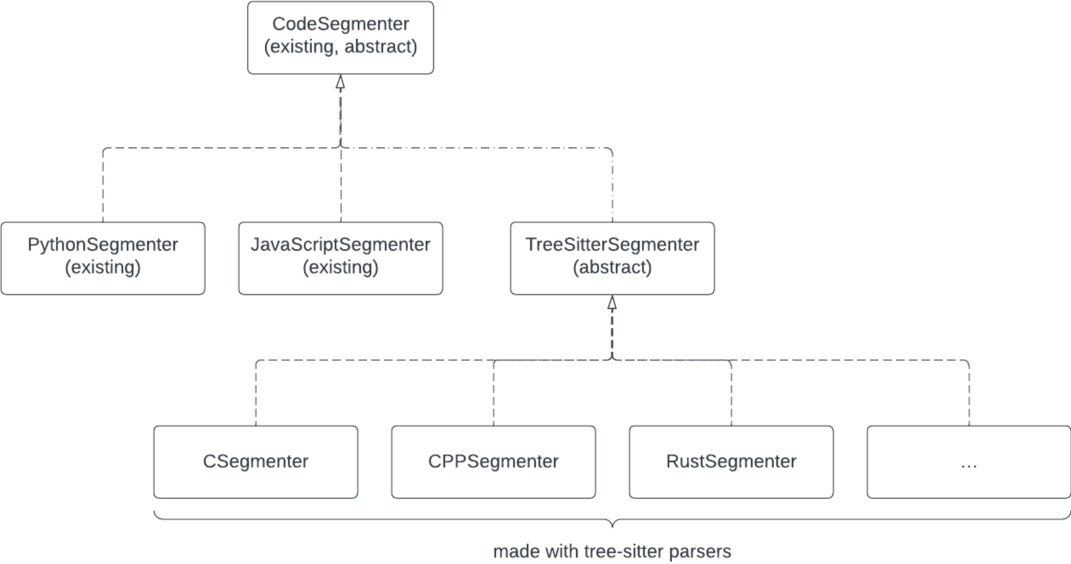
New files

- tree_sitter_segmenter.py – `TreeSitterSegmenter`
  - extends `CodeSegmenter`
  - abstract base class, using the [template method pattern]
    - segmenters that use the tree-sitter parsing library extend this class, overriding the "abstract steps" (methods) to complete the algorithm
    - abstract methods:
      - `get_language()` – returns tree-sitter `Language` object ([example])
      - `get_chunk_query()` – returns [tree-sitter AST query] for top-level functions and classes (chunks)
      - `make_line_comment()` – wraps a string in a line comment for the target language (such as `// [comment-text-here]` for C)
  - implements abstract methods from `CodeSegmenter`
    - `is_valid()` checks validity of source code (to parse)
    - `extract_functions_classes()` divides code into chunks of functions and class declarations, returns a list of these segments
    - `simplify_code()` separates any additional code not included within the previous chunks

- cpp.py, rust.py, … – `CPPSegmenter`, `RustSegmenter`, …
  - extend `TreeSitterSegmenter`
  - implement abstract methods from `TreeSitterSegmenter` (list above)

- one test file per language (test_cpp.py, test_rust.py, …)
  - `test_extract_functions_classes()`
  - `test_simplify_code()`

New class diagram for segmentation strategies



Pseudocode

Outline of `TreeSitterSegmenter`

```
def is_valid(self) -> bool:
    parser = make_parser(self.get_language())
    try:
        parser.parse(source_code)
        return True
    except:
        # exception is thrown if parser is unable to parse
        return False

def extract_functions_classes(self) -> List[str]:
    # get the language from subclass, construct parser
    parser = make_parser(self.get_language())

    # parse source code using tree-sitter parser
    ast = parser.parse()

    # keep track of lines already processed to avoid duplication of chunks
    processed_lines = set()
    chunks = [] # to return parsed chunks
    for node in ast.query(self.get_chunk_query()):
        # chunk ranges from start_line to end_line
        start_line = node.start_point
        end_line = node.end_point
        if any line in [start_line, end_line] is in processed_lines:
            continue
        processed_lines.update(lines)

        # append chunk to resulting list
        chunks.append(node.text)

    return chunks


def simplify_code(self) -> str:
    # as before: get language and chunk query, initialize parser,
    # and parse initial source code

    # as before: initialize the processed_lines set

    simplified_lines = source_code[:] # make a copy of source
    for node in parsed_code:
```

```
        start_line = node.start_point
        end_line = node.end_point

        # same as above
        if any line in [start_line, end_line] is in processed_lines:
            continue

        # replace first line of chunk with a stub comment
        # that tells the LLM what is missing
        simplified_lines[start_line] = \
            make_line_comment("Code for: {source_code[start_line]}")

        # remove all lines below the comment for current chunk
        for line_num in range(start_line + 1, end_line + 1):
            simplified_lines[line_num] = None

        # update processed lines
        processed_lines.update(lines)

return all non-empty chunks in simplified_lines
```

Sample concrete segmentation strategies

C++

```
# Query string for tree-sitter C++ AST
(https://github.com/tree-sitter/tree-sitter-cpp)
CHUNK_QUERY = """
    [
        (class_specifier
            body: (field_declaration_list)) @class
        (function_definition) @function
    ]
""".strip()

class CPPSegmenter(TreeSitterSegmenter):
    """Code segmenter for C++."""

    def get_language(self):
        from tree_sitter import Language
        return Language("tree-sitter-cpp.so", "cpp")

    def get_chunk_query(self) -> str:
        return CHUNK_QUERY

    def make_line_comment(self, text: str) -> str:
        return f"// {text}"
```

Ruby

```
# Query string for tree-sitter Ruby AST
(https://github.com/tree-sitter/tree-sitter-ruby)
CHUNK_QUERY = ...

class RubySegmenter(TreeSitterSegmenter):
    """Code segmenter for Ruby."""

    def get_language(self):
        from tree_sitter import Language
        return Language("tree-sitter-ruby.so", "ruby")

    def get_chunk_query(self) -> str:
        return CHUNK_QUERY

    def make_line_comment(self, text: str) -> str:
        return f"# {text}"
```