Chapter 2: TinyPEG Library Overview

2.1 Architecture and Design Philosophy

The TinyPEG library is designed with simplicity, flexibility, and educational value in mind. Its architecture follows these key principles:

1. **Separation of Concerns**: The library separates the core parsing mechanisms from specific grammar implementations, allowing users to define their own grammars.

2. **Object-Oriented Design**: Grammar elements are represented as objects, making the library extensible and maintainable.

3. **Composability**: Grammar rules can be composed to create complex parsers from simple building blocks.

4. **Minimal Dependencies**: The library relies only on Python's standard library, making it easy to use in any environment.

The library is organized into three main modules:

- **core.py**: Contains the fundamental classes for parsing, including Expression, Reference, ParserContext, and ParseError.
- **parsers.py**: Provides the PEGParser and Rule classes for building parsers.
- **syntax_tree.py**: Defines the GrammarNode and DebugVisitor classes for representing and traversing syntax trees.

Let's explore each of these components in detail.

2.2 Core Components

2.2.1 Expression Class

The `Expression` class is the abstract base class for all parsing expressions in the TinyPEG library. It defines the interface that all concrete expression types must implement.

`python class Expression: """Base class for expressions.""" def parse(self, ctx): raise NotImplementedError("Subclasses should implement this method.")`

The `parse` method takes a `ParserContext` object and attempts to match the expression against the input at the current position. Subclasses override this method to implement specific parsing behaviors.

2.2.2 Reference Class

The `Reference` class is a crucial component that allows grammar rules to reference other rules by name. This enables recursive grammar definitions and modular rule composition.

```python class Reference(Expression): """Reference to another rule or pattern in the grammar."""

```
def __init__(self, name):
    """
    Initialize a reference to another rule or pattern.

    Args:
        name: The name of the rule or pattern being referenced
    """
    self.name = name

def parse(self, ctx):
    # This will be implemented by the grammar system
    # that resolves references to actual rules
    raise NotImplementedError("Reference parsing is handled by the gramm
```

```

When a grammar is being parsed, references are resolved to their corresponding rules, allowing for a clean separation between rule definition and rule usage.

2.2.3 ParserContext Class

The `ParserContext` class maintains the state of the parsing process, including the input text and the current position.

```python class ParserContext: """Context class to hold the parsing state."""

```
def __init__(self, text):
    self.text = text
    self.pos = 0

def eof(self):
    return self.pos >= len(self.text)

def peek(self):
    return self.text[self.pos] if not self.eof() else None

def consume(self):
    """Advance the position by one character."""
    char = self.peek()
    self.pos += 1
    return char
```

```

This class provides methods for checking if the end of input has been reached (`eof`), looking at the current character without consuming it (`peek`), and advancing the position (`consume`).

2.2.4 ParseError Class

The `ParseError` class is a custom exception used to report parsing errors.

`python class ParseError(Exception): """Custom error for parsing issues."""  pass`

This allows the library to distinguish between parsing errors and other types of exceptions that might occur during the parsing process.

2.3 Parser Components

2.3.1 PEGParser Class

The `PEGParser` class is the base class for all parsers built with TinyPEG. It provides the infrastructure for parsing input according to a grammar.

```python class PEGParser: """Parser class that uses PEG grammar rules."""

```
def __init__(self):
    self.grammar = None # To be defined by subclasses

def parse(self, text: str):
    """Parse input text according to the grammar."""
    if self.grammar is None:
        raise ValueError("Grammar not defined")

    # Create parser context
    ctx = ParserContext(text)

    # Apply the start rule (first rule in grammar)
    if self.grammar.rules:
        result = self._parse_expression(self.grammar.rules[0].expr, ctx)
        if not ctx.eof():
            raise ParseError(f"Unexpected input at position {ctx.pos}")
        return result

    raise ParseError("No rules defined in grammar")

def _parse_expression(self, expr, ctx):
    """Parse an expression with the given context."""
    return expr.parse(ctx)
```

```

In a complete implementation, the `parse` method would: 1. Create a `ParserContext` from the input text 2. Resolve references in the grammar 3. Apply the grammar's start rule to the context 4. Return the resulting parse tree or AST

2.3.2 Rule Class

The `Rule` class represents a named rule in a grammar. It associates a name with an expression that defines what the rule matches.

```python class Rule(Expression): """Represents a grammar rule."""

```
def __init__(self, name, expr):
    self.name = name
    self.expr = expr

def parse(self, ctx):
    return self.expr.parse(ctx)
```

```

Rules are the building blocks of grammars. They can reference other rules, creating a network of dependencies that defines the grammar's structure.

2.4 Syntax Tree Components

2.4.1 GrammarNode Class

The `GrammarNode` class represents a node in the grammar's syntax tree. It contains a name and a list of rules.

```python class GrammarNode: """Node in the grammar syntax tree."""

```
def __init__(self, name, rules):
    self.name = name
    self.rules = rules

def accept(self, visitor):
    """Accept a visitor to traverse the grammar tree."""
    visitor.visit_grammar(self)
    for rule in self.rules:
        visitor.visit_rule(rule)
```

```

The `accept` method implements the Visitor pattern, allowing various operations to be performed on the syntax tree without modifying its structure.

2.4.2 DebugVisitor Class

The `DebugVisitor` class is a utility for debugging grammars. It traverses a grammar tree and prints information about each node.

```python class DebugVisitor: """Visitor for debugging grammar trees."""

```
def visit_grammar(self, grammar):
    print(f"Grammar: {grammar.name}")

def visit_rule(self, rule):
    print(f" Rule: {rule.name}")
```

```

This is useful for understanding the structure of a grammar and diagnosing issues with rule definitions.

2.5 Testing the Library

To ensure the reliability of the TinyPEG library, we need a comprehensive suite of tests. Let's create tests for each component, starting with the core classes.

Testing the Core Components

```python

# tests/test_core.py

import unittest from src.peg.core import Expression, Reference, ParserContext, ParseError

class TestParserContext(unittest.TestCase): def test_initialization(self): ctx = ParserContext("test") self.assertEqual(ctx.text, "test") self.assertEqual(ctx.pos, 0)
```

```
def test_eof(self):
    ctx = ParserContext("")
    self.assertTrue(ctx.eof())

    ctx = ParserContext("a")
    self.assertFalse(ctx.eof())
    ctx.pos = 1
    self.assertTrue(ctx.eof())

def test_peek(self):
    ctx = ParserContext("abc")
    self.assertEqual(ctx.peek(), "a")
    ctx.pos = 1
    self.assertEqual(ctx.peek(), "b")
    ctx.pos = 3  # Beyond the end
    self.assertIsNone(ctx.peek())

def test_consume(self):
    ctx = ParserContext("abc")
    self.assertEqual(ctx.consume(), "a")
    self.assertEqual(ctx.pos, 1)
    self.assertEqual(ctx.consume(), "b")
    self.assertEqual(ctx.pos, 2)
    self.assertEqual(ctx.consume(), "c")
    self.assertEqual(ctx.pos, 3)
    self.assertIsNone(ctx.consume())  # Beyond the end
    self.assertEqual(ctx.pos, 4)
```

class TestReference(unittest.TestCase): def test_initialization(self): ref = Reference("TestRule") self.assertEqual(ref.name, "TestRule")

```
def test_parse_not_implemented(self):
    ref = Reference("TestRule")
    ctx = ParserContext("test")
    with self.assertRaises(NotImplementedError):
        ref.parse(ctx)
```

if **name** == "**main**": unittest.main() ```

Testing the Parser Components

```python

# tests/test_parsers.py

import unittest from src.peg.parsers import PEGParser, Rule from src.peg.core import Expression, ParserContext

class MockExpression(Expression): def **init**(self, result=True): self.result = result self.called = False

```
def parse(self, ctx):
    self.called = True
    return self.result
```

class TestRule(unittest.TestCase): def test_initialization(self): expr = MockExpression() rule = Rule("TestRule", expr) self.assertEqual(rule.name, "TestRule") self.assertEqual(rule.expr, expr)

```
def test_parse(self):
    expr = MockExpression(True)
    rule = Rule("TestRule", expr)
    ctx = ParserContext("test")
    result = rule.parse(ctx)
    self.assertTrue(expr.called)
    self.assertTrue(result)
```

class TestPEGParser(unittest.TestCase): def test_initialization(self): parser = PEGParser() # Basic initialization test self.assertIsInstance(parser, PEGParser)

```
def test_parse(self):
    parser = PEGParser()
    result = parser.parse("test input")
```
```

```
    # For now, just check that it returns something
    self.assertIsNotNone(result)
```

if **name** == "**main**": unittest.main() ```

Testing the Syntax Tree Components

```python

# tests/test_syntax_tree.py

import unittest from src.peg.syntax_tree import GrammarNode, DebugVisitor from src.peg.parsers import Rule from src.peg.core import Expression

class MockRule(Rule): def **init**(self, name): super().**init**(name, None)

class MockVisitor: def **init**(self): self.visited_grammars = [] self.visited_rules = []

```
def visit_grammar(self, grammar):
    self.visited_grammars.append(grammar)

def visit_rule(self, rule):
    self.visited_rules.append(rule)
```

class TestGrammarNode(unittest.TestCase): def test_initialization(self): rules = [MockRule("Rule1"), MockRule("Rule2")] grammar = GrammarNode("TestGrammar", rules) self.assertEqual(grammar.name, "TestGrammar") self.assertEqual(grammar.rules, rules)

```
def test_accept(self):
    rules = [MockRule("Rule1"), MockRule("Rule2")]
    grammar = GrammarNode("TestGrammar", rules)
    visitor = MockVisitor()
    grammar.accept(visitor)
    self.assertEqual(len(visitor.visited_grammars), 1)
    self.assertEqual(visitor.visited_grammars[0], grammar)
    self.assertEqual(len(visitor.visited_rules), 2)
```
```

```
    self.assertEqual(visitor.visited_rules[0], rules[0])
    self.assertEqual(visitor.visited_rules[1], rules[1])
```

class TestDebugVisitor(unittest.TestCase): def test_visit_grammar(self): # This is more of a functional test than a unit test # since it just prints to stdout visitor = DebugVisitor() grammar = GrammarNode("TestGrammar", []) visitor.visit_grammar(grammar) # No assertion, just make sure it doesn't raise an exception

```
def test_visit_rule(self):
    visitor = DebugVisitor()
    rule = MockRule("TestRule")
    visitor.visit_rule(rule)
    # No assertion, just make sure it doesn't raise an exception
```

if **name** == "**main**": unittest.main() ```

These tests provide a foundation for ensuring the correctness of the TinyPEG library. As the library evolves, additional tests should be added to cover new functionality and edge cases.

In the next chapter, we'll build our first parser using the TinyPEG library, putting these components to practical use.