Appendix B: Testing Framework

Testing is a crucial aspect of parser development. This appendix provides guidance on testing parsers built with the TinyPEG library.

B.1 Unit Testing Parsers

Unit testing involves testing individual components of your parser in isolation. For a parser built with TinyPEG, you should test:

1. **Core Components**: Test the basic building blocks like `Expression`, `Reference`, and `ParserContext`.
2. **Grammar Rules**: Test each grammar rule individually.
3. **AST Construction**: Test that the parser correctly builds the AST.
4. **Error Handling**: Test that the parser correctly handles invalid input.

Here's an example of unit tests for 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() ```

And here's an example of unit tests for a specific parser:

```python
```

tests/test_calculator.py

import unittest from examples.peg_usage.calculators.advanced_calculator import AdvancedCalculator from src.peg.core import ParseError

class TestCalculatorParser(unittest.TestCase): def setUp(self): self.parser = AdvancedCalculator()

```
def test_parse_number(self):
    result = self.parser.parse("42")
    self.assertIsNotNone(result)
    # Test evaluation
    value = self.parser.evaluate("42")
    self.assertEqual(value, 42)

def test_parse_expression(self):
    result = self.parser.parse("2 + 3")
    self.assertIsNotNone(result)
    # Test evaluation
    value = self.parser.evaluate("2 + 3")
    self.assertEqual(value, 5)

def test_parse_complex_expression(self):
    result = self.parser.parse("2 * (3 + 4)")
    self.assertIsNotNone(result)
    # Test evaluation with proper precedence
    value = self.parser.evaluate("2 * (3 + 4)")
    self.assertEqual(value, 14)

def test_parse_invalid_expression(self):
    with self.assertRaises(ParseError):
        self.parser.parse("2 +")

def test_parse_empty_input(self):
    with self.assertRaises(ParseError):
        self.parser.parse("")
```

```
def test_precedence(self):
    # Test operator precedence
    value = self.parser.evaluate("2 + 3 * 4")
    self.assertEqual(value, 14)  # Should be 2 + (3 * 4) = 14, not (2 +
```

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

B.2 Test Case Design

When designing test cases for parsers, consider the following categories:

B.2.1 Valid Input Tests

Test that the parser correctly handles valid input:

- **Simple Cases**: Basic examples of valid input.
- **Complex Cases**: More complex examples that exercise multiple grammar rules.
- **Edge Cases**: Valid input that might be challenging to parse, such as deeply nested structures.
- **Boundary Cases**: Input that is at the boundary of what's valid, such as the maximum allowed nesting level.

B.2.2 Invalid Input Tests

Test that the parser correctly handles invalid input:

- **Syntax Errors**: Input with syntax errors, such as missing delimiters or invalid tokens.
- **Semantic Errors**: Input that is syntactically valid but semantically invalid, such as undefined variables.
- **Malformed Input**: Input that is malformed in various ways, such as unterminated strings or comments.
- **Empty Input**: Test how the parser handles empty input.

B.2.3 Performance Tests

Test the parser's performance:

- **Large Input**: Test with large input files to ensure the parser can handle them efficiently.
- **Pathological Cases**: Test with input that might cause performance issues, such as deeply nested expressions.
- **Memory Usage**: Monitor memory usage to ensure the parser doesn't use excessive memory.

B.2.4 Regression Tests

Create regression tests to ensure that bugs, once fixed, don't reappear:

- **Bug Fixes**: For each bug you fix, add a test case that would have caught the bug.
- **Feature Additions**: When adding new features, add tests to ensure they work correctly and don't break existing functionality.
- **Refactoring**: After refactoring, run all tests to ensure you haven't introduced regressions.

B.3 Debugging Techniques

Debugging parsers can be challenging. Here are some techniques to help:

B.3.1 Tracing

Add tracing to your parser to see what it's doing:

```
python def parse_expression(self, ctx): print(f"Parsing expression at position {ctx.pos}: '{ctx.text[ctx.pos:ctx.pos+10]}...'") # ... parsing logic ... print(f"Expression result: {result}") return result
```

B.3.2 Visualization

Visualize the parse tree or AST to understand what the parser is producing:

```python def visualize_ast(ast, indent=0): """Visualize an AST.""" if ast is None: return
```

```
    print(" " * indent + f"{type(ast).__name__}")

    if hasattr(ast, "value"):
        print(" " * (indent + 2) + f"value: {ast.value}")

    if hasattr(ast, "name"):
        print(" " * (indent + 2) + f"name: {ast.name}")

    if hasattr(ast, "op"):
        print(" " * (indent + 2) + f"op: {ast.op}")

    if hasattr(ast, "left"):
        print(" " * (indent + 2) + "left:")
        visualize_ast(ast.left, indent + 4)

    if hasattr(ast, "right"):
        print(" " * (indent + 2) + "right:")
        visualize_ast(ast.right, indent + 4)

    if hasattr(ast, "statements"):
        print(" " * (indent + 2) + "statements:")
        for stmt in ast.statements:
            visualize_ast(stmt, indent + 4)
```

```

### B.3.3 Step-by-Step Execution

Use a debugger to step through the parsing process:

1. Set a breakpoint at the start of the parsing function.
2. Run the parser in debug mode.
3. Step through the code to see what's happening.
4. Inspect variables to understand the parser's state.
```

B.3.4 Simplified Test Cases

When debugging a complex issue, try to create a simplified test case that reproduces the issue:

1. Start with the input that's causing the issue.
2. Gradually simplify the input while still reproducing the issue.
3. Once you have a minimal test case, it's often easier to understand and fix the issue.

B.3.5 Logging

Add logging to your parser to record its actions:

```python import logging

logging.basicConfig(level=logging.DEBUG) logger = logging.getLogger(**name**)

def parse_expression(self, ctx): logger.debug(f"Parsing expression at position {ctx.pos}") # ... parsing logic ... logger.debug(f"Expression result: {result}") return result ```

## B.4 Testing Tools

Here are some tools that can help with testing parsers:

### B.4.1 unittest

Python's built-in `unittest` framework is a good choice for unit testing parsers:

```python import unittest

class TestParser(unittest.TestCase): def setUp(self): self.parser = MyParser()

```
def test_parse_valid_input(self):
    result = self.parser.parse("valid input")
    self.assertIsNotNone(result)
    # Add more specific assertions

def test_parse_invalid_input(self):
```

```
    with self.assertRaises(ParseError):
        self.parser.parse("invalid input")
```

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

B.4.2 pytest

`pytest` is a more powerful testing framework that can simplify test writing:

```python import pytest from my_parser import MyParser, ParseError

@pytest.fixture def parser(): return MyParser()

def test_parse_valid_input(parser): result = parser.parse("valid input") assert result is not None # Add more specific assertions

def test_parse_invalid_input(parser): with pytest.raises(ParseError): parser.parse("invalid input") ```

### B.4.3 Coverage.py

`Coverage.py` is a tool for measuring code coverage of Python programs:

```
bash pip install coverage coverage run -m unittest discover coverage report coverage html # Generate HTML report
```

### B.4.4 Hypothesis

`Hypothesis` is a property-based testing library that can generate test cases for you:

```python import hypothesis from hypothesis import given from hypothesis.strategies import text from my_parser import MyParser, ParseError

@given(text()) def test_parser_handles_arbitrary_input(s): parser = MyParser() try: result = parser.parse(s) # If parsing succeeds, make assertions about the result assert result is not None except ParseError: # If parsing fails, that's fine too pass ```

## B.5 Continuous Integration

Set up continuous integration (CI) to automatically run your tests:

### B.5.1 GitHub Actions

```yaml

# .github/workflows/test.yml

name: Test

on: push: branches: [ main ] pull_request: branches: [ main ]

jobs: test: runs-on: ubuntu-latest steps: - uses: actions/checkout@v2 - name: Set up Python uses: actions/setup-python@v2 with: python-version: '3.x' - name: Install dependencies run: | python -m pip install --upgrade pip pip install -r requirements.txt - name: Run tests run: | python -m unittest discover ```

### B.5.2 Travis CI

```yaml

# .travis.yml

language: python python: - "3.6" - "3.7" - "3.8" - "3.9" install: - pip install -r requirements.txt script: - python -m unittest discover ```

## Summary

Testing is an essential part of parser development. By thoroughly testing your parsers, you can ensure they correctly handle both valid and invalid input, perform well, and don't regress when you make changes.

In this appendix, we've covered unit testing, test case design, debugging techniques, testing tools, and continuous integration. These practices will help you build robust and reliable parsers with the TinyPEG library.