Chapter 3: Building Your First Parser

3.1 Setting Up the Environment

Before we start building parsers with TinyPEG, let's ensure our environment is properly set up. We'll need:

1. Python 3.6 or later
2. The TinyPEG library (which we're building in this tutorial)

Let's create a simple project structure:

`my_parser_project/ ├── src/ │ └── peg/ # The TinyPEG library ├── examples/ │ └── simple_parser.py # Our first parser └── tests/ └── test_simple_parser.py # Tests for our parser`

For this chapter, we'll focus on creating a simple numeric expression parser that can handle basic arithmetic operations.

3.2 Creating a Simple Numeric Expression Parser

Let's start by creating a parser that can recognize and evaluate simple numeric expressions like "42" or "3.14".

First, we need to import the necessary components from the TinyPEG library:

```python

# examples/simple_parser.py

from src.peg import PEGParser, Rule, Reference, ParseError, Regex from src.peg.syntax_tree import GrammarNode

class NumberParser(PEGParser): def **init**(self): super().**init**()

```
    # Define grammar for numbers
    self.grammar = GrammarNode(
```
```

```
            name="Number",
            rules=[
                Rule("Number", Regex("[0-9]+"))
            ]
        )

    def parse(self, text: str):
        print(f"Parsing number: {text}")
        try:
            result = super().parse(text)
            # Convert string result to integer
            if isinstance(result, str) and result.isdigit():
                return int(result)
            return result
        except ParseError as e:
            print(f"Parse error: {e}")
            return None
```

~~~

This simple parser can recognize integer numbers. Let's test it:

```python

# Test the number parser

if **name** == "**main**": parser = NumberParser() result = parser.parse("42") print(f"Result: {result}") ```

Running this should output: `Parsing number: 42 Result: 42`

Of course, this is a very simplified implementation. In a real parser, we would: 1. Create a `ParserContext` with the input text 2. Apply the grammar's start rule to the context 3. Build and return a parse tree or AST

Let's enhance our parser to do this properly.
~~~

## 3.3 Adding Support for Operators

Now, let's extend our parser to handle basic arithmetic operators: addition, subtraction, multiplication, and division.

In PEG, we need to carefully define the precedence of operators. We'll use the following grammar:

```
Expression ::= Term (('+' | '-') Term)* Term ::= Factor (('*'
| '/') Factor)* Factor ::= Number | '(' Expression ')'
Number ::= [0-9]+
```

This grammar ensures that multiplication and division have higher precedence than addition and subtraction.

Let's implement this grammar:

```python

# examples/arithmetic_parser.py

from src.peg import ( PEGParser, Rule, Reference, ParseError, Sequence, Choice, ZeroOrMore, Literal, Regex ) from src.peg.syntax_tree import GrammarNode

class ArithmeticParser(PEGParser): def **init**(self): super().**init**()

```
    # Define grammar with proper precedence
    self.grammar = GrammarNode(
        name="Expression",
        rules=[
            # Expression = Term (('+' | '-') Term)*
            Rule("Expression", Sequence(
                Reference("Term"),
                ZeroOrMore(
                    Sequence(
                        Choice(Literal("+"), Literal("-")),
                        Reference("Term")
)
)
```
```

```
            )),

            # Term = Factor (('*' | '/') Factor)*
            Rule("Term", Sequence(
                Reference("Factor"),
                ZeroOrMore(
                    Sequence(
                        Choice(Literal("*"), Literal("/")),
                        Reference("Factor")
                    )
                )
            )),

            # Factor = Number | '(' Expression ')'
            Rule("Factor", Choice(
                Reference("Number"),
                Sequence(
                    Literal("("),
                    Reference("Expression"),
                    Literal(")")
                )
            )),

            # Number = [0-9]+
            Rule("Number", Regex("[0-9]+"))
        ]
    )

def skip_whitespace(self, ctx):
    """Skip whitespace in the input."""
    while not ctx.eof() and ctx.peek() in " \t\n\r":
        ctx.consume()

def _parse_expression(self, expr, ctx):
    """Override to handle whitespace between tokens."""
    # Skip whitespace before parsing
    self.skip_whitespace(ctx)
    # Parse the expression
    result = super()._parse_expression(expr, ctx)

    # Skip whitespace after parsing
    self.skip_whitespace(ctx)

    return result

def evaluate(self, text: str):
    """Parse and evaluate an arithmetic expression."""
    ast = self.parse(text)
    return self._evaluate_node(ast)

def _evaluate_node(self, node):
    """Evaluate an AST node with proper precedence."""
    if isinstance(node, str):
        if node.isdigit():
            return int(node)
        elif node in "+-*/()":
            return node
        else:
            return node
    elif isinstance(node, list):
        if len(node) == 1:
            return self._evaluate_node(node[0])
        elif len(node) == 3 and node[0] == "(" and node[2] == ")":
            # Parenthesized expression
            return self._evaluate_node(node[1])
        else:
            # Handle operations with proper precedence
            return self._evaluate_expression_list(node)
    else:
        return node

def _evaluate_expression_list(self, nodes):
    """Evaluate a list of nodes representing an expression."""
    # Convert to a flat list
    flat = []
    for node in nodes:
        if isinstance(node, list):
            flat.extend(self._flatten_node(node))
        else:
            flat.append(self._evaluate_node(node))

    # Handle multiplication and division first (higher precedence)
    i = 1
    while i < len(flat):
        if flat[i] == '*':
            result = flat[i-1] * flat[i+1]
            flat = flat[:i-1] + [result] + flat[i+2:]
        elif flat[i] == '/':
            result = flat[i-1] / flat[i+1]
            flat = flat[:i-1] + [result] + flat[i+2:]
        else:
            i += 2

    # Handle addition and subtraction (lower precedence)
    i = 1
    while i < len(flat):
        if flat[i] == '+':
            result = flat[i-1] + flat[i+1]
            flat = flat[:i-1] + [result] + flat[i+2:]
        elif flat[i] == '-':
            result = flat[i-1] - flat[i+1]
            flat = flat[:i-1] + [result] + flat[i+2:]
        else:
            i += 2

    return flat[0] if flat else 0

def _flatten_node(self, node):
    """Flatten a nested node structure."""
    if isinstance(node, list):
        result = []
        for item in node:
            if isinstance(item, list):
                result.extend(self._flatten_node(item))
            else:
                result.append(self._evaluate_node(item))
        return result
    else:
        return [self._evaluate_node(node)]
```

~~~

This parser is more complex but still incomplete. In a real implementation, we would need to: 1. Properly handle whitespace 2. Convert the parse tree into an AST 3. Evaluate the AST to compute the result

Let's address these issues in the next sections.

3.4 Handling Parentheses and Precedence

Our grammar already accounts for operator precedence and parentheses, but our parser implementation needs to be enhanced to properly build and evaluate the parse tree.

Let's add some AST node classes:

```python

# AST node classes

class ASTNode: """Base class for AST nodes.""" pass

class NumberNode(ASTNode): """AST node for numbers.""" def **init**(self, value): self.value = value

~~~
def evaluate(self):
    return self.value
```
```

class BinaryOpNode(ASTNode): """AST node for binary operations.""" def **init**(self, op, left, right): self.op = op self.left = left self.right = right

```
def evaluate(self):
    left_val = self.left.evaluate()
    right_val = self.right.evaluate()

    if self.op == "+":
        return left_val + right_val
    elif self.op == "-":
        return left_val - right_val
    elif self.op == "*":
        return left_val * right_val
    elif self.op == "/":
        return left_val / right_val
    else:
        raise ValueError(f"Unknown operator: {self.op}")
```

~~~

Now we need to modify our parser to build these AST nodes during parsing.

3.5 Building an Abstract Syntax Tree

To build an AST, we need to modify our expression classes to construct AST nodes as they parse:

```python class RegexMatcher(Expression): """Match a regular expression pattern.""" def **init**(self, pattern): import re self.pattern = re.compile(pattern)

~~~
def parse(self, ctx):
    if ctx.eof():
        return None

    match = self.pattern.match(ctx.text[ctx.pos:])
    if match:
        matched = match.group(0)
        ctx.pos += len(matched)
```
```

```
        return matched
    return None
```

Modify our parser to build AST nodes

class ArithmeticParser(PEGParser): def **init**(self): super().**init**()

```
    # Define grammar for arithmetic expressions
    # (Same as before)

def parse_expression(self, ctx):
    # Parse a term
    left = self.parse_term(ctx)
    if left is None:
        return None

    # Parse any following +/- operations
    while not ctx.eof():
        # Try to parse an operator
        op_pos = ctx.pos
        if ctx.text[op_pos] == '+' or ctx.text[op_pos] == '-':
            op = ctx.text[op_pos]
            ctx.pos += 1

            # Parse the right term
            right = self.parse_term(ctx)
            if right is None:
                # Backtrack if the right term fails
                ctx.pos = op_pos
                break

            # Create a binary operation node
            left = BinaryOpNode(op, left, right)
        else:
            break
    return left

def parse_term(self, ctx):
    # Parse a factor
    left = self.parse_factor(ctx)
    if left is None:
        return None

    # Parse any following */÷ operations
    while not ctx.eof():
        # Try to parse an operator
        op_pos = ctx.pos
        if ctx.text[op_pos] == '*' or ctx.text[op_pos] == '/':
            op = ctx.text[op_pos]
            ctx.pos += 1

            # Parse the right factor
            right = self.parse_factor(ctx)
            if right is None:
                # Backtrack if the right factor fails
                ctx.pos = op_pos
                break

            # Create a binary operation node
            left = BinaryOpNode(op, left, right)
        else:
            break

    return left

def parse_factor(self, ctx):
    # Skip whitespace
    while not ctx.eof() and ctx.text[ctx.pos].isspace():
        ctx.pos += 1

    if ctx.eof():
        return None
    # Try to parse a number
    if ctx.text[ctx.pos].isdigit():
        start = ctx.pos
        while not ctx.eof() and ctx.text[ctx.pos].isdigit():
            ctx.pos += 1
        return NumberNode(int(ctx.text[start:ctx.pos]))

    # Try to parse a parenthesized expression
    if ctx.text[ctx.pos] == '(':
        ctx.pos += 1

        # Parse the inner expression
        expr = self.parse_expression(ctx)
        if expr is None:
            return None

        # Expect a closing parenthesis
        if ctx.eof() or ctx.text[ctx.pos] != ')':
            return None
        ctx.pos += 1

        return expr

    return None

def parse(self, text: str):
    print(f"Parsing expression: {text}")
    # Create a ParserContext
    ctx = ParserContext(text)

    # Skip initial whitespace
    while not ctx.eof() and ctx.text[ctx.pos].isspace():
        ctx.pos += 1

    # Parse the expression
    ast = self.parse_expression(ctx)
    # Skip trailing whitespace
    while not ctx.eof() and ctx.text[ctx.pos].isspace():
        ctx.pos += 1

    # Check if we consumed all input
    if ctx.eof():
        return ast
    else:
        print(f"Error: Unexpected input at position {ctx.pos}")
        return None
```

~~~

This implementation directly builds an AST during parsing, rather than first building a parse tree and then converting it to an AST. This is a common approach in hand-written recursive descent parsers.

3.6 Evaluating Expressions

Now that we have an AST, evaluating expressions is straightforward:

```python

# Test the arithmetic parser

if **name** == "**main**": parser = ArithmeticParser()

~~~
test_expressions = [
    "42",
    "2+3",
    "2 + 3",
    "2 * 3 + 4",
    "2 + 3 * 4",
    "(2 + 3) * 4",
    "10 / 2 - 3"
]
```
```

```python
print("=== Arithmetic Parser Test ===")
for expr in test_expressions:
    try:
        result = parser.evaluate(expr)
        print(f"{expr} = {result}")
    except Exception as e:
        print(f"Failed to parse: {expr} - Error: {e}")
```

\`\`\`

Running this should output: `=== Arithmetic Parser Test === 42 = 42 2+3 = 5 2 + 3 = 5 2 * 3 + 4 = 10 2 + 3 * 4 = 14 (2 + 3) * 4 = 20 10 / 2 - 3 = 2.0`

This demonstrates that our parser correctly handles operator precedence and parentheses.

Summary

In this chapter, we've built a simple arithmetic expression parser using the TinyPEG library. We've learned how to:

1. Define a grammar for arithmetic expressions
2. Implement parsing functions for each grammar rule
3. Build an AST during parsing
4. Evaluate the AST to compute the result

While our implementation is still somewhat simplified, it demonstrates the key concepts of recursive descent parsing and AST construction. In a real-world parser, we would need to handle more complex cases, such as:

- Error reporting and recovery
- More sophisticated tokenization
- Support for variables and functions
- Type checking and semantic analysis

In the next chapter, we'll explore the example parsers included with the TinyPEG library, which demonstrate these more advanced features.