Chapter 4: Example Parsers and Applications

In this chapter, we'll explore the comprehensive collection of example parsers included with the TinyPEG library. These examples demonstrate different aspects of parser implementation, from simple calculators to complete programming languages, and can serve as templates for your own parsers.

Our examples are organized into three main categories: - **Calculator Examples**: Arithmetic expression parsers with increasing complexity - **Language Parser Examples**: Simple programming language constructs - **TinyCL Language**: Complete programming language implementation

4.1 Calculator Examples

The calculator examples demonstrate how to build arithmetic expression parsers with increasing complexity, showing proper operator precedence and evaluation.

4.1.1 Simple Calculator

The simple calculator (`examples/peg_usage/calculators/simple_calculator.py`) supports only addition and subtraction:

```python

# !/usr/bin/env python3

""" Simple calculator example using the TinyPEG library. Supports only addition and subtraction. """

from calculator_base import SimpleCalculator

def main(): """Test the simple calculator.""" calculator = SimpleCalculator()
```

```
# Test expressions for simple arithmetic
expressions = [
    "3",
    "42",
    "3+5",
    "3 + 5",
    "10 - 4",
    "3 + 5 - 2",
    "100 + 200 - 50"
]

print("=== Simple Calculator (Addition/Subtraction Only) ===")
calculator.test_expressions(expressions)
```

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

This calculator uses a base class that handles the common parsing logic and provides a clean interface for testing expressions.

4.1.2 Advanced Calculator

The advanced calculator (`examples/peg_usage/calculators/advanced_calculator.py`) supports full arithmetic with proper precedence:

```python

# !/usr/bin/env python3

""" Advanced calculator with full arithmetic operations and precedence. """

from calculator_base import AdvancedCalculator

def main(): """Test the advanced calculator.""" calculator = AdvancedCalculator()

```
# Test expressions with precedence and parentheses
expressions = [
    "3",
    "42",
```
```

```
    "3+5",
    "3 + 5",
    "10 - 4",
    "3 * 5",
    "10 / 2",
    "3 + 5 * 2",         # Should be 13 with proper precedence
    "10 - 2 * 3",        # Should be 4 with proper precedence
    "3 * 5 + 2",         # Should be 17
    "10 / 2 - 3",        # Should be 2
    "(3 + 5) * 2",       # Should be 16
    "3 + (5 * 2)",       # Should be 13
    "3 * (5 + 2)",       # Should be 21
    "(3 + 5) * (2 + 1)"  # Should be 24
]

print("=== Advanced Calculator (Full Arithmetic with Precedence) ===")
calculator.test_expressions(expressions)
```

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

4.1.3 Calculator Base Classes

Both calculators inherit from base classes in `calculator_base.py` that provide the core parsing logic:

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

class AdvancedCalculator(PEGParser): """Advanced calculator with full arithmetic operations and precedence."""

```
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]+"))
        ]
    )
~~~

```
```

This implementation demonstrates proper operator precedence and parentheses handling.

4.1.4 Testing and Usage

To test the calculator examples, run them directly:

```bash

# Test simple calculator (addition/ subtraction only)

cd examples/peg_usage/calculators python simple_calculator.py

# Test advanced calculator (full arithmetic with precedence)

python advanced_calculator.py ```

To use the calculators in your own code:

```python from examples.peg_usage.calculators.calculator_base import AdvancedCalculator

calculator = AdvancedCalculator() result = calculator.evaluate("3 + 5 * 2") print(f"Result: {result}") # Output: Result: 13 ```

## 4.2 Language Parser Examples

The language parser examples demonstrate how to parse various programming language constructs. These are located in `examples/peg_usage/language_parsers/`.

### 4.2.1 Basic Language Constructs

Our examples include parsers for fundamental programming language elements:

**Number Parser**

The simplest example (`number_parser.py`) parses just numbers:

```python from src.peg import PEGParser, Rule, Regex from src.peg.syntax_tree import GrammarNode

class NumberParser(PEGParser): def **init**(self): super().**init**() self.grammar = GrammarNode( name="Number", rules=[ Rule("Number", Regex("[0-9]+")) ] ) ```

**If Statement Parser**

The if statement parser (`ifstmt.py`) demonstrates conditional parsing:

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

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

```
    self.grammar = GrammarNode(
        name="IfStatement",
        rules=[
            Rule("IfStatement", Sequence(
                Literal("if"),
                Literal("("),
                Reference("Condition"),
                Literal(")"),
                Reference("Block")
            )),
            Rule("Condition", Reference("Expression")),
            Rule("Block", Sequence(
                Literal("{"),
                Reference("Statement"),
                Literal("}")
            )),
            Rule("Statement", Reference("PrintStatement")),
            Rule("PrintStatement", Sequence(
                Literal("print"),
                Literal("("),
```

```
                Reference("Expression"),
                Literal(")"),
                Literal(";")
            )),
            Rule("Expression", Reference("Identifier")),
            Rule("Identifier", Regex("[a-zA-Z_][a-zA-Z0-9_]*"))
        ]
    )
```

While Loop Parser

The while loop parser (`while.py`) handles iterative constructs:

```python
class WhileLoopParser(PEGParser): def init(self): super().init()

    self.grammar = GrammarNode(
        name="WhileLoop",
        rules=[
            Rule("WhileLoop", Sequence(
                Literal("while"),
                Literal("("),
                Reference("Condition"),
                Literal(")"),
                Reference("Block")
            )),
            # ... similar structure to if statement
        ]
    )
```

4.2.2 TinyCL Language Variants

We also have several TinyCL language parser examples that demonstrate different levels of complexity:

Minimal TinyCL

The minimal TinyCL parser (`minimal_tinycl.py`) implements a very basic subset:

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

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

```
    self.grammar = GrammarNode(
        name="MinimalTinyCL",
        rules=[
            Rule("Program", ZeroOrMore(Reference("Statement"))),
            Rule("Statement", Choice(
                Reference("VariableDecl"),
                Reference("PrintStatement")
            )),
            Rule("VariableDecl", Sequence(
                Literal("var"),
                Reference("Identifier"),
                Literal("="),
                Reference("Expression"),
                Literal(";")
            )),
            Rule("PrintStatement", Sequence(
                Literal("print"),
                Literal("("),
                Reference("Expression"),
                Literal(")"),
                Literal(";")
            )),
            Rule("Expression", Reference("Number")),
            Rule("Number", Regex("[0-9]+")),
            Rule("Identifier", Regex("[a-zA-Z_][a-zA-Z0-9_]*"))
        ]
)
```

```

Simple TinyCL

The simple TinyCL parser (`simple_tinycl.py`) adds arithmetic expressions:

```python

# Extends minimal TinyCL with arithmetic operations

Rule("Expression", Sequence( Reference("Term"),
ZeroOrMore(Sequence( Choice(Literal("+"), Literal("-")), Reference("Term") )) )),
Rule("Term", Sequence( Reference("Factor"),
ZeroOrMore(Sequence( Choice(Literal("*"), Literal("/")), Reference("Factor") )) )),
Rule("Factor", Choice( Reference("Number"), Reference("Identifier"),
Sequence(Literal("("), Reference("Expression"), Literal(")")) )) ```

**Standalone TinyCL**

The standalone TinyCL parser (`standalone_tinycl.py`) is a complete, self-contained implementation that can be used independently.

## 4.3 Complete TinyCL Implementation

The complete TinyCL (Tiny C-Like Language) implementation is our flagship example, demonstrating a full-featured programming language with parser, interpreter, and compiler.

### 4.3.1 TinyCL Features

The complete TinyCL implementation (`examples/tinycl_language/`) includes:

- **Variables and Constants**: `var x = 10;` and `const PI = 3;`
- **Functions**: `func add(a, b) { return a + b; }`
- **Arrays**: `[1, 2, 3]` with indexing `arr[0]`
- **Control Flow**: If-else statements and while loops
- **Full Expression System**: Arithmetic, logical, and comparison operators
- **Data Types**: Numbers, strings, characters, booleans, arrays
```

- **Built-in Functions**: `print()` for output

4.3.2 Comprehensive Test

The comprehensive test (`comprehensive_test.py`) demonstrates all TinyCL features:

```python

# !/usr/bin/env python3

""" Comprehensive test of the TinyCL language implementation. """

from src.tinycl.parser import TinyCLParser from src.tinycl.interpreter import TinyCLInterpreter

def test_complete_program(): """Test a complete TinyCL program with all features."""

```
program = '''
# TinyCL Comprehensive Test Program

# Constants and variables
const MAX = 10;
var numbers = [5, 3, 8, 1, 9];
var sum = 0;

# Function to calculate factorial
func factorial(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

# Calculate sum of array
var i = 0;
while (i < 5) {
```
```

```
    sum = sum + numbers[i];
    i = i + 1;
}

print("Array sum: " + sum);

# Test factorial function
var fact5 = factorial(5);
print("Factorial of 5: " + fact5);

# Test logical operations
if (sum > 20 && fact5 > 100) {
    print("Both conditions are true!");
}
'''

# Parse the program
parser = TinyCLParser()
ast = parser.parse(program)

# Interpret the program
interpreter = TinyCLInterpreter()
interpreter.interpret(ast)
```

if **name** == "**main**": test_complete_program() ```

4.3.3 Multi-Target Compilation

TinyCL also includes compilers that can generate code for different targets:

Python Compiler

```python from src.tinycl.compiler import PythonCompiler

compiler = PythonCompiler() python_code = compiler.compile(ast) print("Generated Python code:") print(python_code) ```
```

C Compiler

```python from src.tinycl.compiler import CCompiler

compiler = CCompiler() c_code = compiler.compile(ast) print("Generated C code:") print(c_code) ```

## 4.4 Running the Examples

All examples can be run directly from their respective directories:

### Calculator Examples

```bash

# Navigate to calculator examples

cd examples/peg_usage/calculators

# Run simple calculator

python simple_calculator.py

# Run advanced calculator

python advanced_calculator.py

# Run number parser

python number_parser.py ```

### Language Parser Examples

```bash
```

Navigate to language parser examples

cd examples/peg_usage/language_parsers

Run minimal TinyCL

python minimal_tinycl.py

Run simple TinyCL

python simple_tinycl.py

Run if statement parser

python ifstmt.py

Run while loop parser

python while.py

Run EmLang parser

python emlang.py ```

Complete TinyCL

```bash

# Navigate to TinyCL examples

cd examples/tinycl_language
```

Run comprehensive test

python comprehensive_test.py ```

4.5 Example Organization

Our examples are organized to show progression from simple to complex:

Category	Complexity	Features Demonstrated
Basic Parsers	Simple	Single constructs, basic parsing
Calculator Examples	Medium	Expression parsing, precedence, evaluation
Language Parsers	Medium-High	Multiple constructs, grammar composition
TinyCL Complete	High	Full language, interpreter, compiler

Summary

This chapter has explored the comprehensive collection of example parsers included with the TinyPEG library. These examples demonstrate:

1. **Progressive Complexity**: From simple number parsing to complete programming languages
2. **Real-World Applications**: Practical examples that can be adapted for your own projects
3. **Best Practices**: Proper grammar design, error handling, and code organization
4. **Complete Implementation**: Full language implementation with parser, interpreter, and compiler

Key takeaways:

- **Start Simple**: Begin with basic constructs and gradually add complexity
- **Proper Structure**: Organize grammars with clear precedence and modularity
- **Testing**: Each example includes comprehensive testing to verify functionality
- **Documentation**: All examples are well-documented and self-contained

By studying these examples and experimenting with them, you'll gain practical experience with the TinyPEG library and be ready to build parsers for your own domain-specific languages and applications.

The examples serve as both learning tools and starting points for your own parser projects. Whether you're building a simple calculator or a complete programming language, these examples provide the foundation and patterns you need to succeed.