Appendix A: TinyPEG Library Reference

A.1 TinyPEG Implementation Overview

This appendix provides a comprehensive reference for our TinyPEG library implementation. Unlike traditional PEG notation, our library uses Python classes to represent parsing expressions, providing a programmatic approach to grammar definition.

A.1.1 Core Architecture

Our TinyPEG library consists of three main modules:

- **core.py**: Fundamental classes (`Expression`, `Reference`, `ParserContext`, `ParseError`, `Rule`, `GrammarNode`)
- **parsers.py**: Complete PEG expression implementations and the main `PEGParser` class
- **syntax_tree.py**: Grammar representation and visitor pattern support

A.1.2 Grammar Definition

In our implementation, grammars are defined using Python classes rather than traditional PEG notation:

```python from src.peg import Rule, GrammarNode, Literal, Reference

# Define a grammar using Python classes

grammar = GrammarNode( name="MyGrammar", rules=[ Rule("Start", Reference("Expression")), Rule("Expression", Literal("hello")) ] ) ```
```

A.1.3 TinyPEG Expression Classes

Our library implements PEG expressions as Python classes:

| Class | Description | Usage Example | |-------|-------------|---------------| | `Literal` | Match a literal string | `Literal("while")` | | `Regex` | Match a regular expression pattern | `Regex("[0-9]+")` | | `Sequence` | Match expressions in order | `Sequence(Literal("if"), Reference("Condition"))` | | `Choice` | Try alternatives in order | `Choice(Reference("IfStmt"), Reference("WhileStmt"))` | | `ZeroOrMore` | Match zero or more times | `ZeroOrMore(Reference("Statement"))` | | `OneOrMore` | Match one or more times | `OneOrMore(Regex("[0-9]"))` | | `Optional` | Match optionally | `Optional(Sequence(Literal("else"), Reference("Block")))` | | `AndPredicate` | Positive lookahead (don't consume) | `AndPredicate(Regex("[a-z]"))` | | `NotPredicate` | Negative lookahead (don't consume) | `NotPredicate(Regex("[0-9]"))` | | `Reference` | Reference to another rule | `Reference("Expression")` |

A.1.4 Complete Example

Here's a complete example showing how to define and use a grammar:

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

# Define a simple arithmetic grammar

grammar = GrammarNode( name="Arithmetic", rules=[ 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"), Sequence(Literal("("), Reference("Expression"), Literal(")")) )), Rule("Number", Regex("[0-9]+")) ] )

# Create and use the parser

parser = PEGParser() parser.grammar = grammar
```

result = parser.parse("2 + 3 * 4") print(result) # Parses successfully ```

A.1.5 TinyPEG Semantics

Our TinyPEG implementation follows standard PEG semantics:

1. **Ordered Choice**: The `Choice` class tries alternatives in order, selecting the first successful match
2. **Unlimited Lookahead**: Predicates (`AndPredicate`, `NotPredicate`) can look ahead without consuming input
3. **Memoization**: Our parser includes basic memoization to improve performance
4. **Automatic Whitespace Handling**: The parser automatically skips whitespace between tokens

A.2 Common TinyPEG Patterns

Here are common patterns implemented using our library:

A.2.1 Whitespace Handling

```python

# Our parser automatically handles whitespace, but you can control it:

class MyParser(PEGParser): def skip_whitespace(self, ctx): """Custom whitespace handling.""" while not ctx.eof() and ctx.peek() in " \t\n\r": ctx.consume() ```

### A.2.2 Identifiers

```python

# Match an identifier (letters, digits, underscore)

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

### A.2.3 Numbers

```python

# Match an integer

Rule("Integer", Regex("[0-9]+"))

# Match a floating-point number

Rule("Float", Regex("[0-9]+\.[0-9]+")) ```

### A.2.4 Strings

```python

# Match a double-quoted string

Rule("String", Regex("\"[^\"]*\""))

# More complex string with escape sequences

Rule("String", Sequence( Literal("\""), ZeroOrMore(Choice( Regex("[^\"\\]"), # Normal characters Sequence(Literal("\"), Regex(".")) # Escape sequences )), Literal("\"") )) ```

### A.2.5 Comments

```python

# Match a single-line comment

Rule("Comment", Regex("#[^\n]*"))
```

Match a multi-line comment

Rule("MultiLineComment", Sequence(Literal("/*"),
ZeroOrMore(Sequence(NotPredicate(Literal("*/")), Regex("."))), Literal("*/"))) ```

A.2.6 Expressions with Precedence

```python

# Expression with proper precedence levels

rules = [ 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"), Sequence(Literal("("), Reference("Expression"), Literal(")")) )), Rule("Number", Regex("[0-9]+")) ] ```

## A.3 Comparison with Regular Expressions

PEGs and regular expressions are both pattern-matching formalisms, but they have different capabilities and use cases:

| Feature | Regular Expressions | PEGs | |---------|---------------------|------| | Recursion | No | Yes | | Context-Sensitivity | Limited | Yes | | Backtracking | Implementation-dependent | Yes | | Ambiguity | Possible | No | | Lookahead | Limited | Unlimited | | Capture Groups | Yes | Implementation-dependent | | Performance | Generally faster | Can be slower without memoization |

### A.3.1 When to Use Regular Expressions

Regular expressions are best suited for: - Simple pattern matching - Lexical analysis (tokenization) - Search and replace operations - Validation of simple formats (e.g., email addresses, phone numbers)
```

A.3.2 When to Use PEGs

PEGs are better suited for: - Parsing structured languages - Handling nested constructs - Context-sensitive parsing - Building parsers for domain-specific languages

A.3.3 Converting Between Regular Expressions and PEGs

Many regular expressions can be directly translated to PEGs:

| Regular Expression | PEG Equivalent | |--------------------|----------------| | a | "a" | | a\|b | "a" / "b" | | a* | "a"* | | a+ | "a"+ | | a? | "a"? | | [a-z] | [a-z] | | (ab) | ("a" "b") | | ^a | !. "a" | | a$ | "a" !. |

However, some regular expression features, like backreferences, don't have direct equivalents in PEGs.

A.4 TinyPEG Implementation Details

Our TinyPEG library addresses several important implementation considerations:

A.4.1 Memoization

Our `PEGParser` class includes basic memoization to prevent exponential time complexity:

```python class PEGParser: def **init**(self): self.rule_cache = {} # Memoization cache

```
def _parse_rule(self, rule, ctx):
    # Check cache first
    cache_key = (rule.name, ctx.pos)
    if cache_key in self.rule_cache:
        result, new_pos = self.rule_cache[cache_key]
        ctx.pos = new_pos
        return result

    # Parse and cache result
    result = self._parse_expression(rule.expr, ctx)
```
```

```
    self.rule_cache[cache_key] = (result, ctx.pos)
    return result
```

```

### A.4.2 Left Recursion Handling

Our library handles left recursion by rewriting grammars to use right recursion with repetition:

```python

# Instead of left recursion, use this right-recursive pattern:

Rule("Expression", Sequence( Reference("Term"), ZeroOrMore(Sequence(Literal("+"), Reference("Term"))) )) ```

### A.4.3 Error Reporting

Our parser provides detailed error messages with position information:

`python try: result = parser.parse("invalid input") except ParseError as e: print(f"Parse error: {e}") # Output: Parse error: Expected pattern '[0-9]+', got 'invalid...'`

### A.4.4 AST Building

Our library supports AST building through custom parser classes:

```python class MyParser(PEGParser): def parse(self, text): result = super().parse(text) return self._build_ast(result)

```
def _build_ast(self, parse_result):
    # Convert parse result to AST nodes
    return MyASTNode(parse_result)
```

```

A.4.5 Whitespace Handling

Automatic whitespace handling is built into our parser:

```
python class PEGParser: def _parse_rule(self, rule, ctx): ctx.skip_whitespace() # Skip whitespace before parsing result = self._parse_expression(rule.expr, ctx) return result
```

A.5 TinyPEG vs Other PEG Libraries

Our TinyPEG library compared to other PEG tools:

Feature	TinyPEG	PEG.js	TatSu	Parsec
Language	Python	JavaScript	Python	Haskell
Approach	Class-based	Grammar files	Grammar files	Combinator
Memoization	Basic	Full packrat	Optional	Manual
Error Messages	Position-based	Good	Excellent	Good
AST Building	Manual	Automatic	Automatic	Manual
Learning Curve	Low	Medium	Medium	High

A.5.1 TinyPEG Advantages

- **Programmatic**: Define grammars using Python classes
- **Lightweight**: Minimal dependencies, easy to embed
- **Extensible**: Easy to customize parsing behavior
- **Educational**: Clear, readable implementation

A.5.2 When to Use TinyPEG

TinyPEG is ideal for: - Learning PEG concepts and implementation - Building domain-specific languages - Prototyping parsers quickly - Educational projects and tutorials - Small to medium parsing tasks

A.6 Complete API Reference

A.6.1 Core Classes

```python
```

Import all classes

from src.peg import (PEGParser, # Main parser class Rule, # Grammar rule definition GrammarNode, # Grammar container Reference, # Rule reference Sequence, # Sequential matching Choice, # Alternative matching ZeroOrMore, # Zero or more repetition OneOrMore, # One or more repetition Optional, # Optional matching AndPredicate, # Positive lookahead NotPredicate, # Negative lookahead Literal, # Exact string matching Regex, # Pattern matching ParseError, # Parsing exceptions ParserContext # Parsing state) ```

A.6.2 Usage Pattern

```python

# 1. Define grammar

grammar = GrammarNode( name="MyGrammar", rules=[ Rule("Start", Reference("Expression")), # ... more rules ] )

# 2. Create parser

parser = PEGParser() parser.grammar = grammar

# 3. Parse input

try: result = parser.parse("input text") print("Success:", result) except ParseError as e: print("Error:", e) ```

## Summary

TinyPEG provides a clean, educational implementation of Parsing Expression Grammars in Python. Unlike traditional PEG tools that use grammar files, TinyPEG uses Python classes to define grammars programmatically, making it ideal for learning, prototyping, and building domain-specific languages.
```

Key features of our implementation: - **Class-based grammar definition** for maximum flexibility - **Automatic whitespace handling** for convenience - **Basic memoization** for performance - **Clear error reporting** with position information - **Extensible architecture** for custom parsing behavior

This appendix has covered the complete TinyPEG API, common patterns, implementation details, and usage examples. With this information, you should be able to effectively use TinyPEG for your parsing projects.