Chapter 1: Understanding PEG Parsers

1.1 Introduction to Parsing Expression Grammars

Parsing Expression Grammars (PEGs) were formally introduced by Bryan Ford in 2004 as an alternative to Context-Free Grammars (CFGs) for describing syntax. PEGs provide a formal foundation for recursive descent parsers with backtracking, which have been used in practice for decades.

The key insight of PEGs is that they view parsing as a recognition process rather than a generative one. Instead of describing all possible strings that could be generated by a grammar (as CFGs do), PEGs describe a procedure for recognizing whether a string belongs to a language.

Key Characteristics of PEGs:

1. **Unambiguous**: PEGs always yield at most one valid parse tree for any input string
2. **Ordered Choice**: When multiple alternatives could match, the first matching one is chosen
3. **Unlimited Lookahead**: PEGs can use arbitrary lookahead to make parsing decisions
4. **Integrated Lexical and Syntactic Analysis**: No separate lexer/tokenizer is needed
5. **Recognition-Based**: Focused on recognizing valid inputs rather than generating them

1.2 PEG vs. Context-Free Grammars

While PEGs and CFGs may appear similar in notation, they differ fundamentally in semantics:

Aspect	Context-Free Grammars	Parsing Expression Grammars
Ambiguity	Can be ambiguous	Always unambiguous
Choice Operator	Unordered (any production can be chosen)	Ordered (first matching rule wins)
Implementation	Various algorithms

(LL, LR, etc.) | Recursive descent with backtracking | | Expressiveness | Can express some patterns PEGs cannot | Can express some patterns CFGs cannot | | Parsing Complexity | Varies by algorithm (O(n) to O(n³)) | O(n) with memoization (Packrat parsing) | | Whitespace Handling | Requires separate lexer | Can be handled directly in grammar |

The ordered choice operator (`/` in standard PEG notation, often represented as alternatives in implementations) is perhaps the most significant difference. In a CFG, the expression `A → B | C` means "A can be either B or C," with no preference specified. In a PEG, the equivalent expression `A ← B / C` means "Try to match A as B; if that fails, try to match it as C."

1.3 PEG Operators and Notation

PEGs use a set of operators to define parsing expressions:

| Operator | Name | Description | Example | |----------|------|-------------|---------| | `e1 e2` | Sequence | Match e1 followed by e2 | `"if" Condition Block` | | `e1 / e2` | Ordered Choice | Try e1; if it fails, try e2 | `Addition / Multiplication / Primary` | | `e*` | Zero-or-More | Match e zero or more times | `Statement*` | | `e+` | One-or-More | Match e one or more times | `Digit+` | | `e?` | Optional | Match e or nothing | `"else" Block?` | | &e | And-Predicate | Succeed if e matches but don't consume input | `&[a-z]` | | `!e` | Not-Predicate | Succeed if e doesn't match and don't consume input | `![0-9]` | | `(e)` | Grouping | Group expressions | `("+" / "-") Term` |

In our TinyPEG implementation, these operators are represented through Python classes rather than this specific syntax. For example:

- **Sequence** → `Sequence(expr1, expr2)`
- **Ordered Choice** → `Choice(expr1, expr2)`
- **Zero-or-More** → `ZeroOrMore(expr)`
- **One-or-More** → `OneOrMore(expr)`
- **Optional** → `Optional(expr)`
- **And-Predicate** → `AndPredicate(expr)`
- **Not-Predicate** → `NotPredicate(expr)`

This class-based approach provides better integration with Python and enables more sophisticated parsing behaviors.

1.4 Recursive Descent Parsing

PEGs are naturally implemented using recursive descent parsing, a top-down parsing technique where each non-terminal in the grammar corresponds to a function in the parser.

The basic algorithm for a recursive descent PEG parser is:

1. For each grammar rule, create a parsing function
2. The function attempts to match its rule against the current input position
3. If successful, it returns the matched result and advances the input position
4. If unsuccessful, it backtracks to the original position and reports failure
5. For ordered choice, try each alternative in order until one succeeds or all fail

Here's a simplified example of how a recursive descent parser might handle a simple arithmetic expression grammar:

```python def parse_expression(input, pos): # Try to parse as a term result, new_pos = parse_term(input, pos) if result is not None: return result, new_pos return None, pos # Backtrack if unsuccessful

def parse_term(input, pos): # Try to parse as a factor result, new_pos = parse_factor(input, pos) if result is not None: return result, new_pos return None, pos # Backtrack if unsuccessful

def parse_factor(input, pos): # Try to parse as a number if pos < len(input) and input[pos].isdigit(): # Match one or more digits start = pos while pos < len(input) and input[pos].isdigit(): pos += 1 return int(input[start:pos]), pos return None, pos # Backtrack if unsuccessful ```

## 1.5 Packrat Parsing and Memoization

A naive recursive descent implementation of PEG parsing can have exponential time complexity in the worst case due to backtracking. This is where Packrat parsing comes in.

Packrat parsing is a technique that applies memoization to recursive descent parsing. By caching the results of parsing functions at each input position, it ensures that no parsing function is called more than once at any given position, resulting in linear time complexity.
```

The basic idea is:

1. Before attempting to parse a rule at a position, check if we've already tried this combination
2. If we have, return the cached result
3. Otherwise, perform the parsing and cache the result before returning it

Here's how our TinyPEG implementation incorporates memoization:

```python class PEGParser: def **init**(self): self.grammar = None self.rule_cache = {} # Memoization cache

```
def _parse_rule(self, rule, ctx):
    """Parse a rule with memoization."""
    # Check if we've already parsed this rule at this position
    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

    # Save position for backtracking
    start_pos = ctx.pos

    try:
        # Parse the rule's expression
        result = self._parse_expression(rule.expr, ctx)

        # Cache the successful result
        self.rule_cache[cache_key] = (result, ctx.pos)
        return result
    except ParseError:
        # Backtrack on failure
        ctx.pos = start_pos
        raise
```

```

Our TinyPEG implementation automatically incorporates memoization in the `PEGParser` class, ensuring efficient parsing without requiring manual cache management.

1.6 TinyPEG Implementation Preview

Our TinyPEG library implements these PEG concepts using a clean, object-oriented Python design:

- **Class-based Grammar Definition**: Grammars are defined using Python classes rather than text files
- **Automatic Memoization**: Built-in packrat parsing for optimal performance
- **Integrated Whitespace Handling**: Automatic whitespace management between tokens
- **Comprehensive Error Reporting**: Detailed error messages with position information
- **Extensible Architecture**: Easy to extend with custom parsing behaviors

Here's a preview of how a simple grammar looks in TinyPEG:

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

# Define a simple number parser

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

# Create and use the parser

parser = PEGParser() parser.grammar = grammar result = parser.parse("42") # Returns "42" ```

In the next chapter, we'll explore the complete architecture and components of the TinyPEG library, seeing how these concepts are implemented in detail.
```