TinyPEG: A Parsing Expression Grammar Library

Welcome to the documentation for TinyPEG, a lightweight and easy-to-use Parsing Expression Grammar (PEG) library implemented in Python.

What is TinyPEG?

TinyPEG is a parser library that allows you to define grammars using Parsing Expression Grammar notation and use them to parse and analyze text. It's designed to be:

- **Simple**: Easy to understand and use
- **Flexible**: Adaptable to various parsing needs
- **Educational**: A great tool for learning about parsers and language design

Getting Started

Quick Example

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

# Create a simple number parser

class SimpleParser(PEGParser): def **init**(self): super().**init**()
```

```
    # Define a simple grammar for numbers
    self.grammar = GrammarNode(
        name="Simple",
        rules=[
            Rule("Number", Regex("[0-9]+"))
        ]
    )
```

Use the parser

parser = SimpleParser() result = parser.parse("42") print(f"Parsed result: {result}") # Output: Parsed result: 42 ```

For a complete working example, see the Simple Fixed Test in the examples directory.

License

Preface

The Art and Science of Parsing

Parsing is a fundamental concept in computer science that bridges the gap between human-readable text and machine-processable structures. At its core, parsing is the process of analyzing a sequence of symbols (like text) according to the rules of a formal grammar.

Whether you're building a programming language, a configuration file parser, or a domain-specific language for a specialized application, understanding parsing techniques is essential. This knowledge empowers you to create tools that can interpret and process structured text in meaningful ways.

Why Parsing Expression Grammars?

Parsing Expression Grammars (PEGs) represent a powerful approach to syntax analysis that combines the expressiveness of context-free grammars with the efficiency and predictability of recursive descent parsing.

Unlike traditional context-free grammars that can be ambiguous, PEGs are inherently unambiguous due to their ordered choice operator. This means that when multiple parsing rules could match, a PEG parser will always choose the first matching rule, eliminating ambiguity.

Key advantages of PEGs include:

- **Unambiguous parsing**: The ordered choice operator ensures a single valid parse for any input
- **Integrated lexical and syntactic analysis**: No separate lexer/tokenizer is needed
- **Powerful pattern matching**: Including lookahead and negative lookahead assertions
- **Intuitive notation**: Grammar rules that closely resemble EBNF notation
- **Recursive descent implementation**: Straightforward to implement and understand

About This Documentation

This documentation serves as both a reference guide and a tutorial for the TinyPEG library. We'll explore:

1. The theoretical foundations of Parsing Expression Grammars
2. The architecture and components of the TinyPEG library
3. Practical examples of building parsers for various use cases
4. A step-by-step guide to creating a complete tiny programming language

Current Status

The TinyPEG library is being actively developed. The current version includes:

- A fixed implementation of the core parsing components in `fixed_core.py` and `fixed_parsers.py`
- Working examples in the `examples` directory, including a standalone TinyCL implementation
- Comprehensive documentation and tutorials

For the most up-to-date information about the project's status, see the Project Status document.

Whether you're a student learning about parsing for the first time, a developer looking to build a custom parser, or an educator teaching language design, we hope this documentation provides valuable insights and practical guidance.

Let's embark on this journey into the fascinating world of parsing and language design!

"Programming languages are how programmers express and communicate ideas — and, occasionally, frustrations." — Simon Peyton Jones

Table of Contents

Chapter 3: Building Your First Parser

Chapter 4: Example Parsers and Applications

Chapter 5: Creating TinyCL - A Complete Programming Language

Appendix B: Testing Framework

Appendix C: TinyCL Language Reference

- Summary

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.
```

Chapter 2: TinyPEG Library Overview

2.1 Architecture and Design Philosophy

The TinyPEG library is designed with simplicity, flexibility, and educational value in mind. Its architecture follows these key principles:

1. **Separation of Concerns**: The library separates the core parsing mechanisms from specific grammar implementations, allowing users to define their own grammars.

2. **Object-Oriented Design**: Grammar elements are represented as objects, making the library extensible and maintainable.

3. **Composability**: Grammar rules can be composed to create complex parsers from simple building blocks.

4. **Minimal Dependencies**: The library relies only on Python's standard library, making it easy to use in any environment.

The library is organized into three main modules:

- **core.py**: Contains the fundamental classes for parsing, including Expression, Reference, ParserContext, and ParseError.
- **parsers.py**: Provides the PEGParser and Rule classes for building parsers.
- **syntax_tree.py**: Defines the GrammarNode and DebugVisitor classes for representing and traversing syntax trees.

Let's explore each of these components in detail.

2.2 Core Components

2.2.1 Expression Class

The `Expression` class is the abstract base class for all parsing expressions in the TinyPEG library. It defines the interface that all concrete expression types must implement.

`python class Expression: """Base class for expressions.""" def parse(self, ctx): raise NotImplementedError("Subclasses should implement this method.")`

The `parse` method takes a `ParserContext` object and attempts to match the expression against the input at the current position. Subclasses override this method to implement specific parsing behaviors.

2.2.2 Reference Class

The `Reference` class is a crucial component that allows grammar rules to reference other rules by name. This enables recursive grammar definitions and modular rule composition.

```python class Reference(Expression): """Reference to another rule or pattern in the grammar."""

```
def __init__(self, name):
    """
    Initialize a reference to another rule or pattern.

    Args:
        name: The name of the rule or pattern being referenced
    """
    self.name = name

def parse(self, ctx):
    # This will be implemented by the grammar system
    # that resolves references to actual rules
    raise NotImplementedError("Reference parsing is handled by the gramm
```

```

When a grammar is being parsed, references are resolved to their corresponding rules, allowing for a clean separation between rule definition and rule usage.

2.2.3 ParserContext Class

The `ParserContext` class maintains the state of the parsing process, including the input text and the current position.

```python class ParserContext: """Context class to hold the parsing state."""

```
def __init__(self, text):
    self.text = text
    self.pos = 0

def eof(self):
    return self.pos >= len(self.text)

def peek(self):
    return self.text[self.pos] if not self.eof() else None

def consume(self):
    """Advance the position by one character."""
    char = self.peek()
    self.pos += 1
    return char
```

```

This class provides methods for checking if the end of input has been reached (`eof`), looking at the current character without consuming it (`peek`), and advancing the position (`consume`).

2.2.4 ParseError Class

The `ParseError` class is a custom exception used to report parsing errors.

`python class ParseError(Exception): """Custom error for parsing issues."""  pass`

This allows the library to distinguish between parsing errors and other types of exceptions that might occur during the parsing process.

2.3 Parser Components

2.3.1 PEGParser Class

The `PEGParser` class is the base class for all parsers built with TinyPEG. It provides the infrastructure for parsing input according to a grammar.

```python class PEGParser: """Parser class that uses PEG grammar rules."""

```
def __init__(self):
    self.grammar = None # To be defined by subclasses

def parse(self, text: str):
    """Parse input text according to the grammar."""
    if self.grammar is None:
        raise ValueError("Grammar not defined")

    # Create parser context
    ctx = ParserContext(text)

    # Apply the start rule (first rule in grammar)
    if self.grammar.rules:
        result = self._parse_expression(self.grammar.rules[0].expr, ctx)
        if not ctx.eof():
            raise ParseError(f"Unexpected input at position {ctx.pos}")
        return result

    raise ParseError("No rules defined in grammar")

def _parse_expression(self, expr, ctx):
    """Parse an expression with the given context."""
    return expr.parse(ctx)
```

```

In a complete implementation, the `parse` method would: 1. Create a `ParserContext` from the input text 2. Resolve references in the grammar 3. Apply the grammar's start rule to the context 4. Return the resulting parse tree or AST

2.3.2 Rule Class

The `Rule` class represents a named rule in a grammar. It associates a name with an expression that defines what the rule matches.

```python class Rule(Expression): """Represents a grammar rule."""

```
def __init__(self, name, expr):
    self.name = name
    self.expr = expr

def parse(self, ctx):
    return self.expr.parse(ctx)
```

```

Rules are the building blocks of grammars. They can reference other rules, creating a network of dependencies that defines the grammar's structure.

2.4 Syntax Tree Components

2.4.1 GrammarNode Class

The `GrammarNode` class represents a node in the grammar's syntax tree. It contains a name and a list of rules.

```python class GrammarNode: """Node in the grammar syntax tree."""

```
def __init__(self, name, rules):
    self.name = name
    self.rules = rules

def accept(self, visitor):
    """Accept a visitor to traverse the grammar tree."""
    visitor.visit_grammar(self)
    for rule in self.rules:
        visitor.visit_rule(rule)
```

```

The `accept` method implements the Visitor pattern, allowing various operations to be performed on the syntax tree without modifying its structure.

2.4.2 DebugVisitor Class

The `DebugVisitor` class is a utility for debugging grammars. It traverses a grammar tree and prints information about each node.

```python class DebugVisitor: """Visitor for debugging grammar trees."""

```
def visit_grammar(self, grammar):
    print(f"Grammar: {grammar.name}")

def visit_rule(self, rule):
    print(f" Rule: {rule.name}")
```

```

This is useful for understanding the structure of a grammar and diagnosing issues with rule definitions.

2.5 Testing the Library

To ensure the reliability of the TinyPEG library, we need a comprehensive suite of tests. Let's create tests for each component, starting with the core classes.

Testing 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() ```

Testing the Parser Components

```python

# tests/test_parsers.py

import unittest from src.peg.parsers import PEGParser, Rule from src.peg.core import Expression, ParserContext

class MockExpression(Expression): def **init**(self, result=True): self.result = result self.called = False

```
def parse(self, ctx):
    self.called = True
    return self.result
```

class TestRule(unittest.TestCase): def test_initialization(self): expr = MockExpression() rule = Rule("TestRule", expr) self.assertEqual(rule.name, "TestRule") self.assertEqual(rule.expr, expr)

```
def test_parse(self):
    expr = MockExpression(True)
    rule = Rule("TestRule", expr)
    ctx = ParserContext("test")
    result = rule.parse(ctx)
    self.assertTrue(expr.called)
    self.assertTrue(result)
```

class TestPEGParser(unittest.TestCase): def test_initialization(self): parser = PEGParser() # Basic initialization test self.assertIsInstance(parser, PEGParser)

```
def test_parse(self):
    parser = PEGParser()
    result = parser.parse("test input")
```
```

```
    # For now, just check that it returns something
    self.assertIsNotNone(result)
```

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

Testing the Syntax Tree Components

```python

# tests/test_syntax_tree.py

import unittest from src.peg.syntax_tree import GrammarNode, DebugVisitor from src.peg.parsers import Rule from src.peg.core import Expression

class MockRule(Rule): def **init**(self, name): super().**init**(name, None)

class MockVisitor: def **init**(self): self.visited_grammars = [] self.visited_rules = []

```
def visit_grammar(self, grammar):
    self.visited_grammars.append(grammar)

def visit_rule(self, rule):
    self.visited_rules.append(rule)
```

class TestGrammarNode(unittest.TestCase): def test_initialization(self): rules = [MockRule("Rule1"), MockRule("Rule2")] grammar = GrammarNode("TestGrammar", rules) self.assertEqual(grammar.name, "TestGrammar") self.assertEqual(grammar.rules, rules)

```
def test_accept(self):
    rules = [MockRule("Rule1"), MockRule("Rule2")]
    grammar = GrammarNode("TestGrammar", rules)
    visitor = MockVisitor()
    grammar.accept(visitor)
    self.assertEqual(len(visitor.visited_grammars), 1)
    self.assertEqual(visitor.visited_grammars[0], grammar)
    self.assertEqual(len(visitor.visited_rules), 2)
```
```

```
    self.assertEqual(visitor.visited_rules[0], rules[0])
    self.assertEqual(visitor.visited_rules[1], rules[1])
```

class TestDebugVisitor(unittest.TestCase): def test_visit_grammar(self): # This is more of a functional test than a unit test # since it just prints to stdout visitor = DebugVisitor() grammar = GrammarNode("TestGrammar", []) visitor.visit_grammar(grammar) # No assertion, just make sure it doesn't raise an exception

```
def test_visit_rule(self):
    visitor = DebugVisitor()
    rule = MockRule("TestRule")
    visitor.visit_rule(rule)
    # No assertion, just make sure it doesn't raise an exception
```

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

These tests provide a foundation for ensuring the correctness of the TinyPEG library. As the library evolves, additional tests should be added to cover new functionality and edge cases.

In the next chapter, we'll build our first parser using the TinyPEG library, putting these components to practical use.

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"
]
```

```
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.
```

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.

Chapter 5: Creating TinyCL - A Complete Programming Language

In this chapter, we'll explore the complete implementation of TinyCL (Tiny C-Like Language), a fully-functional programming language built using our PEG parser library. TinyCL demonstrates how to create a production-quality language with parser, interpreter, and multi-target compiler.

5.1 TinyCL Language Overview

TinyCL is a comprehensive programming language that showcases all aspects of language implementation:

5.1.1 Language Features

TinyCL supports the following modern language features:

1. **Variables and Constants**: `var x = 10;` and `const PI = 3;`
2. **Full Arithmetic**: +, -, *, / with proper precedence
3. **Logical Operations**: &&, `||`, `!` for boolean logic
4. **Comparison Operations**: ==, !=, <, >, <=, >=
5. **Arrays**: `[1, 2, 3]` and array access `arr[0]`
6. **Characters**: `'A'` for single characters
7. **Control Structures**: If-else statements and while loops
8. **Functions**: Declaration, calls, and return values
9. **Comments**: `# This is a comment`
10. **String Operations**: Concatenation and manipulation

Here's an example of a TinyCL program:

```
```

Calculate factorial with functions and arrays

var numbers = [5, 6, 7]; const MAX = 10;

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

var i = 0; while (i < 3) { var num = numbers[i]; var result = factorial(num); print("Factorial of " + num + " is " + result); i = i + 1; } ```

5.1.2 Complete Grammar Specification

Here's the complete EBNF grammar for TinyCL:

```ebnf Program ::= Statements

Statements ::= Statement*

Statement ::= FunctionDecl | VariableDecl | ConstantDecl | "if" "(" Expression ")" Block ( "else" Block )? | "while" "(" Expression ")" Block | "print" "(" Expression ")" ";" | "return" Expression? ";" | Id "=" Expression ";" | Id "(" Arguments? ")" ";" | Block | Comment

FunctionDecl ::= "func" Id "(" Parameters? ")" Block VariableDecl ::= "var" Id "=" Expression ";" ConstantDecl ::= "const" Id "=" Expression ";"

Parameters ::= Id ( "," Id ) *Arguments ::= Expression ( "," Expression )*

Block ::= "{" Statements? "}"

# Expression hierarchy with proper precedence

Expression ::= LogicalOr LogicalOr ::= LogicalAnd ( "||" LogicalAnd ) *LogicalAnd ::= Equality ( "&&" Equality )* Equality ::= Comparison ( ( "!=" | "==" ) Comparison ) *Comparison ::= Term ( ( "<=" | ">=" | "<" | ">" ) Term )* Term ::= Factor ( ( "+" | "-" )
```

Factor) *Factor ::= Unary (("*" | "/") Unary) Unary ::= ("!" | "-")? Postfix Postfix ::= Primary ("[" Expression "]")*

Primary ::= "(" Expression ")" | Id "(" Arguments? ")" | "[" Arguments? "]" | Id | Number | String | Character | "true" | "false"

Literals

String ::= "" StringChar* "" StringChar ::= [#x20-#x21] | [#x23-#x5B] | [#x5D-#x7E] | "\" EscapeChar EscapeChar ::= "" | "'" | "\" | "n" | "r" | "t" | "0" | "b" | "f" | "v" | "l"

Character ::= "'" CharChar "'" CharChar ::= [#x20-#x26] | [#x28-#x5B] | [#x5D-#x7E] | "\" EscapeChar

Id ::= Letter (Letter | Digit | "_")* Number ::= Digit+ Letter ::= [a-zA-Z] Digit ::= [0-9]

Comment ::= "#" [^\n]* ```

This comprehensive grammar defines all the syntax of TinyCL programs, including modern features like arrays, logical operators, and proper expression precedence.

5.2 Complete Implementation Overview

The TinyCL implementation consists of three main components:

1. **Parser** (`src/tinycl/parser.py`) - Converts source code to AST
2. **Interpreter** (`src/tinycl/interpreter.py`) - Executes TinyCL programs directly
3. **Compiler** (`src/tinycl/compiler.py`) - Generates Python and C code

5.2.1 Parser Implementation

The TinyCL parser uses our PEG library to define a complete grammar:

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

class TinyCLParser(PEGParser): """Parser for the TinyCL language - Complete Implementation."""
```

```
def __init__(self):
    super().__init__()

    # Define complete grammar for TinyCL
    self.grammar = GrammarNode(
        name="TinyCL",
        rules=[
            # Program structure
            Rule("Program", ZeroOrMore(Reference("Statement"))),

            # Statements - All implemented features
            Rule("Statement", Choice(
                Reference("FunctionDecl"),
                Reference("VariableDecl"),
                Reference("ConstantDecl"),
                Reference("IfStatement"),
                Reference("WhileStatement"),
                Reference("PrintStatement"),
                Reference("ReturnStatement"),
                Reference("AssignmentStatement"),
                Reference("ExpressionStatement"),
                Reference("Block"),
                Reference("Comment")
            )),

            # Function declaration - Fully implemented
            Rule("FunctionDecl", Sequence(
                Literal("func"),
                Reference("Identifier"),
                Literal("("),
                Optional(Reference("Parameters")),
                Literal(")"),
                Reference("Block")
            )),

            # Parameters - Supports multiple parameters
            Rule("Parameters", Sequence(
```

```
            Reference("Identifier"),
            ZeroOrMore(Sequence(
                Literal(","),
                Reference("Identifier")
            ))
        )),

        # Variable declaration - Complete implementation
        Rule("VariableDecl", Sequence(
            Literal("var"),
            Reference("Identifier"),
            Literal("="),
            Reference("Expression"),
            Literal(";")
        )),

        # Constant declaration - Complete implementation
        Rule("ConstantDecl", Sequence(
            Literal("const"),
            Reference("Identifier"),
            Literal("="),
            Reference("Expression"),
            Literal(";")
        )),

        # If statement with else support
        Rule("IfStatement", Sequence(
            Literal("if"),
            Literal("("),
            Reference("Expression"),
            Literal(")"),
            Reference("Block"),
            Optional(Sequence(
                Literal("else"),
                Reference("Block")
            ))
        )),
```

```
        # While statement - Complete implementation
        Rule("WhileStatement", Sequence(
            Literal("while"),
            Literal("("),
            Reference("Expression"),
            Literal(")"),
            Reference("Block")
        )),

        # Print statement - Built-in function
        Rule("PrintStatement", Sequence(
            Literal("print"),
            Literal("("),
            Reference("Expression"),
            Literal(")"),
            Literal(";")
        )),

        # Return statement - Complete implementation
        Rule("ReturnStatement", Sequence(
            Literal("return"),
            Optional(Reference("Expression")),
            Literal(";")
        )),

        # Block - Supports nested statements
        Rule("Block", Sequence(
            Literal("{"),
            ZeroOrMore(Reference("Statement")),
            Literal("}")
        )),

        # Expression hierarchy with proper precedence
        Rule("Expression", Reference("LogicalOr")),

        # Logical OR - Complete implementation
```

```
            Rule("LogicalOr", Sequence(
                Reference("LogicalAnd"),
                ZeroOrMore(Sequence(
                    Literal("||"),
                    Reference("LogicalAnd")
                ))
            )),

            # Logical AND - Complete implementation
            Rule("LogicalAnd", Sequence(
                Reference("Equality"),
                ZeroOrMore(Sequence(
                    Literal("&&"),
                    Reference("Equality")
                ))
            )),

            # Equality operators
            Rule("Equality", Sequence(
                Reference("Comparison"),
                ZeroOrMore(Sequence(
                    Choice(
                        Literal("!="),
                        Literal("==")
                    ),
                    Reference("Comparison")
                ))
            )),

            # Comparison operators
            Rule("Comparison", Sequence(
                Reference("Term"),
                ZeroOrMore(Sequence(
                    Choice(
                        Literal("<="),
                        Literal(">="),
                        Literal("<"),
```

```
                Literal(">")
            ),
            Reference("Term")
        ))
    )),

    # Arithmetic: Addition and Subtraction
    Rule("Term", Sequence(
        Reference("Factor"),
        ZeroOrMore(Sequence(
            Choice(
                Literal("+"),
                Literal("-")
            ),
            Reference("Factor")
        ))
    )),

    # Arithmetic: Multiplication and Division
    Rule("Factor", Sequence(
        Reference("Unary"),
        ZeroOrMore(Sequence(
            Choice(
                Literal("*"),
                Literal("/")
            ),
            Reference("Unary")
        ))
    )),

    # Unary operators
    Rule("Unary", Choice(
        Sequence(
            Choice(
                Literal("!"),
                Literal("-")
            ),
```

```
                Reference("Unary")
            ),
            Reference("Postfix")
        )),

        # Postfix: Array access
        Rule("Postfix", Sequence(
            Reference("Primary"),
            ZeroOrMore(Sequence(
                Literal("["),
                Reference("Expression"),
                Literal("]")
            ))
        )),

        # Primary expressions
        Rule("Primary", Choice(
            Sequence(
                Literal("("),
                Reference("Expression"),
                Literal(")")
            ),
            Sequence(
                Reference("Identifier"),
                Literal("("),
                Optional(Reference("Arguments")),
                Literal(")")
            ),
            Sequence(
                Literal("["),
                Optional(Reference("Arguments")),
                Literal("]")
            ),
            Reference("Identifier"),
            Reference("Number"),
            Reference("String"),
            Reference("Character"),
```

```
                Literal("true"),
                Literal("false")
            )),

            # Arguments
            Rule("Arguments", Sequence(
                Reference("Expression"),
                ZeroOrMore(Sequence(
                    Literal(","),
                    Reference("Expression")
                ))
            )),

            # Terminals - All data types
            Rule("Number", Regex("[0-9]+")),
            Rule("String", Regex("\"[^\"]*\"")),
            Rule("Character", Regex("'[^']*'")),
            Rule("Identifier", Regex("[a-zA-Z_][a-zA-Z0-9_]*")),
            Rule("Comment", Regex("#[^\n]*"))
        ]
    )

def skip_whitespace(self, ctx):
    """Skip whitespace and comments."""
    while not ctx.eof():
        if ctx.peek() in " \t\n\r":
            ctx.consume()
            continue
        if ctx.peek() == '#':
            while not ctx.eof() and ctx.peek() != '\n':
                ctx.consume()
            continue
        break

def _parse_expression(self, expr, ctx):
    """Override to handle whitespace between tokens."""
    self.skip_whitespace(ctx)
```

```
    result = super()._parse_expression(expr, ctx)
    self.skip_whitespace(ctx)
    return result

def parse(self, text):
    """Parse a TinyCL program and build an AST."""
    result = super().parse(text)
    return self._build_ast(result)

def _build_ast(self, parse_result):
    """Build an AST from the parse result."""
    if parse_result is None:
        return None
    return ProgramNode(self._build_statements(parse_result))

# ... (AST building methods implemented in actual parser)
```

```

This parser defines the grammar for TinyCL and implements the `parse` method to parse TinyCL programs. However, it doesn't yet build an AST or interpret the programs.

### 5.2.1 Lexical Elements

Let's enhance our parser to handle lexical elements like whitespace and comments properly:

```python

# Add to TinyCLParser class

def skip_whitespace(self, ctx): """Skip whitespace and comments.""" while not ctx.eof(): # Skip whitespace if ctx.peek().isspace(): ctx.consume() continue

```
    # Skip comments
    if ctx.peek() == '#':
        while not ctx.eof() and ctx.peek() != '\n':
```

```
            ctx.consume()
        continue

    # No more whitespace or comments to skip
    break
```

```

### 5.2.2 Expressions

Now, let's implement the parsing of expressions:

```python

# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

# Add to TinyCLParser class

def parse_expression(self, ctx): """Parse an expression.""" self.skip_whitespace(ctx)

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

# Parse any following +/- operations
while not ctx.eof():
    self.skip_whitespace(ctx)

    # Try to parse an operator
    op_pos = ctx.pos
    if ctx.peek() == '+' or ctx.peek() == '-':
        op = ctx.consume()
```

```
        self.skip_whitespace(ctx)

        # 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 term.""" self.skip_whitespace(ctx)

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

# Parse any following */÷ operations
while not ctx.eof():
    self.skip_whitespace(ctx)

    # Try to parse an operator
    op_pos = ctx.pos
    if ctx.peek() == '*' or ctx.peek() == '/':
        op = ctx.consume()

        self.skip_whitespace(ctx)

        # 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): """Parse a factor.""" self.skip_whitespace(ctx)

```
if ctx.eof():
    return None

# Try to parse a number
if ctx.peek().isdigit():
    return self.parse_number(ctx)

# Try to parse a string
if ctx.peek() == '"':
    return self.parse_string(ctx)

# Try to parse an identifier
if ctx.peek().isalpha() or ctx.peek() == '_':
    return self.parse_identifier(ctx)

# Try to parse a parenthesized expression
if ctx.peek() == '(':
    ctx.consume()  # Consume '('

    self.skip_whitespace(ctx)

    # Parse the inner expression
    expr = self.parse_expression(ctx)
```

```
    if expr is None:
        return None

    self.skip_whitespace(ctx)

    # Expect a closing parenthesis
    if ctx.eof() or ctx.peek() != ')':
        return None
    ctx.consume()  # Consume ')'

    return expr

return None
```

def parse_number(self, ctx): """Parse a number.""" start = ctx.pos while not ctx.eof() and ctx.peek().isdigit(): ctx.consume()

```
if start == ctx.pos:
    return None

return NumberNode(int(ctx.text[start:ctx.pos]))
```

def parse_string(self, ctx): """Parse a string.""" if ctx.eof() or ctx.peek() != "": return None

```
ctx.consume()  # Consume opening quote

start = ctx.pos
while not ctx.eof() and ctx.peek() != '"':
    ctx.consume()

if ctx.eof():
    return None  # Unterminated string

value = ctx.text[start:ctx.pos]
ctx.consume()  # Consume closing quote
```

```
        return StringNode(value)
```

def parse_identifier(self, ctx): """Parse an identifier.""" if ctx.eof() or not (ctx.peek().isalpha() or ctx.peek() == '_'): return None

```
        start = ctx.pos
        ctx.consume()  # Consume first character

        while not ctx.eof() and (ctx.peek().isalnum() or ctx.peek() == '_'):
            ctx.consume()

        name = ctx.text[start:ctx.pos]
        return IdentifierNode(name)
```

```

#### 5.2.3 Statements

Now, let's implement the parsing of statements:

```python
# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

# Add to TinyCLParser class
```

def parse_statement(self, ctx): """Parse a statement.""" self.skip_whitespace(ctx)

```
        if ctx.eof():
            return None

        # Try to parse a var statement
```
```

```
if ctx.pos + 3 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+3] == "var"
    return self.parse_var_statement(ctx)

# Try to parse an if statement
if ctx.pos + 2 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+2] == "if":
    return self.parse_if_statement(ctx)

# Try to parse a while statement
if ctx.pos + 5 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+5] == "whil
    return self.parse_while_statement(ctx)

# Try to parse a print statement
if ctx.pos + 5 <= len(ctx.text) and ctx.text[ctx.pos:ctx.pos+5] == "prin
    return self.parse_print_statement(ctx)

# Try to parse a block
if ctx.peek() == '{':
    return self.parse_block(ctx)

# Try to parse an assignment statement
return self.parse_assignment_statement(ctx)
```

def parse_var_statement(self, ctx): """Parse a var statement.""" start_pos = ctx.pos

```
# Expect "var"
if ctx.pos + 3 > len(ctx.text) or ctx.text[ctx.pos:ctx.pos+3] != "var":
    return None
ctx.pos += 3

self.skip_whitespace(ctx)

# Parse identifier
identifier = self.parse_identifier(ctx)
if identifier is None:
    ctx.pos = start_pos
    return None
```

```
self.skip_whitespace(ctx)

# Expect "="
if ctx.eof() or ctx.peek() != '=':
    ctx.pos = start_pos
    return None
ctx.consume()

self.skip_whitespace(ctx)

# Parse expression
expression = self.parse_expression(ctx)
if expression is None:
    ctx.pos = start_pos
    return None

self.skip_whitespace(ctx)

# Expect ";"
if ctx.eof() or ctx.peek() != ';':
    ctx.pos = start_pos
    return None
ctx.consume()

return VariableDeclNode(identifier, expression)
```

Similar implementations for parse_assignment_statement, parse_if_statement,

parse_while_statement, parse_print_statement, and parse_block

```

#### 5.2.4 Program Structure

Finally, let's implement the parsing of the overall program structure:

```python

# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

# Add to TinyCLParser class

def parse_program(self, ctx): """Parse a program.""" statements = []

```
while not ctx.eof():
    self.skip_whitespace(ctx)

    if ctx.eof():
        break
```

```
    statement = self.parse_statement(ctx)
    if statement is None:
        break

    statements.append(statement)

return ProgramNode(statements)
```

def parse(self, text: str): """Parse a TinyCL program.""" print(f"Parsing TinyCL program:\n{text}")

```
# Create a ParserContext
ctx = ParserContext(text)

# Parse the program
program = self.parse_program(ctx)

# Skip any trailing whitespace
self.skip_whitespace(ctx)

# Check if we consumed all input
if ctx.eof():
    return program
else:
    raise ParseError(f"Unexpected input at position {ctx.pos}: '{ctx.tex
```

```

## 5.3 Building the Abstract Syntax Tree

Now that we have the parsing functions, we need to define the AST node classes:

```python

# Note: In practice, these AST classes are defined in src/tinycl/ast.py

# and imported with: from src.tinycl.ast import *

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

class ProgramNode(ASTNode): """AST node for a program.""" def **init**(self, statements): self.statements = statements

class StatementNode(ASTNode): """Base class for statement nodes.""" pass

class VariableDeclNode(StatementNode): """AST node for a variable declaration.""" def **init**(self, identifier, expression): self.identifier = identifier self.expression = expression

class AssignStatementNode(StatementNode): """AST node for an assignment statement.""" def **init**(self, identifier, expression): self.identifier = identifier self.expression = expression

class IfStatementNode(StatementNode): """AST node for an if statement.""" def **init**(self, condition, then_statement, else_statement=None): self.condition = condition self.then_statement = then_statement self.else_statement = else_statement

class WhileStatementNode(StatementNode): """AST node for a while statement.""" def **init**(self, condition, body): self.condition = condition self.body = body

class PrintStatementNode(StatementNode): """AST node for a print statement.""" def **init**(self, expression): self.expression = expression

class BlockNode(StatementNode): """AST node for a block.""" def **init**(self, statements): self.statements = statements

class ExpressionNode(ASTNode): """Base class for expression nodes.""" pass

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

class NumberNode(ExpressionNode): """AST node for a number.""" def **init**(self, value): self.value = value

class StringNode(ExpressionNode): """AST node for a string.""" def **init**(self, value): self.value = value

class IdentifierNode(ExpressionNode): """AST node for an identifier.""" def **init**(self, name): self.name = name

class ConditionNode(ASTNode): """AST node for a condition.""" def **init**(self, left, op, right): self.left = left self.op = op self.right = right ```

### 5.3.1 AST Node Classes

These AST node classes represent the different elements of a TinyCL program. Each class corresponds to a specific grammar rule and contains the necessary information to represent that element in the AST.

### 5.3.2 Tree Construction

The parsing functions we implemented earlier construct the AST as they parse the input. Each parsing function returns an AST node representing the parsed element.

## 5.4 Semantic Analysis

After parsing the program and building the AST, we need to perform semantic analysis to check for errors and prepare for interpretation.

### 5.4.1 Symbol Table

The symbol table keeps track of variables and their types:

```python class SymbolTable: """Symbol table for tracking variables.""" def **init**(self): self.symbols = {}

```
def define(self, name, value):
    """Define a variable."""
    self.symbols[name] = value

def lookup(self, name):
```
```

```
    """Look up a variable."""
    return self.symbols.get(name)

def update(self, name, value):
    """Update a variable's value."""
    if name not in self.symbols:
        raise NameError(f"Variable '{name}' not defined")
    self.symbols[name] = value
```

```

### 5.4.2 Type Checking

We can add type checking to ensure that operations are performed on compatible types:

```python

# Note: This code assumes AST classes are imported: from src.tinycl.ast import *

def check_types(node, symbol_table): """Check types in the AST.""" if isinstance(node, ProgramNode): for statement in node.statements: check_types(statement, symbol_table)

```
elif isinstance(node, VariableDeclNode):
    # Check that the expression has a valid type
    expr_type = get_expression_type(node.expression, symbol_table)
    if expr_type is None:
        raise TypeError(f"Invalid expression in variable declaration")

    # Define the variable
    symbol_table.define(node.identifier.name, None)

elif isinstance(node, AssignStatementNode):
```

```
    # Check that the variable is defined
    if symbol_table.lookup(node.identifier.name) is None:
        raise NameError(f"Variable '{node.identifier.name}' not defined'

    # Check that the expression has a valid type
    expr_type = get_expression_type(node.expression, symbol_table)
    if expr_type is None:
        raise TypeError(f"Invalid expression in assignment")

# Similar implementations for other node types

return True
```

def get_expression_type(node, symbol_table): """Get the type of an expression.""" if isinstance(node, NumberNode): return "number"

```
elif isinstance(node, StringNode):
    return "string"

elif isinstance(node, IdentifierNode):
    # Check that the variable is defined
    if symbol_table.lookup(node.name) is None:
        raise NameError(f"Variable '{node.name}' not defined")

    # Return the type of the variable
    value = symbol_table.lookup(node.name)
    if isinstance(value, int):
        return "number"
    elif isinstance(value, str):
        return "string"
    else:
        return None

elif isinstance(node, BinaryOpNode):
    # Get the types of the operands
    left_type = get_expression_type(node.left, symbol_table)
    right_type = get_expression_type(node.right, symbol_table)
```

```
    # Check that the operation is valid for the operand types
    if node.op in ['+', '-', '*', '/']:
        if left_type == "number" and right_type == "number":
            return "number"
        elif node.op == '+' and (left_type == "string" or right_type ==
            return "string"  # String concatenation
        else:
            raise TypeError(f"Invalid operand types for operator '{node.
    else:
        raise TypeError(f"Unknown operator: {node.op}")

# Similar implementations for other node types

return None
```

```

## 5.5 Interpreter Implementation

Now, let's implement the interpreter that will execute TinyCL programs:

```python from src.tinycl.ast import *

class TinyCLInterpreter: """Interpreter for TinyCL programs.""" def **init**(self): self.symbol_table = SymbolTable()

```
def interpret(self, program):
    """Interpret a TinyCL program."""
    if not isinstance(program, ProgramNode):
        raise TypeError("Expected a ProgramNode")

    # Perform semantic analysis
    check_types(program, self.symbol_table)

    # Execute the program
    return self.execute_program(program)
```

```
def execute_program(self, program):
    """Execute a program."""
    result = None
    for statement in program.statements:
        result = self.execute_statement(statement)
    return result

def execute_statement(self, statement):
    """Execute a statement."""
    if isinstance(statement, VariableDeclNode):
        return self.execute_var_statement(statement)
    elif isinstance(statement, AssignStatementNode):
        return self.execute_assign_statement(statement)
    elif isinstance(statement, IfStatementNode):
        return self.execute_if_statement(statement)
    elif isinstance(statement, WhileStatementNode):
        return self.execute_while_statement(statement)
    elif isinstance(statement, PrintStatementNode):
        return self.execute_print_statement(statement)
    elif isinstance(statement, BlockNode):
        return self.execute_block(statement)
    else:
        raise TypeError(f"Unknown statement type: {type(statement)}")

def execute_var_statement(self, statement):
    """Execute a variable declaration statement."""
    value = self.evaluate_expression(statement.expression)
    self.symbol_table.define(statement.identifier.name, value)
    return None

def execute_assign_statement(self, statement):
    """Execute an assignment statement."""
    value = self.evaluate_expression(statement.expression)
    self.symbol_table.update(statement.identifier.name, value)
    return None

def execute_if_statement(self, statement):
```

```
    """Execute an if statement."""
    condition = self.evaluate_condition(statement.condition)
    if condition:
        return self.execute_statement(statement.then_statement)
    elif statement.else_statement is not None:
        return self.execute_statement(statement.else_statement)
    return None

def execute_while_statement(self, statement):
    """Execute a while statement."""
    result = None
    while self.evaluate_condition(statement.condition):
        result = self.execute_statement(statement.body)
    return result

def execute_print_statement(self, statement):
    """Execute a print statement."""
    value = self.evaluate_expression(statement.expression)
    print(value)
    return None

def execute_block(self, block):
    """Execute a block."""
    result = None
    for statement in block.statements:
        result = self.execute_statement(statement)
    return result

def evaluate_expression(self, expression):
    """Evaluate an expression."""
    if isinstance(expression, NumberNode):
        return expression.value
    elif isinstance(expression, StringNode):
        return expression.value
    elif isinstance(expression, IdentifierNode):
        return self.symbol_table.lookup(expression.name)
    elif isinstance(expression, BinaryOpNode):
```

```
        left = self.evaluate_expression(expression.left)
        right = self.evaluate_expression(expression.right)

        if expression.op == '+':
            return left + right
        elif expression.op == '-':
            return left - right
        elif expression.op == '*':
            return left * right
        elif expression.op == '/':
            return left / right
        else:
            raise ValueError(f"Unknown operator: {expression.op}")
    else:
        raise TypeError(f"Unknown expression type: {type(expression)}")

def evaluate_condition(self, condition):
    """Evaluate a condition."""
    left = self.evaluate_expression(condition.left)
    right = self.evaluate_expression(condition.right)

    if condition.op == '==':
        return left == right
    elif condition.op == '!=':
        return left != right
    elif condition.op == '<':
        return left < right
    elif condition.op == '>':
        return left > right
    elif condition.op == '<=':
        return left <= right
    elif condition.op == '>=':
        return left >= right
    else:
        raise ValueError(f"Unknown comparison operator: {condition.op}")
```

5.5.1 Runtime Environment

The `SymbolTable` class provides the runtime environment for TinyCL programs. It keeps track of variables and their values.

5.5.2 Expression Evaluation

The `evaluate_expression` method evaluates expressions by recursively evaluating their components and applying the appropriate operations.

5.5.3 Statement Execution

The `execute_statement` method executes statements by dispatching to the appropriate method based on the statement type.

5.6 Example Programs and Testing

Let's create some example TinyCL programs to test our implementation:

```python

# Example 1: Factorial

factorial_program = """

# Calculate factorial

var n = 5; var factorial = 1;

while (n > 0) { factorial = factorial * n; n = n - 1; }

print("Factorial: " + factorial); """

# Example 2: Fibonacci

fibonacci_program = """
```

Calculate Fibonacci numbers

var n = 10; var a = 0; var b = 1; var i = 0;

print("Fibonacci sequence:"); print(a); print(b);

while (i < n - 2) { var c = a + b; print(c); a = b; b = c; i = i + 1; } """

Test the TinyCL interpreter

if **name** == "**main**": parser = TinyCLParser() interpreter = TinyCLInterpreter()

```
print("Testing factorial program:")
try:
    ast = parser.parse(factorial_program)
    interpreter.interpret(ast)
except Exception as e:
    print(f"Error: {e}")

print("\nTesting Fibonacci program:")
try:
    ast = parser.parse(fibonacci_program)
    interpreter.interpret(ast)
except Exception as e:
    print(f"Error: {e}")
```

```

## Summary

In this chapter, we've built a complete tiny programming language called TinyCL using our PEG parser library. We've:

1. Designed the language features and grammar
2. Implemented the parser to build an AST
3. Added semantic analysis for type checking
4. Created an interpreter to execute TinyCL programs
```

5. Tested the implementation with example programs

TinyCL demonstrates the power and flexibility of the TinyPEG library for building parsers and interpreters. While it's a simple language, it includes many of the fundamental concepts found in larger programming languages.

In the next chapter, we'll explore advanced topics and extensions to both the TinyPEG library and the TinyCL language.

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 PEG`Parser` 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.

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.

# Appendix C: TinyCL (Tiny C-Like Language) Reference

This appendix provides a comprehensive reference for the TinyCL language, including its complete syntax, semantics, built-in features, and example programs.

## C.1 Language Overview

TinyCL (Tiny C-Like Language) is a modern, fully-featured programming language that demonstrates advanced language implementation techniques. It includes:

- **Variables and Constants**: `var` and `const` declarations
- **Functions**: User-defined functions with parameters and return values
- **Arrays**: Dynamic arrays with indexing
- **Full Expression System**: Arithmetic, logical, and comparison operators with proper precedence
- **Control Flow**: If-else statements and while loops
- **Data Types**: Numbers, strings, characters, booleans, and arrays
- **Comments**: Single-line comments with #

## C.2 Complete Syntax Reference

### C.2.1 Program Structure

A TinyCL program consists of a sequence of statements:

```
ebnf Program ::= Statements Statements ::= Statement*
```

C.2.2 Statements

TinyCL supports the following types of statements:

```
ebnf Statement ::= FunctionDecl | VariableDecl | ConstantDecl
| IfStatement | WhileStatement | PrintStatement |
ReturnStatement | AssignmentStatement | ExpressionStatement |
Block | Comment
```

Variable Declaration

A variable declaration creates a new variable and initializes it:

```ebnf
VariableDecl ::= "var" Identifier "=" Expression ";"
```

Example: `var x = 42; var name = "Alice"; var numbers = [1, 2, 3];`

Constant Declaration

A constant declaration creates an immutable value:

```ebnf
ConstantDecl ::= "const" Identifier "=" Expression ";"
```

Example: `const PI = 3; const MAX_SIZE = 100;`

Function Declaration

A function declaration defines a reusable block of code:

```ebnf
FunctionDecl ::= "func" Identifier "(" Parameters? ")" Block
Parameters ::= Identifier ("," Identifier)*
```

Example: ``` func add(a, b) { return a + b; }

func factorial(n) { if (n <= 1) { return 1; } else { return n * factorial(n - 1); } } ```

Assignment Statement

An assignment statement assigns a new value to an existing variable:

```ebnf
AssignmentStatement ::= Identifier "=" Expression ";"
```

Example: `x = 42; name = "Bob"; numbers[0] = 10;`

If Statement

An if statement conditionally executes code based on a condition:

```ebnf
IfStatement ::= "if" "(" Expression ")" Block ("else" Block)?
```

Example: ``` if (x > 0) { print("Positive"); } else { print("Non-positive"); }

if (x > 0 && x < 10) { print("Single digit positive"); } ```

While Statement

A while statement repeatedly executes code as long as a condition is true:

```
ebnf WhileStatement ::= "while" "(" Expression ")" Block
```

Example: `while (x > 0) { print(x); x = x - 1; }`

Print Statement

A print statement outputs a value:

```
ebnf PrintStatement ::= "print" "(" Expression ")" ";"
```

Example: `print("Hello, world!"); print(42); print("Result: " + result);`

Return Statement

A return statement exits a function and optionally returns a value:

```
ebnf ReturnStatement ::= "return" Expression? ";"
```

Example: `return 42; return x + y; return; # Return without a value`

Block

A block groups multiple statements together:

```
ebnf Block ::= "{" Statements? "}"
```

Example: `{ var x = 1; var y = 2; print(x + y); }`

Comment

A comment is a line of text that is ignored by the parser:

```
ebnf Comment ::= "#" [^\n]*
```

Example: ```

This is a comment

```

### C.2.3 Complete Expression System

TinyCL has a comprehensive expression system with proper operator precedence:

```
ebnf Expression ::= LogicalOr LogicalOr ::= LogicalAnd ("||" LogicalAnd)* LogicalAnd ::= Equality ("&&" Equality)* Equality ::= Comparison (("!=" | "==") Comparison)* Comparison ::= Term (("<=" | ">=" | "<" | ">") Term)* Term ::= Factor (("+" | "-") Factor)* Factor ::= Unary (("*" | "/") Unary)* Unary ::= ("!" | "-")? Postfix Postfix ::= Primary ("[" Expression "]")* Primary ::= "(" Expression ")" | Identifier "(" Arguments? ")" | "[" Arguments? "]" | Identifier | Number | String | Character | "true" | "false"
```

#### Arithmetic Operators

```
2 + 3 * 4 # Result: 14 (proper precedence) (2 + 3) * 4 # Result: 20 (parentheses override precedence) 10 / 2 - 1 # Result: 4
```

Logical Operators

```
true && false # Result: false true || false # Result: true ! true # Result: false x > 0 && x < 10 # Compound condition
```

Comparison Operators

```
x == 42 # Equal to x != 0 # Not equal to x < 10 # Less than x > 5 # Greater than x <= 100 # Less than or equal to x >= 1 # Greater than or equal to
```
```

Array Operations

```
var arr = [1, 2, 3]; # Array literal var first = arr[0]; # Array access arr[1] = 42; # Array assignment var mixed = [1, "hello", true]; # Mixed types
```

Function Calls

```
var result = add(10, 20); var fact = factorial(5); print("Hello");
```

C.2.4 Data Types and Literals

TinyCL supports multiple data types:

Numbers

`ebnf Number ::= [0-9]+` Example: `42`, `0`, `123`

Strings

`ebnf String ::= '"' StringChar* '"' StringChar ::= [printable characters] | EscapeSequence EscapeSequence ::= '\' ('"' | '\' | 'n' | 'r' | 't' | '0' | 'b' | 'f' | 'v' | 'l')` Example: `"Hello, world!"`, `"Line 1\nLine 2"`, `"Quote: \"Hello\""`

Characters

`ebnf Character ::= "'" CharChar "'" CharChar ::= [printable character] | EscapeSequence` Example: `'A'`, `'1'`, `'\n'`

Booleans

```
true false
```

Arrays

```
[1, 2, 3] ["hello", "world"] [1, "mixed", true] [] # Empty array
```

C.2.5 Identifiers

Identifiers are used for variable, constant, and function names:

```
ebnf Identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
```

Example: `x counter first_name calculateTotal MAX_SIZE`

C.3 Standard Library

TinyCL has a minimal standard library with the following built-in functionality:

C.3.1 Input/Output

- `print(expression)`: Print the value of an expression.

Example: `print("Hello, world!"); print(42); print("The answer is " + 42);`

C.3.2 Arithmetic Operations

TinyCL supports the following arithmetic operations:

- Addition: `a + b`
- Subtraction: `a - b`
- Multiplication: `a * b`
- Division: `a / b`

Example: `let x = 2 + 3; # x = 5 let y = x * 4; # y = 20 let z = y / 2; # z = 10 let w = z - 1; # w = 9`

C.3.3 String Operations

TinyCL supports string concatenation using the + operator:

Example: `let name = "Alice"; let greeting = "Hello, " + name + "!"; # greeting = "Hello, Alice!"`

C.3.4 Comparison Operations

TinyCL supports the following comparison operations:

- Equal to: `a == b`
- Not equal to: `a != b`
- Less than: `a < b`
- Greater than: `a > b`
- Less than or equal to: `a <= b`
- Greater than or equal to: `a >= b`

Example: ``` if (x == 42) { print("x is 42"); }

if (y != 0) { print("y is not 0"); }

if (z < 10) { print("z is less than 10"); } ```

C.4 Example Programs

Here are some example TinyCL programs to demonstrate the language's features:

C.4.1 Hello, World!

```

# Hello, World! program

print("Hello, World!"); ```

### C.4.2 Factorial with Functions

```

Calculate factorial using recursion

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

var n = 5; var result = factorial(n); print("Factorial of " + n + " is " + result); ```

C.4.3 Fibonacci Sequence

```
# Calculate Fibonacci numbers

var n = 10; var a = 0; var b = 1; var i = 0;

print("Fibonacci sequence:"); print(a); print(b);

while (i < n - 2) { var c = a + b; print(c); a = b; b = c; i = i + 1; } ```

### C.4.4 Array Processing

```
Working with arrays

var numbers = [5, 2, 8, 1, 9]; var sum = 0; var i = 0;

Calculate sum

while (i < 5) { sum = sum + numbers[i]; i = i + 1; }

print("Sum: " + sum);

Find maximum

var max = numbers[0]; i = 1; while (i < 5) { if (numbers[i] > max) { max = numbers[i]; } i = i + 1; }

print("Maximum: " + max); ```

C.4.5 FizzBuzz

```
```

FizzBuzz program

let i = 1;

while (i <= 100) { if (i % 15 == 0) { print("FizzBuzz"); } else { if (i % 3 == 0) { print("Fizz"); } else { if (i % 5 == 0) { print("Buzz"); } else { print(i); } } } i = i + 1; } ```

C.4.6 Prime Numbers

```

# Print prime numbers up to n

let n = 100; let i = 2;

while (i <= n) { let is_prime = 1; let j = 2;

```
while (j < i) {
    if (i % j == 0) {
        is_prime = 0;
    }
    j = j + 1;
}

if (is_prime == 1) {
    print(i);
}

i = i + 1;
```

} ```

## C.5 Language Features Summary

TinyCL is a comprehensive programming language with the following implemented features:
```

✅ Implemented Features

1. **Complete Data Types**: Numbers, strings, characters, booleans, and arrays
2. **User-Defined Functions**: Function declarations with parameters and return values
3. **Arrays and Indexing**: Dynamic arrays with element access and assignment
4. **Full Expression System**: Arithmetic, logical, and comparison operators with proper precedence
5. **Control Flow**: If-else statements and while loops
6. **Variable Management**: Variable and constant declarations
7. **Comments**: Single-line comments with #
8. **Built-in I/O**: Print statement for output

📋 Current Limitations

1. **Limited I/O**: Only supports output via `print` statement (no input capabilities)
2. **No Exception Handling**: No try-catch blocks or error handling mechanisms
3. **No Modules**: No support for importing code from other files
4. **Integer-Only Numbers**: No floating-point number support
5. **Minimal Standard Library**: Only basic built-in functions

🚀 Possible Future Extensions

1. **Floating-Point Numbers**: Add support for decimal numbers
2. **Input Functions**: Add `input()` or `read()` functions
3. **Exception Handling**: Add try-catch blocks for error handling
4. **Module System**: Add `import` statements for code reuse
5. **Object-Oriented Features**: Add classes and objects
6. **Standard Library**: Expand with string manipulation, math functions, etc.
7. **File I/O**: Add file reading and writing capabilities

Despite the current limitations, TinyCL is a fully functional programming language capable of expressing complex algorithms and computations.

Summary

TinyCL is a simple but complete programming language with variables, control structures, and basic I/O. Its syntax is inspired by popular programming languages like JavaScript and Python, making it easy to learn and use.

This appendix has provided a comprehensive reference for TinyCL, including its syntax, semantics, standard library, and example programs. With this information, you should be able to write and understand TinyCL programs.