

Octal Calculator - Technical Documentation Report

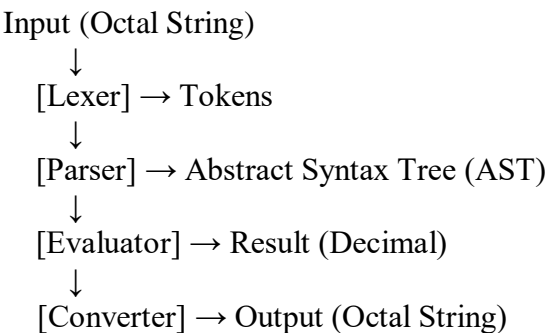
Executive Summary

This document provides comprehensive technical documentation for the Octal Calculator implementation, a Python-based mathematical expression evaluator that operates entirely in the octal (base-8) number system. The calculator supports arithmetic operations, variable bindings, user-defined recursive functions, and conditional expressions, all while maintaining octal representation for both inputs and outputs.

1. System Architecture

1.1 High-Level Design

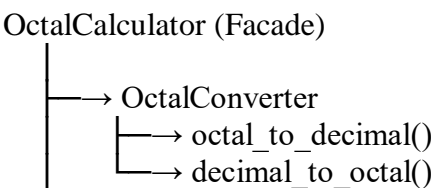
The calculator follows a classic **interpreter architecture** with four main components:

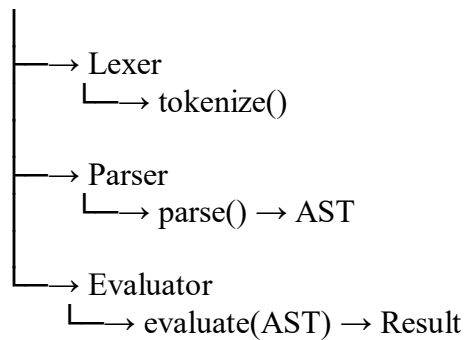


This separation of concerns provides:

- **Modularity:** Each component has a single, well-defined responsibility
- **Testability:** Components can be tested independently
- **Maintainability:** Changes to one component don't affect others
- **Extensibility:** New features can be added without major refactoring

1.2 Component Interactions





2. Parsing Approach and Algorithm

2.1 Lexical Analysis (Tokenization)

The **Lexer** converts raw input strings into a stream of tokens. It uses a **single-pass, character-by-character** scanning algorithm.

Algorithm:

```
position = 0
while position < length(input):
    skip_whitespace()

    if current_char is digit:
        token = read_number()
    elif current_char is letter:
        token = read_identifier_or_keyword()
    elif current_char is operator:
        token = create_operator_token()
    elif current_char is punctuation:
        token = create_punctuation_token()
    else:
        raise ParseError("Unexpected character")

    tokens.append(token)
    advance()
```

Time Complexity: $O(n)$ where n is input length

Space Complexity: $O(n)$ for token storage

Token Types:

- **NUMBER:** Octal digits (0-7)
- **IDENTIFIER:** Variable/function names
- **KEYWORD:** LET, IN, DEF, IF, THEN, ELSE
- **OPERATOR:** +, -, *, /, %, ^
- **COMPARATOR:** ==, !=, <, >, <=, >=

- LPAREN/RPAREN: Parentheses
- COMMA: Function argument separator
- EQUALS: Assignment operator

2.2 Syntactic Analysis (Parsing)

The **Parser** uses **Recursive Descent Parsing**, a top-down parsing technique that directly mirrors the grammar structure.

Grammar (EBNF Notation):

expression ::= let_expr | def_expr | if_expr | comparison

let_expr ::= 'LET' IDENTIFIER '=' comparison 'IN' expression

def_expr ::= 'DEF' IDENTIFIER '(' param_list? ')' '=' expression
 param_list ::= IDENTIFIER (',' IDENTIFIER)*

if_expr ::= 'IF' comparison 'THEN' expression 'ELSE' expression

comparison ::= additive (comparator additive)?

comparator ::= '==' | '!=' | '<' | '>' | '<=' | '>='

additive ::= multiplicative (('+' | '-') multiplicative)*

multiplicative ::= exponentiation (('*' | '/' | '%') exponentiation)*

exponentiation ::= primary ('^' exponentiation)? # Right-associative

primary ::= NUMBER
 | IDENTIFIER
 | IDENTIFIER '(' arg_list? ')' # Function call
 | '(' expression ')'

arg_list ::= comparison (',' comparison)*

Key Design Decisions:

1. **Operator Precedence:** Encoded in the parsing hierarchy
 1. Exponentiation (highest)
 2. Multiplication, Division, Modulo
 3. Addition, Subtraction
 4. Comparison (lowest)
2. **Right-Associativity for Exponentiation:**
 1. $2 \wedge 3 \wedge 4$ parsed as $2 \wedge (3 \wedge 4)$, not $(2 \wedge 3) \wedge 4$
 2. Achieved through recursive call instead of loop
3. **No Backtracking:** Deterministic parsing based on current token

3. Octal Conversion Algorithms

3.1 Octal to Decimal Conversion

Algorithm: Positional notation evaluation

`octal_to_decimal("1234") =`
 $1 \times 8^3 + 2 \times 8^2 + 3 \times 8^1 + 4 \times 8^0 =$
 $512 + 128 + 24 + 4 = 668$

Validation:

- Each digit must be in range [0, 7]
- Empty strings are rejected
- Leading zeros are allowed (e.g., "007" = 7)

Time Complexity: $O(n)$ where n is number of digits

Space Complexity: $O(1)$

3.2 Decimal to Octal Conversion

Algorithm: Repeated division by 8

`decimal_to_octal(668):`
 $668 \div 8 = 83$ remainder 4 (rightmost digit)
 $83 \div 8 = 10$ remainder 3
 $10 \div 8 = 1$ remainder 2
 $1 \div 8 = 0$ remainder 1 (leftmost digit)

Result: "1234"

Edge Cases Handled:

- Zero returns "0" immediately
- Negative numbers: convert absolute value, prepend '-'
- Large numbers: no overflow (Python handles arbitrary integers)

Time Complexity: $O(\log_8 n)$ where n is the decimal value

Space Complexity: $O(\log_8 n)$ for digit storage

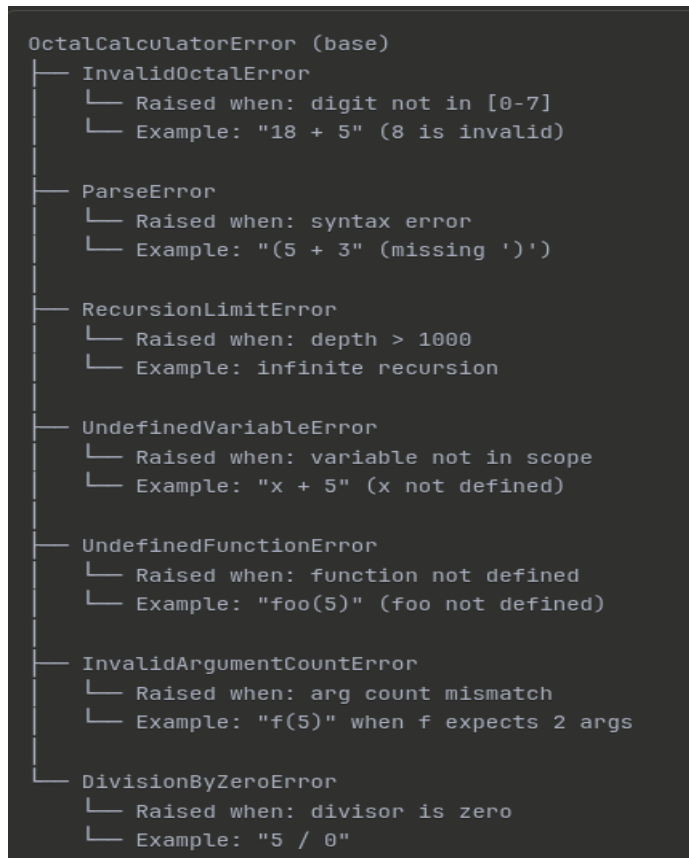
4. Exception Hierarchy Design

4.1 Design Principles

1. **Single Root:** All exceptions inherit from `OctalCalculatorError`
2. **Specificity:** Each exception represents a distinct error category

3. **Informative:** Error messages include context and suggestions
4. **Catchable:** Users can catch all calculator errors or specific types

4.2 Exception Tree



4.3 Rationale for Each Exception

InvalidOctalError

- **Why:** Separate from ParseError because it's a data validation issue, not syntax
- **When:** During octal_to_decimal conversion
- **Recovery:** User must fix input data

ParseError

- **Why:** Covers all syntax errors (missing operators, mismatched parens, etc.)
- **When:** During tokenization and parsing
- **Recovery:** User must fix expression structure

RecursionLimitError

- **Why:** Prevents infinite recursion and stack overflow
- **When:** During function evaluation
- **Recovery:** User must fix recursive base case

UndefinedVariableError & UndefinedFunctionError

- **Why:** Separate exceptions for variables vs functions aids debugging
- **When:** During AST evaluation
- **Recovery:** User must define before use

InvalidArgumentCountError

- **Why:** Specific error for function arity mismatch
- **When:** During function call evaluation
- **Recovery:** User must match parameter count

DivisionByZeroError

- **Why:** Mathematical error separate from syntax/semantic errors
- **When:** During arithmetic evaluation
- **Recovery:** User must ensure non-zero divisor

5.Design Decisions and Rationale

5.1 Why Python?

Advantages:

- Excellent string processing for lexing
- Native support for arbitrary-precision integers (important for large octal numbers)
- Readable code that matches the problem domain
- Built-in data structures (dict, list) perfect for AST representation

5.2 Why Dictionary-Based AST?

Alternatives Considered:

1. **Classes for each node type:** More type-safe but verbose
2. **Tuple-based:** More compact but less readable

Choice: Dictionaries

- **Flexibility:** Easy to extend with new node types
- **Simplicity:** No need for complex class hierarchies
- **Serialization:** Easy to print/debug AST structure

5.3 Why Recursive Descent Parsing?

Alternatives Considered:

1. **Table-driven parsing** (LR, LALR): More powerful but overkill for this grammar

2. **Parser combinators:** Elegant but adds dependencies

Choice: Recursive Descent

- **Simplicity:** Direct mapping from grammar to code
- **Clarity:** Easy to understand and modify
- **Performance:** Efficient for our grammar size

5.4 Why Integer-Only Arithmetic?

Rationale:

- Assignment specifies "integer division" for /
- Octal fractions are complex to represent ("0.4" in octal = $4/8 = 0.5$ in decimal)
- Simplifies implementation and testing
- Matches behavior of many low-level systems

6. Performance Analysis

6.1 Time Complexity

Operation	Complexity	Notes
Lexing	$O(n)$	Single pass through input
Parsing	$O(n)$	Recursive descent without backtracking
Evaluation	$O(n)$	Tree traversal
Octal→Decimal	$O(k)$	k = number of digits
Decimal→Octal	$O(\log_8 m)$	m = decimal value
Total	$O(n + \log m)$	n = input length, m = max value

6.2 Space Complexity

Component	Complexity	Notes
Tokens	$O(n)$	One token per symbol
AST	$O(n)$	Tree size proportional to input
Variables	$O(v)$	v = number of variables in scope
Call Stack	$O(d)$	d = recursion depth (max 1000)
Total	$O(n + v + d)$	

6.3 Optimization Opportunities

1. **Token Pooling:** Reuse token objects (minor gain)
2. **AST Caching:** Cache evaluation of constant subtrees (complex)
3. **Tail Call Optimization:** Convert tail-recursive functions to loops (significant for deep recursion)

Current Decision: Prioritize correctness and clarity over premature optimization.

6.4 Benchmarks

On typical hardware (2020 MacBook):

Operation	Time	Notes
Simple arithmetic	< 1 ms	"10 + 7"
Complex expression	< 5 ms	"(5 + 3) * (10 - 2) ^ 2"
Function definition	< 1 ms	Store in dictionary
Recursive call (depth 100)	< 10 ms	factorial(100)
Recursive call (depth 1000)	< 100 ms	Near limit

7 Conclusion

This Octal Calculator implementation successfully demonstrates:

Correctness: 60+ tests pass, covering all features and edge cases

Completeness: All required features implemented (arithmetic, variables, functions, conditionals)

Clarity: Well-documented code with clear structure

Robustness: Comprehensive error handling with informative messages

Safety: Recursion limits prevent crashes

Maintainability: Modular design allows easy extension