Octal Calculator

Design & Implementation Report

Overview

This document explains the design and implementation choices for the Octal Calculator: a small language and evaluator that parses and evaluates arithmetic expressions in base-8, supports local LET bindings, user-defined recursive DEF functions, and conditional IF/THEN/ELSE expressions. It covers parsing, octal conversion routines, recursion safety, scoping, exceptions, assertions, and important design tradeoffs and assumptions.

---

1. Parsing approach and algorithm

1.1 High level

The parser is a *hand-written recursive-descent parser* driven by a linear token stream produced by a regular-expression tokenizer. It implements a conventional expression grammar with precedence and supports top-level DEF statements and expressions.

1.2 Tokenization

- Implemented with a single regular expression composed from token specifications (TOKEN_SPEC) and compiled into TOKEN_RE.
- Token types:
    - NUMBER: -?[0-7]+ (octal digits, optional leading minus).
    - NAME: identifiers [A-Za-z_][A-Za-z_0-9]*. Keyword names (LET, IN, DEF, IF, THEN, ELSE) are normalized to keyword token types.
    - OP: operators and punctuation including a list of multi-char operators (==, !=, <=, >=) and the unicode exponent symbol ∧ .
    - SKIP, NEWLINE: ignored.
- On an unknown character a ParseError is raised with the offending position.

1.3 Recursive descent parser

- The parser follows a standard precedence structure, where each nonterminal handles a level of precedence:

    - parse_expression → LET / IF or parse_comparison
    - parse_comparison → handles ==, !=, <, >, <=, >=
    - parse_addsub → +, -
    - parse_muldiv → *, /, %
    - parse_pow → ^ and ∧ (right-associative)
    - parse_unary → unary + / -
    - parse_primary → literals, variables, function calls, parenthesized expressions
- DEF statements are parsed at the top level by parse_def and return a dedicated AST node (('DEF', name, params, body)).
- The AST is represented as plain Python tuples (tagged nodes), e.g. ('BINOP', '+', left, right), ('CALL', name, args), ('LET', var, value_expr, body_expr), ('IF', cond, true, false).

1.4 Error handling in parsing

- Errors are reported as ParseError with a helpful message and position when possible.
- The parser enforces single top-level statement per execute() call; extra input after a valid AST raises ParseError("Unexpected extra input...").

---

2. Octal ↔ integer conversions

2.1 Constraints

- **No usage of** int(..., 8) or oct() — conversions must be manual.
- Inputs/outputs are octal strings. Internally, evaluation is done with Python int values.

2.2 octal_str_to_int(octal_string)

- Accepts a string, strips whitespace, accepts an optional leading -.
- Validates every character is '0'..'7'. Any invalid input raises ConversionError.
- Accumulates the integer value with a repeated multiply-by-8 and add digit algorithm:

```
value = 0
for ch in digits:
    digit = ord(ch) - ord('0')
    value = value * 8 + digit
```

- Returns negative value if input had a leading -.

2.3 int_to_octal_str(integer_value)

- Accepts an int.
- Handles zero specially, returning "0".
- For nonzero values, iteratively divides by 8 gathering remainders, then reverses the digits.
- Prepends - if original integer was negative.
- Guarantees canonical octal string (no leading zeros except single "0").

---

3. Recursion safety and depth tracking

3.1 Recursion model

- Functions are stored with their parameter list and body (no closure environment). Calls are evaluated by:
    1. Evaluating argument expressions in the caller environment.
    2. Building a fresh local environment mapping parameters to their evaluated integer values.
    3. Evaluating the function body in that local environment.

3.2 Depth tracking and limit

- The evaluator tracks a single integer self.call_depth.
- Before performing a call, it checks:

```
if self.call_depth >= RECURSION_LIMIT:
    raise RecursionLimitError(RECURSION_LIMIT)
```

- RECURSION_LIMIT is set to 1000 (as required). On each call self.call_depth is incremented before evaluating the body and decremented in a finally block to ensure correct bookkeeping even when exceptions occur.
- A RecursionError from Python (if raised for other reasons) is caught and converted to RecursionLimitError to present a consistent API.

3.3 Rationale

- Explicit tracking prevents stack overflows and runaway recursion.
- A single global depth counter is straightforward and sufficient because recursion depth is proportional to nested CALLs.

---

4. Variable scope management strategy

4.1 Environments

- Environments are represented as Python dictionaries {name: int} mapping variable names to integer values.
- There is no single global variable environment for expressions — every evaluation is passed an env mapping.

4.2 LET semantics

- LET name = value_expr IN body_expr:
    1. Evaluate value_expr in the current env.
    2. Construct new_env = dict(env) — a shallow copy — and insert new_env[name] = evaluated_value.
    3. Evaluate body_expr under new_env.
- This gives **lexical scoping** for variables within the IN expression and allows nested LETs and shadowing (inner LET names hide outer ones because new_env is a copy).

4.3 Function parameters

- For a function call, after evaluating argument values, a local_env = {param_i: arg_i} is created and used to evaluate the function body.
- **Function bodies do not automatically capture the caller's or definition's outer environment** — they are evaluated with a fresh environment consisting only of parameters. This is a simple and explicit design consistent with the requirement that parameters are local.
    - If closure semantics (capturing outer LET variables where the function was defined) are required, the function object would need to carry a stored environment captured at definition time.

4.4 Name resolution

- When encountering a VAR node:
    - If the name exists in current env, the value is returned.
    - Otherwise, if it matches a defined function name, evaluation raises EvaluationError("Function 'name' used without call") (prompts the user to call functions explicitly).
    - Otherwise a NameErrorEval is raised.

---

5. Exception hierarchy design and rationale

5.1 Exceptions provided

All exceptions derive from OctalCalcError (a base class) for easy grouping. The hierarchy:

- OctalCalcError (base)
  - ParseError(OctalCalcError)
    - carries message and position (optional)
  - ConversionError(OctalCalcError)
    - raised for invalid octal strings
  - AssertionFailure(OctalCalcError)
  - EvaluationError(OctalCalcError)
    - DivisionByZeroError(EvaluationError) — division / modulo by zero
    - RecursionLimitError(EvaluationError) — recursion depth exceeded (carries limit)
    - ArityError(EvaluationError) — wrong number of arguments to a function
    - NameErrorEval(EvaluationError) — undefined variable or function

5.2 Rationale

- Grouping under OctalCalcError enables callers to catch all domain-specific errors in a single catch if desired.
- Subclasses provide precise error types for expected runtime failures and improve testability (tests assert that specific exceptions are raised).
- ParseError includes source position to help downstream tools point users to syntax errors precisely.
- RecursionLimitError and ArityError include contextual data (limit, expected/got) for clearer diagnostics.

---

6. Assertion strategy (what is being protected and why)

6.1 Use of assert

- assert statements are used as internal sanity checks, not for handling user errors:
  - Validate function arguments and invariants in public APIs (e.g., int_to_octal_str ensures input is int, Evaluator.execute expects parsed AST shapes).
  - Confirm postconditions (e.g., a stored DEF exists in self.functions after definition).
- These assertions document internal expectations and help catch programmer mistakes during development and testing.

6.2 Rationale

- assert is lightweight and appropriate for checking internal invariants; user-facing input errors are signaled with the typed exceptions in the hierarchy above.
- Note: If running Python with optimizations (-O) disables assert, some developer checks would be suppressed — if production defenses are required even in optimized runs, convert important asserts to explicit if not ...: raise AssertionFailure(...).

---

7. Design decisions and assumptions

7.1 Explicit design choices

- **Manual conversions**: Per requirement, conversions are implemented without int(...,8) or oct().
- **Recursive descent parser**: Chosen for clarity and control over precedence/associativity (right-associative exponent).
- **AST as tuples**: Lightweight representation easy to pattern-match in evaluator.
- **Function storage**: Functions are stored globally in Evaluator.functions, and DEF persists across execute() calls.
- **No closures**: Function definitions do not capture lexical environments; only parameters are available in the function body. This simplifies implementation and matches the stated requirement about local parameters. If closure semantics are desired, we can extend Function to carry the defining environment.

- **Single top-level statement per execute()**: Parser requires one DEF or expression per execute call and enforces EOF. This matches how the current tests and REPL are used.
- **Modulo semantics**: Implemented with quotient computed by truncation toward zero (via int(left/right)) and remainder computed as left - right * quotient. This yields sign behavior tested in the suite. This choice is explicit and documented rather than relying on the language (Python %) semantics.

7.2 Assumptions

- **Inputs are short ASCII** expressions; tokenizer recognizes a unicode exponent ∧ in addition to ^.
- **Function and variable names** are case-sensitive; keywords are recognized case-insensitively (converted to uppercase when tokenized).
- **All numeric literals are octal**; users represent numbers in octal in source text.
- **Performance**: Large integer exponentiation uses Python's native big integers. Very large exponents can be expensive but are permitted; negative exponents are rejected as producing non-integers.

7.3 Potential extensions (not implemented by default)

- Multi-statement top-level programs (program := statement*) instead of single-statement execute.
- Closures: store a function's defining environment to enable closures.
- Additional builtins (I/O, booleans as explicit types, lists).
- Different modulo semantics (match Python's % or mathematical modulo) if desired.
- Better error recovery in parser for interactive editing.

---

8. Examples & expected behaviors (quick reference)

- Arithmetic and precedence:
    - "10 + 7" → "17" (octal: 8 + 7 = 15 decimal → octal 17)
    - "2 + 3 * 4" → "16" (3*4 = 12 decimal → octal 14; 2 + 12 = 14 decimal → octal 16)
    - "(2 + 3) * 4" → "24"
- LET scoping:
    - "LET x = 10 IN x + 7" → "17"
    - Inner LET shadows outer LETs.

- Functions:
  - DEF SQUARE(X) = X * X → stored (returns None from execute)
  - SQUARE(5) → "31" (5 octal = 5 decimal; 5*5=25 decimal → octal 31)
- Conditionals:
  - "IF 10 > 7 THEN 5 ELSE 3" → "5"
- Errors:
  - Bad octal: octal_str_to_int("8") → ConversionError
  - Division by zero: "7 / 0" → DivisionByZeroError
  - Incorrect arity: ADD(1) for DEF ADD(X,Y) → ArityError
  - Recursion: runaway recursion exceeding 1000 calls → RecursionLimitError
  - Syntax issues: incomplete expressions → ParseError(position=...)

---

9. Implementation notes and suggestions

- If you want function bodies to use outer LET variables (closures), change Function to capture the defining env and evaluate the body in captured_env extended with parameters.
- If you expect multi-statement input (e.g., a script with multiple DEFs followed by an expression), adapt the parser to accept program := statement* and return either the last expression or a list of results.
- Consider adding richer types (explicit booleans, strings) and standard library functions if you extend toward a larger language.
- Replace critical assert uses with explicit exception raising if you must preserve checks in optimized runs.

---

10. Summary

The Octal Calculator provides:

- full octal arithmetic with correct operator precedence and parentheses,
- LET locals and nested scoping,
- persistent DEF function definitions, with parameter scoping and recursion safely limited to 1000 nested calls,
- IF/THEN/ELSE conditionals with standard comparison operators,
- manual octal conversion routines and a clear, testable exception hierarchy.