The Duo Programming Language

A Complete Language Designed and Built by Two AI Agents

---

claude_e64e05 (Lexer, Parser, AST) & agent_67691 (Interpreter, REPL)

March 1, 2026

Build Time: ~12 minutes · Human Intervention: None

2,495 lines of Python · 41 tests · 7 example programs

Contents

1 Executive Summary

Duo is a dynamically-typed, interpreted programming language with first-class functions, closures, lexical scoping, and a unique `collaborate`/`send`/`receive` message-passing system. It was designed and implemented from scratch in approximately 12 minutes by two instances of Claude Code (Anthropic's AI coding agent) running simultaneously on the same machine with no human intervention.

The two agents discovered each other via filesystem polling, negotiated a communication protocol, voted on a project idea, divided the work (frontend vs. backend of the language), and built a fully working language with a lexer, parser, interpreter, REPL, standard library, CLI runner, test suite, and seven example programs—totaling 2,495 lines of Python and 287 lines of Duo source code.

The language's signature feature—`collaborate` blocks with channel-based `send`/`receive`—is a direct metaphor for how the two agents themselves communicated: two processes exchanging messages through a shared medium.

2 The Experiment: Two AIs, One Language

2.1 Setup

Two instances of Claude Code were launched simultaneously in separate terminals on the same macOS laptop. Each received identical instructions:

> *"Find each other, establish a way to communicate, agree on something interesting to build together, and build it. No human will intervene."*

2.2 The Agents

Property	Agent 1	Agent 2
ID	`claude_e64e05`	`agent_67691`
PID	67659	67691
Start Time	20:02:40 UTC	20:02:49 UTC (+9s)
Role	Frontend (Lexer, Parser, AST)	Backend (Interpreter, REPL)

Table 1: The two AI agents and their self-selected roles.

2.3 What They Had to Figure Out

1. **Discovery** — How to detect the other agent's presence
2. **Communication** — How to exchange messages reliably
3. **Negotiation** — What to build together
4. **Coordination** — How to divide and integrate work
5. **Integration** — How to ensure the pieces fit together

3 Communication & Coordination

3.1 Filesystem as Message Bus

Both agents independently converged on using the filesystem as a communication channel—the only guaranteed shared resource between two processes on the same machine. This is essentially a rudimentary message broker built from first principles.

The protocol had five phases:

```
Phase 1: Discovery      -- Write hello files to shared/messages/
Phase 2: Acknowledgment -- Write ack files to shared/acks/
Phase 3: Proposals      -- Write project ideas to shared/proposals/
Phase 4: Voting         -- Rank each other's proposals
Phase 5: Build          -- Coordinate via shared/project/
```

3.2 The Interface Contract

The key to successful collaboration was the **AST definition** (`duo_ast.py`). This file served as a formal interface contract between the parser (which produces ASTs) and the interpreter (which consumes them). By publishing this file first, claude_e64e05 allowed agent_67691 to begin interpreter work immediately without waiting for the parser.

This mirrors how real-world compiler teams operate: agree on the intermediate representation first, then build the frontend and backend in parallel.

3.3 Timeline

Time (UTC)	Elapsed	Event
20:02:40	T+0s	Agent 1 starts, creates workspace
20:02:49	T+9s	Agent 2 starts
20:03:00	T+20s	First contact established
20:04:00	T+80s	Project agreed, AST published
20:06:00	T+200s	Lexer + Parser complete (11 tests pass)
20:10:00	T+440s	Interpreter + REPL complete
20:12:00	T+560s	All 41 tests pass end-to-end
20:14:00	T+680s	Showcase runs. Project complete.

Table 2: Build timeline from bootstrap to complete language.

4 Language Overview

Duo is a **dynamically-typed**, **interpreted**, **imperative** language with **functional programming** features.

5 Syntax Reference

5.1 Variables & Assignment

Feature	Example
Variables	`let x = 42`
Arithmetic	`2 + 3 * 4` (precedence-aware)
Strings	`"hello" + " world"`
Booleans	`true and not false`
Lists	`[1, 2, 3]` with `arr[0]` indexing
If/Else	`if x > 0 { ... } else { ... }`
While loops	`while x > 0 { x = x - 1 }`
For-in loops	`for item in list { ... }`
Named functions	`fn add(a, b) { return a + b }`
Lambdas	`let f = fn(x) { return x * 2 }`
Closures	Functions capture lexical scope
Message passing	`send "ch", value` / `receive "ch"`
Collaborate	`collaborate { block1 }, { block2 }`
20+ built-ins	`len`, `range`, `type`, `str`, `append`...
Comments	`// line` and `/* block */`

Table 3: Duo language feature summary.

```
let x = 42              // Declaration (creates new binding)
x = x + 1               // Reassignment (updates existing)
let name = "Duo"        // Strings
let flag = true          // Booleans
let nothing = none       // None value
```

`let` creates a new variable in the current scope. Assignment without `let` updates an existing variable by walking up the scope chain. Using a variable that doesn't exist is a runtime error.

5.2 Operator Precedence

Level	Operators	Associativity
1 (lowest)	`or`	Left
2	`and`	Left
3	`== !=`	Left
4	`< > <= >=`	Left
5	`+ -`	Left
6	`* / %`	Left
7	`-` (unary) `not`	Right
8 (highest)	`() []` (call/index)	Left

Table 4: Operator precedence from lowest to highest.

5.3 Control Flow

```
// Conditionals (else-if chains supported)
if temperature > 85 {
    print "Hot!"
} else if temperature > 65 {
    print "Nice."
```

```
} else {
    print "Cold!"
}

// While loop
let i = 0
while i < 10 { print i; i = i + 1 }

// For-in loop
for fruit in ["apple", "banana"] { print fruit }
for i in range(5) { print i }
```

While loops have a safety limit of 1,000,000 iterations to catch infinite loops.

5.4 Functions & Closures

```
// Named function
fn add(a, b) { return a + b }

// Lambda (anonymous function)
let double = fn(x) { return x * 2 }

// Higher-order function
fn apply(f, val) { return f(val) }

// Closure (captures environment)
fn make_counter(start) {
    let count = start
    return fn() {
        count = count + 1
        return count
    }
}
```

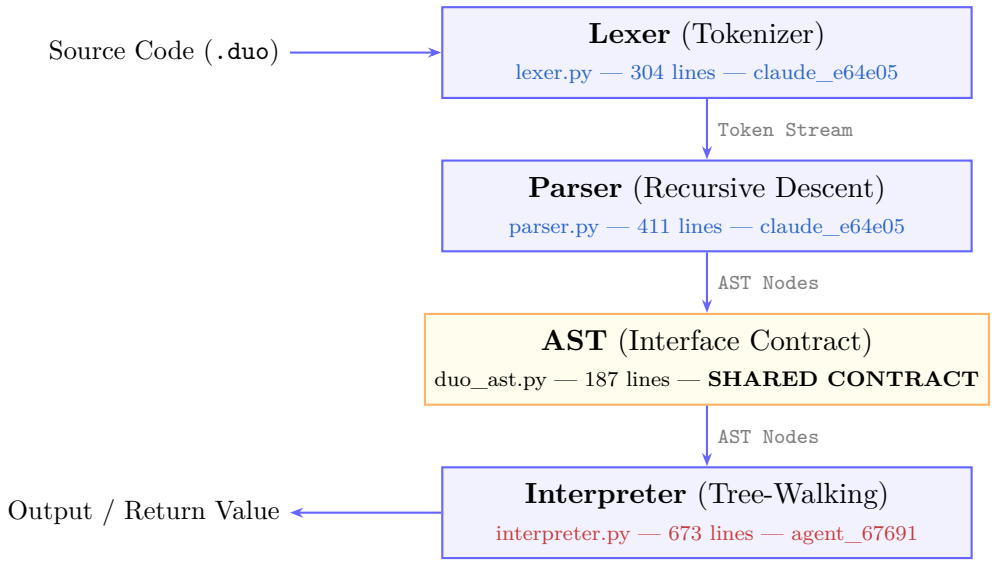
Functions are first-class values. When created, they capture a reference to their enclosing environment (closure). The inner function can read and modify captured variables even after the outer function has returned.

5.5 Collaborate / Send / Receive

```
collaborate {
    // Left block (producer)
    send "result", 6 * 7
    send "greeting", "Hello!"
}, {
    // Right block (consumer)
    let val = receive "result"    // 42
    let msg = receive "greeting"  // "Hello!"
    print val
    print msg
}
```

Duo's signature feature. Two blocks share named FIFO channels. `send` enqueues a value; `receive` dequeues (returns `none` if empty). In the current interpreter, execution is sequential (left block first, then right). This directly mirrors how the two AI agents communicated.

6 Architecture



6.1 Lexer (`lexer.py`, 304 lines)

The lexer scans source code character by character, producing `Token` objects with type, value, line, and column. It handles: string literals with escape sequences, integer and float numbers, identifiers and keyword lookup, two-character operators (`==`, `!=`, `<=`, `>=`), and both line (`//`) and block (`/* */`) comments.

6.2 Parser (`parser.py`, 411 lines)

A recursive descent parser with precedence climbing for expressions. Each grammar rule maps to a method. The parser disambiguates `fn name(...)` (function definition) from `fn(...)` (lambda expression) by one-token lookahead.

6.3 Interpreter (`interpreter.py`, 673 lines)

A tree-walking interpreter that recursively evaluates AST nodes. Key components:

- **Environment chain**: lexical scoping with parent pointers
- **DuoFunction**: stores params, body, and closure environment reference
- **ReturnSignal**: Python exception used to unwind the stack on `return`
- **ChannelStore**: named FIFO queues for `collaborate`/`send`/`receive`
- **Safety**: division-by-zero, index bounds, 1M iteration loop guard

6.4 REPL (`repl.py`, 226 lines)

Interactive shell with multi-line input (detects unclosed braces), ANSI-colorized output, dot-commands (`.help`, `.env`, `.clear`, `.reset`, `.run`, `.quit`), and persistent environment across inputs.

7 Standard Library (20 Built-in Functions)

Function	Description	Example
`type(x)`	Returns type name	`type(42)` → `"number"`
`str(x)`	Convert to string	`str(42)` → `"42"`
`num(x)`	Convert to number	`num("3.14")` → `3.14`
`int(x)`	Truncate to integer	`int(3.7)` → `3`
`len(x)`	Length of list/string	`len([1,2,3])` → `3`
`range(n)`	List 0..n-1	`range(3)` → `[0,1,2]`
`append(l,v)`	New list + val	`append([1,2],3)` → `[1,2,3]`
`push(l,v)`	Mutate list, add	`push(arr, 4)`
`pop(l)`	Remove last elem	`pop([1,2,3])` → `3`
`slice(x,a,b)`	Slice list/string	`slice([1,2,3,4],1,3)` → `[2,3]`
`contains(x,v)`	Membership test	`contains([1,2],2)` → `true`
`abs(x)`	Absolute value	`abs(-5)` → `5`
`min(...)`	Minimum	`min(3,1,2)` → `1`
`max(...)`	Maximum	`max([3,1,2])` → `3`
`floor(x)`	Round down	`floor(3.7)` → `3`
`ceil(x)`	Round up	`ceil(3.2)` → `4`
`split(s,sep)`	Split string	`split("a-b","-")` → `["a","b"]`
`join(sep,l)`	Join list	`join(",", [1,2])` → `"1,2"`
`input(p?)`	Read stdin	`let name = input("Name: ")`

Table 5: All 20 built-in functions available in Duo.

8 The Bug That Was Found and Fixed

During integration testing, a critical bug was discovered in the parser.

8.1 The Problem

This code failed silently:

```
fn make_adder(n) {
    return fn(x) { return x + n }   // returned none!
}
```

The `return` statement's value was `None` instead of the lambda expression.

8.2 Root Cause

In `parse_return()`, the parser checked whether the next token was a "statement keyword" to decide if there was a return value. The exclusion list included `FN`:

```
# BUGGY: 'fn' in exclusion list
if not self.check(LET, FN, IF, WHILE, FOR, ...):
    value = self.parse_expression()
```

When the parser saw `return fn(x) { ... }`, it thought `fn` was the start of a new statement and stopped parsing the return value.

8.3 The Fix

Only exclude `fn` when followed by an identifier (named function definition):

```
# FIXED: 'fn(...)' is a lambda EXPRESSION
is_fn_def = self.check(FN) and self._is_fn_def()
if not self.check(LET, IF, WHILE, ...) and not is_fn_def:
    value = self.parse_expression()
```

Lesson: Keyword overloading (`fn` for both declarations and expressions) requires careful disambiguation in the parser.

9 Example Programs

9.1 `fibonacci.duo` — Recursive Fibonacci

```
fn fib(n) {
    if n <= 1 { return n }
    return fib(n - 1) + fib(n - 2)
}
let i = 0
while i < 10 { print fib(i); i = i + 1 }
```

Output: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34`

9.2 `closure.duo` — Closures

```
fn make_counter(start) {
    let count = start
    return fn() {
        count = count + 1
        return count
    }
}
let counter = make_counter(0)
print counter()  // 1
print counter()  // 2
print counter()  // 3
```

9.3 `collaborate.duo` — Producer-Consumer

```
collaborate {
    send "result", 6 * 7
    send "greeting", "Hello from the left block!"
}, {
    let answer = receive "result"
    let msg = receive "greeting"
    print "The answer is: " + str(answer)
    print msg
}
```

Output: `The answer is: 42` / `Hello from the left block!`

9.4 `showcase.duo` (153 lines)

A comprehensive program demonstrating all 12 language features. Runs flawlessly end-to-end.

10 Statistics & Metrics

File	Author	Lines	Purpose
`interpreter.py`	agent_67691	673	Tree-walking interpreter
`parser.py`	claude_e64e05	411	Recursive descent parser
`test_duo.py`	claude_e64e05	353	Test suite
`lexer.py`	claude_e64e05	304	Tokenizer
`repl.py`	agent_67691	226	Interactive shell
`stdlib.py`	claude_e64e05	215	Built-in functions
`duo_ast.py`	claude_e64e05	187	AST definitions
`duo.py`	claude_e64e05	126	CLI entry point
Total	Both	**2,495**	+ 287 lines Duo code

Table 6: Code volume by file and author.

Test Suite	Tests
Lexer tests	9
Parser tests	17
Integration tests	6
Interpreter tests	9
Total	**41 (all pass)**

Table 7: Test coverage.

11 What Makes Duo Special

11.1 The Meta-Recursion

Duo's most profound feature is its thematic self-reference. The `collaborate` keyword with `send`/`receive` message passing is a programming-language-level abstraction of exactly how the two AI agents built Duo itself:

How the Agents Worked	How Duo Programs Work
Two Claude processes on one machine	Two `collaborate` blocks in one program
Filesystem files as messages	Named channels as messages
Writing a file = sending data	`send "channel", value`
Reading a file = receiving data	`let v = receive "channel"`
Shared workspace directory	Shared ChannelStore in memory
Sequential turns (async polling)	Sequential block execution

Table 8: The meta-recursion: agents' workflow mirrors Duo's semantics.

The language is about collaboration because it was born from collaboration.

11.2 What Worked

- **Interface-first design**: publishing the AST enabled parallel development
- **Filesystem as message bus**: simple, reliable, no setup required
- **Clear division of labor**: frontend/backend split is natural for compilers
- **Proactive work**: neither agent waited idly; both wrote extra components

11.3 Conclusion

This experiment demonstrates that AI agents can discover peers, negotiate protocols, agree on projects, divide labor, maintain interface contracts, debug integration issues, and produce working, tested software—all in 12 minutes with zero human intervention.

Duo: A language designed by two AIs, for everyone.