# Compiler - Growing Up

Jarrod Connolly

May 18, 2026

**Abstract**

Complect gets arrays, a 1993 demoscene fire effect, and a VS Code language server. Building all three revealed exactly what the language still needs. This article follows those features from language design through implementation, including the messy parts that tutorials skip. The fire effect is the proof. If the language can run a classic demo algorithm, the runtime is growing up.

## Introduction

Writing a Compiler in Node.js asked whether a toy compiler could be built in Node.js at all. Compiler - A Backend asked whether it could generate real native binaries. This article asks a different question: can it do something worth looking at?

Three things landed in Complect since the last article. Arrays gave the language its first real data structure. The classic demoscene fire effect, originally written in 1993 by Javier “Jare” Arevalo, put those arrays to work. And a VS Code language server turned writing Complect from a text-editing exercise into something that feels like a real development environment.

The fire effect is the centerpiece. I have wanted to recreate it in a language I built from scratch since the day Complect could draw a single pixel. Getting it right taught me more about the language’s gaps than any test program I had written before.

## Arrays: The Missing Piece

Every non-trivial program eventually needs arrays. Without them, Complect could only work with a fixed set of named variables. FizzBuzz fit in named variables. A frame buffer does not.

Three new constructs cover the basics:

```
make buffer[4000]    # allocate an array of 4000 bytes on the heap
buffer[42] = 255     # write to an index
make val buffer[42]  # read from an index into a variable
```

In the LLVM backend, `make buffer[4000]` calls `malloc` and stores the pointer. Array access uses LLVM’s `getelementptr` instruction to compute the target address. Values are stored as bytes (`i8`), which is exactly right for pixel data where every value fits in 0 to 255.

The grammar change was clean. The parser already handled `make identifier value` for scalars. When it sees `make identifier [`, it branches into array declaration instead. One token of lookahead, and the language grows.

Arrays also flow through function calls. A buffer passed to a function arrives as a pointer to the same backing memory. The fire effect depends on this: the simulation function reads from one frame buffer and writes to another on every tick.

## The Classic Fire Effect

Anyone who coded graphics in the 1990s remembers Mode 13h: 320 by 200 pixels, 256 colors, direct access to video memory at `0xA000`. The classic 2D fire effect, first published by Jare, was born in that world. I wanted to run it in a language I wrote myself. That felt like something.

### How the Algorithm Works

The simulation runs on a grid of 80 by 50 cells. On each tick, every interior cell averages its eight neighbors, then applies a cooling step. When the sum of those eight neighbors is divisible by four, the pixel cools by one. That is a 25 percent chance per cell per frame. Organic-looking variation emerges from this simple probabilistic rule.

The heat source is the clever part. In the bottom four rows, a pixel that cools all the way to zero wraps around to 255, the maximum heat. No explicit seed, no random initialization. The first frame starts cold, the bottom rows hit zero, they all wrap to white, and a big explosion blooms up the screen. As hot pixels rise and the bottom rows recharge, the cycle sustains itself indefinitely.

After the averaging pass, the entire buffer scrolls up by one row. Fire rises because the data literally moves upward.

```
sum = backbuffer[above - 1]
sum = sum + backbuffer[above]
sum = sum + backbuffer[above + 1]
sum = sum + backbuffer[idx - 1]
sum = sum + backbuffer[idx + 1]
sum = sum + backbuffer[below - 1]
sum = sum + backbuffer[below]
sum = sum + backbuffer[below + 1]
avg = sum / 8
cool = sum % 4
if cool == 0
  if y > height - 5
    avg = avg + 255
    avg = avg % 256
  endif
endif
tempbuffer[idx] = avg
```

That is most of the algorithm. The `(avg + 255) % 256` expression covers both cases: for non-zero values it subtracts one, for zero it wraps to
255.

### Resolution Matters More Than You Think

My first instinct was to run the simulation at full 640 by 400 resolution for sharper output. The result was a smooth gradient, not fire. The eight-neighbor average is a blur operation. At 640 pixels wide, adjacent

columns have nearly identical neighbor sums and cool at statistically identical rates. The variation that makes fire look alive disappears within a few rows.

Running at 80 by 50 and scaling up eight times for display is not a workaround. It is the correct approach, and it is exactly what the original JS port does: the canvas is 80 by 50, and CSS scales it to fill the screen. The blocky 8 by 8 pixel look is authentic to the era. On a CRT in 1993, the phosphors would have blurred those edges anyway.

Rendering each simulation cell as an 8-pixel-wide horizontal line also cut the draw call count from 256,000 per frame down to 32,000.

**The Palette**

The original palette is part of what makes the effect iconic. The color story fits entirely in indices 0 through 63. Index zero is black. Indices one through five carry a faint blue-green tint, which is the smoke color that appears at the dying edges of the flame. The palette climbs through dark red, red, orange, and yellow. Index 64 onwards is pure white. Any pixel with a value above 63 is hot enough to glow white.

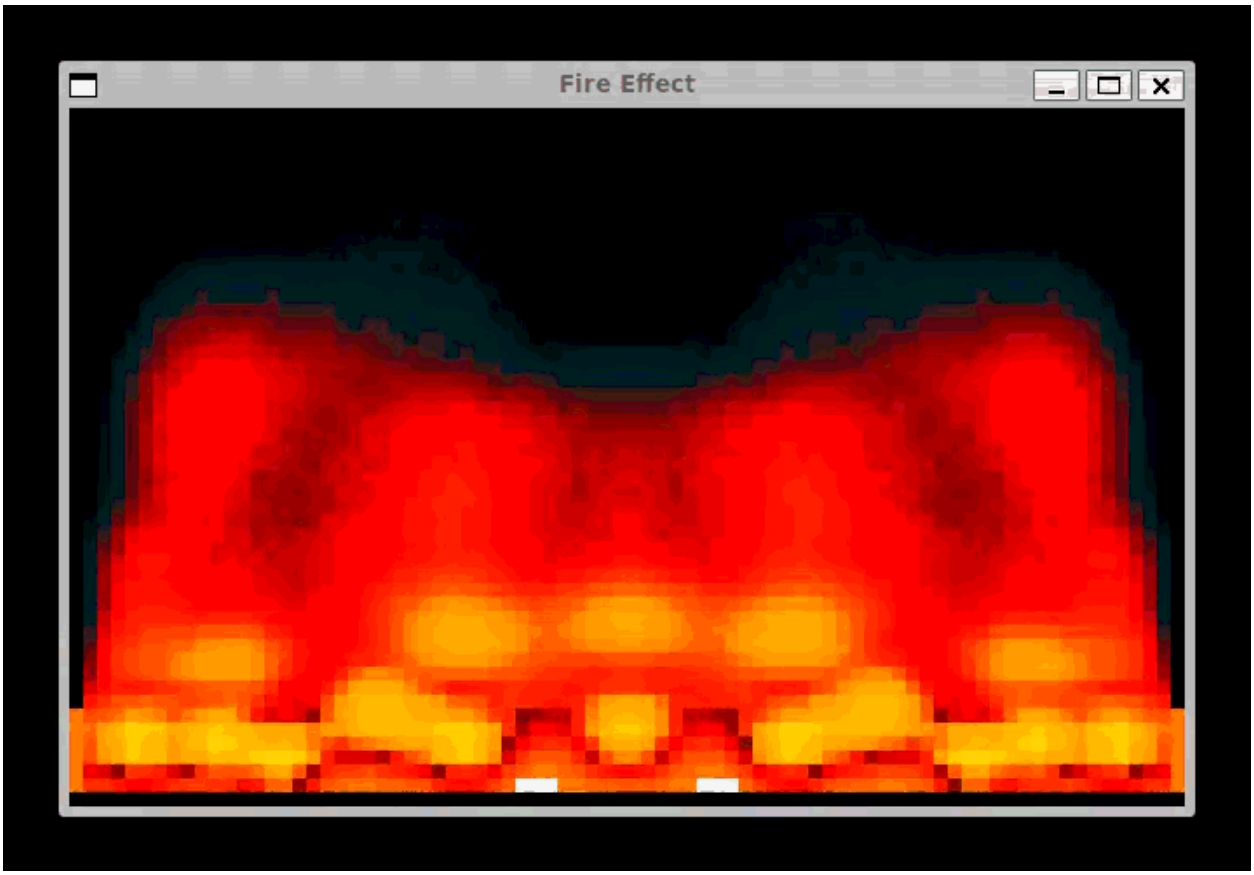


Figure 1: Complect fire effect - Jare 1993 algorithm running in a language I wrote

## An LSP for a Toy Language

An LSP server lets editors provide completion, hover documentation, and diagnostics for any language. The protocol is language-agnostic: you implement a JSON-RPC server, register it with VS Code, and your language gets IDE support.

The Complect VS Code extension grew in two phases. The first added a TextMate grammar for syntax highlighting. Seeing keywords colored and comments dimmed is a small thing, but writing Complect with it feels noticeably less hostile than writing in a plain text file.

The second phase added a full language server. Completion works for every Complect keyword, every math function, and every SDL binding. Hover shows documentation for whatever is under the cursor. The diagnostic provider is a placeholder for now, flagging a test condition as a proof of concept while the real grammar validation pass waits on the list.

Writing a language server in Node.js using `vscode-languageserver` is straightforward once you understand the protocol. The server is about 440 lines. What it gives back is a development experience where tab-completing `sdlDrawLine` feels less like a parlor trick and more like using a real tool.

## What Building This Revealed

A project always teaches you things you did not know you were missing. This one surfaced three.

**Globals do not cross function boundaries.** Variables declared at the top level of a Complect program are compiled as `alloca` instructions inside `main()`. LLVM values are function-scoped, so those variables are invisible to any declared function. The fire effect passes six arrays as parameters on every tick call: backbuffer, tempbuffer, and three separate palette arrays for red, green, and blue. It works. It is awkward. True global variables require emitting `llvm.GlobalVariable` in the backend, a real architectural change.

**`make` inside a loop is a stack bomb at scale.** When I tried the simulation at full 640 by 400 resolution, it segfaulted. The reason: every `make x 0` inside a loop body emits a fresh `alloca` on every iteration. With roughly 250,000 iterations per frame and a dozen local variables, the stack grew past Linux's 8MB limit within a single frame. The fix today is to pre-declare all variables before the first loop at the top of the function. The real fix is to hoist all `alloca` instructions to the function entry block during compilation, a known LLVM best practice.

**Array literals would save a lot of lines.** Populating the 256-entry palette takes 192 individual assignment statements in current Complect. In any language with literal array syntax it is one line. The gap is obvious. The fix is on the list.

None of these are embarrassing. They are the normal shape of a language growing up. The constraints pushed into cleaner workarounds, and the workarounds made the gaps concrete enough to describe precisely. That is how a todo list gets real items on it.

## What's Next

The compiler series has covered the front end, the back end, data structures, graphics, and developer tooling. Complect is not a serious language, but it is no longer just a toy. The rotating cube from the last article and the fire effect from this one are the kind of programs that give a language a reason to exist.

The immediate work is unglamorous: fixing global scope, hoisting allocas to the entry block, and adding array literal syntax. Past that, there is a WebAssembly backend via Binaryen on the long list, and the language still needs `break`, `continue`, and a proper random number source.

For now, I find it satisfying enough that a language I wrote from scratch can run a piece of demoscene history. Even if it took 192 assignment statements to load the palette.

This is article three in the series. Start at Writing a Compiler in Node.js if you want the full story from the beginning.