TinyC for the SD-8516

Introduction

> This is not intended to be a general purpose programming language. It is intended to be the language I write the general purpose programming language in. However, I keep running into bugs. I am not sure if I should keep improving it or whether it's time to re-write it.

TinyC refers to V1 (tinyc.asm) and V2 (tinyc.c), which is the same as V1 but written in C. V1 is intended to compile V2, and then we have self-hosting C. Thus, this TinyC should be more like the “B” language in it's feature set, and in the idea that it is intended to compile a version of itself written in C. From there, we can iterate on the C compiler and make it better.

V1 Current Feature Set:

- Expressions with precedence
- Local variables
- if/else/else if
- Comparison operators
- while
- break/continue
- Multiple functions + function calls
- Function arguments
- Pointers (read, write, address-of)
- Array indexing
- char* (8-bit access)
- String literals
- Hex literals
- Character literals
- Comments (//)
- Global variables
- Builtins (putchar, getchar, halt, yield)

This is enough that I was able to start writing functions in C like abs, min, max, strlen, strcmp, atoi/itoa, and so on.

Function Library

- List of TinyC built-in functions

History

A self-hosting C compiler had been my goal for quite some time. But I didn't know how to write one. I hacked out a BASIC first, then followed some guides and source code to get a Forth running. But C

eluded me for several months. Eventually I was able to leverage what I had learned doing BASIC to get a parser that could compile int main(void) { return 0; } and emitted LDA #0 and RET. The debugger was simply printing the return value. The rest of the story is found in the TinyC Developer Diary. Mainly so I don't forget what I learned, as this project has gotten quite big.

Issues

The big issue is that I don't *really* know what I am doing. Meaning, it's really no issue to mechanically process a script and write an interpreter or compiler for it, if you *just start coding*. The real issues will be operational, later on. Here, the issue is that the compiler doesn't know where to compile itself to. TinyC v1 lives in bank 1, TinyC v2 lives in bank 3 (because that's where it was compiled). So should it compile to bank 1? Well, originally it wasn't written this way but when you talk about it like this the solution seems easy; have an int variable such as **target_bank** and we just compile into that. What would really be good is to compile into an area but the code itself is targeted for a different bank.

What is even better is if I could write relocatable code. You know, simply writing this out all gives me ideas. Relocatable code! Could be interesting. Ok let's write down some ideas– Relocatable Code

Compile to disk

The other solution I am mulling is to rewrite the emitter. There's no reason it has to output to the exact location it will run at. It could output the bytes to a file instead.

What's Missing?

- Preprocessor: no #include, #define, #ifdef, macros, conditional compilation. Every program is monolithic.

Structs and unions: cannot define composite types. No member access (., →).

- Typedef: no type aliases.
- Multi-dimensional arrays and array declarations: int arr[10] doesn't exist; arrays only as pointers.
- Initializers: int g = 5; parses but doesn't run. No aggregate initializers {1, 2, 3}.
- String literals as data: strings work only as immediate operands, not as initialized arrays.
- Function pointers: no syntax, no calling convention.
- Variable arguments: no ..., no va_list.
- enum: no constant enumeration.
- static, extern, register, volatile, const: storage class and qualifier keywords missing.
- Multiple return types: only int (well, 24-bit). No void, char, short, long, float, double returns.
- void type: parsed in (void) parameter list only; can't declare void variables, return void, or use void

Operator gaps

- Bitwise ops are critical for systems programming. Without them, no masking, no flag

manipulation.
- Logical && and || with short-circuit semantics matter for correctness, not just convenience.
- ++/– are syntactic sugar but their absence is glaring.
- % (modulo) for any non-trivial arithmetic.
- Compound assignment is just sugar but expected.

Library

- No standard library at all. No printf, malloc, strcmp, memcpy, strlen. Programs must be self-contained or call only the six builtins.
- No file I/O from compiled programs (peek/poke don't qualify).
- No dynamic memory.

Compiler quality

- No optimization passes. Every operation goes through BLA via the stack.
- No type checking. Assigning char * to int is silent.
- No warnings about unused variables, missing returns, narrowing conversions.
- Error messages are minimal (“undefined variable” without line number context).
- No symbol scoping beyond function-local — all locals share one flat namespace within a function.

Language semantics

- Pointer arithmetic doesn't scale by element size: int *p; p + 1 adds 1 byte, not 3.
- sizeof would be needed to write portable code; absent.
- Char arithmetic doesn't promote to int per C rules.
- No automatic conversions, casts, or type coercion.

Roadmap

This is not a roadmap.

Easy

- Bitwise operators.
- ++/– and compound assignment.
- % operator; opcode already exists in the CPU, just needs parser.
- Pre-processor. Separate pass before lexing; probably harder than it sounds.

Medium

- &&/|| with short-circuit. Needs branch generation.
- Local arrays. int arr[10] allocates 30 bytes on stack, needs frame offset adjustment and array-as-pointer-decay.

Hard/Time-consuming/Don't know

- Initialized globals. This is, unfortunately, non-trivial :(It requires either an init phase before main or compile-time init.
- Standard library subset – strlen, strcmp, memcpy, printf (formatted output is a project of its own).
- Type checking. Proper type expression evaluation, conversion rules.
- Structs. Type representation grows from 1 byte to multi-byte type descriptor.