TinyC for the SD-8516

Introduction

This TinyC is 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.

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.

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.

Expression Parser

I did expressions first because I had already done a parser for BASIC. I will go into some detail here over how I did it. First, it's almost exactly the same as in Stellar BASIC. Stellar BASIC's `parse_expression` calls `parse_term` which calls `parse_factor`. An identical structure with identical precedence handling. The only difference is that BASIC evaluates expressions immediately (pushing results onto a value stack), while TinyC emits machine code that will perform the evaluation

later at runtime. The parser shape is the blueprint. The big dif is that this was compiled, so I had to emit code versus just evaluating everything into the accumulator/TOS.

Simplest explanation possible

The simplest and easiest way is to imagine everything has brackets. So an expression like 2 + 3 * 4 is required to be in the form (2 + (3 * 4)). What the parser has to do is evaluate every () expression. This is done recursively and is the recursive part of “recursive descent parser”. But not everything has brackets. You just imagine it does, because you evaluate the left term first then pass the right term down in terms of precedence (“precedence descent” is the “descent” in “recursive descent parser”).

> It was the early days of computer science. 1958 at MIT. John McCarthy was a Mathematician, not a “Computer Scientist”. He wanted to represent logic and math on the computer. This is how LISP was invented. First, they made + the name of a function. They said, + is a function that takes two parameters. That makes sense. And, you could just say “add” for clarity, since we want the computer to serve us English speaking humans. So
>
> ```
> 2 + 3 * 4
> ```
>
> became
>
> ```
> (2 + (3 * 4))
> ```
>
> Now each bracket has a left and a right and an operand. One operation. So let's re-write this as parameters to function calls, because thats how computers work (ex. `ADD A, B`):
>
> ```
> (+ 2 (* 3 4))
> ```
>
> And lets use English to name the functions, as we agreed earlier:
>
> ```
> (add 2 (add 3 4))
> ```
>
> Now let's move the bracket to after the function name, which follows principles of English language (ex. we say “I am not hungry” not, “I am hungry not” because we don't want to introduce out of context information:
>
> ```
> add(2 add(3 4))
> ```
>
> Now separate parameters with commas, which is another principle of English:
>
> ```
> add(2, add(3,4))
> ```
>
> This is the link between Stack Machines, Forth, LISP, BASIC and C. C is that moment where we deal with things in English, but they're still at the bare metal in terms of being able to parse them directly into machine code.

So the first kind of parser you should write has a pre-processor that encases everying in parentheses. Then brute force it; just scan for ()'s and emit the innermost functions first. It would end up looking like (2 + [1]. This makes parsing robotic and easy.

There's nothing wrong with parsing like that. Its a good way to emit code. It's reliable and safe.

Everything which follows is merely a refactor of a refactor of a refactor going back to the early days of 1950s computer science when all this stuff was originally invented.

Three-Level Descent

The expression parser we use is exactly similar to the above but does it using special rules. It evaluates the left term, then finds an operator, then passes the entire right side down to the next level. This way, higher precedence operators bind their operands before lower-precedence operators get a chance to. As it recursively unravels, everything eventually gets parsed and evaluated.

Level	Function	Operators	Calls Down To
Comparison	`parse_expr`	== != < > ⇐ >= etc.	`parse_additive`
Additive	`parse_additive`	+ -	`parse_term`
Multiplicative	'parse_term	* /	`parse_primary`
Primary	`parse_primary`	numbers, variables, `()`, unary -	none (equates to value)

Example; 2 + 3 * 4

- 1. `parse_expr` calls `parse_additive`
- 2. `parse_additive` calls `parse_term`
- 3. `parse_term` calls `parse_primary`, emits `LDA #2`
- 4. `parse_term` checks for * or `/`, sees +, *not its operator,* returns
- 5. `parse_additive` sees +, saves left operand, emits `PUSH A`
- 6. `parse_additive` calls `parse_term` for the right side
- 7. `parse_term` calls `parse_primary` and emits `LDA #3`
- 8. `parse_term` sees * saves left operand, emits `PUSH A`
- 9. `parse_term` calls `parse_primary` and emits `LDA #4`
- 10. `parse_term` emits `MOV B, A` / `POP A` / `MUL A, B`, A now holds 12
- 11. `parse_term` checks for more * or `/`, sees end, returns
- 12. `parse_additive` emits `MOV B, A` / `POP A` / `ADD A, B`, A now holds 14
- 13. `parse_additive` checks for more + or `-`, sees `;` then returns

```
  2 + 3 * 4    --->    2 (3+4) +
```

The parser puts the 2 in A then evaluates whatever is in the parentheses, then adds them. The parentheses aren't there. The compiler imagines they are there by considering “whatever is left” as one expression. Therefore, “parse expression”, by its very nature of considering an entire expression, considers the entire expression as if it were surrounded by parentheses.

The key insight comes from step 4: `parse_term` returns immediately when it sees + *because addition is not its concern.* This forces the + to be handled one level up, as if the multiplication was surrounded by parentheses.

Emits

Every operator follows the same code generation template. The left operand is already in register A (from the recursive call). The compiler saves it, evaluates the right operand (which also lands in A), then combines them:

```
  PUSH A          ; save left operand on stack
  <code here>     ; evaluate right operand into A
  MOV B, A        ; move right operand to B
  POP A           ; restore left operand to A
  ADD A, B        ; A = left + right (or SUB, MUL, DIV)
```

This looks inefficient but remember each individual lego block puts it's result into A. So we need to move A around a bit. Actually we can do this:

```
  MOV B, A        ; move left operand to B
  <code here>     ; evaluate right operand into A
  ADD A, B        ; A = right plus left (or SUB, MUL, DIV)
```

With a little extra code we can emit into B. A later optimization perhaps.

```
  <code here>     ; evaluate right operand into B
  ADD A, B        ; A = left + right (or SUB, MUL, DIV)
```

This pattern is uniform across all arithmetic operators. The only difference is the final instruction: `ADD`, `SUB`, `MUL`, or `DIV`.

From the C version

The additive level in the C version illustrates the pattern:

```
int parse_additive() {
    parse_term();                   // left operand = A
    if (g_error) { return 0; }
    while (1) {
        if (g_tok_type == TOK_PLUS) {
            next_token();           // consume ''+''
            emit_push_a();          // save left
            parse_term();           // right operand = A
            emit_mov_b_a();         // B = right
            emit_pop_a();           // A = left
            emit_add_a_b();         // A = left + right
        } else if (g_tok_type == TOK_MINUS) {
            next_token();
            emit_push_a();
            parse_term();
            emit_mov_b_a();
```

```
            emit_pop_a();
            emit_sub_a_b();
        } else {
            break;
        }
    }
    return 0;
}
```

The multiplicative level is structurally identical, with `parse_primary` replacing `parse_term` and `MUL`/`DIV` replacing `ADD`/`SUB`.

The B in BEDMAS

Brackets in expressions (parentheses) override precedence by recursing back to the top level. When `parse_primary` encounters `(`, it calls `parse_expr`, the full expression parser, including all precedence levels. This means `(2 + 3) * 4` evaluates the addition first because `parse_primary` fully resolves the parenthesized sub-expression before returning to `parse_term`, which then multiplies by 4.

```
// Inside parse_primary:
if (g_tok_type == TOK_LPAREN) {
    next_token();           // consume '('
    parse_expr();           // full precedence chain
    expect(TOK_RPAREN);     // consume ')'
    return 0;
}
```

Conclusion

The parser turns the whole (expression) into one number. The expression parser is a big simplification loop. Chunk by chunk, we simplify everything as if it was surrounded by parentheses. The big reveal is that all of this started by writing a parser that per-processed expressions by surrounding the left and right side of every operator in parentheses, and the “three level recursive descent parser” is a refactoring of that. So if you don't understand it how it is now, go back to surrounding everything in parentheses. It will work exactly the same.

Comparisons (<, ==, >)

Comparisons sit at the lowest precedence level, handled by `parse_expr` itself. Unlike arithmetic operators, comparisons can not loop. `a < b < c` is not valid chaining in C. The parser checks for a comparison operator once, evaluates both sides, then emits a compare-and-set sequence that leaves 1 (true) or 0 (false) in register A.

The emitted code for `a == 5` is simple. It kind of reminds me of forth; imagine if 'A' was the TOS (top of stack). We compare the value with whats in A, then we push a 1 or 0 (true/false) variable onto the

TOS for evaluation by an IF.

```
    (load a value into A)
    LDB #5              ; load value to check against
    CMP A, B            ; sets CPU flags
    JZ true_case        ; jump if equal
    LDA #0              ; false
    JMP done

true_case:
    LDA #1              ; true
done:
    ; A = 0 or 1
```

Symbol table

Arguably the symbol table is the core milestone. I had been working on various variable storage systems until this, and this is, I think, the best one so far.

All I wanted from a variable system was the ability to map a string to a string. That was the core idea. If I could do that I could have a variable system. So when the programmer writes `int x = 5;` on one line and `return x;` three lines later, the compiler needs to set the string “x” to refer to the string “5”. This is what the NVAR system does, and it was easy to implement. I just created a record system and scanned for the variable names from the start of record-space.

But in BASIC, every time a line executes, the interpreter searches for the variable by name. From the beginning. This was very slow. A compiler is different: it resolves names to locations once during compilation and bakes the result into the machine code. The compiled program never sees the symbol table. That's the C way. Compiled, not interpreted. The idea comes directly from the javascript assembler. It inserts dummy values for pointers until the labels can be resolved. AKA back-patching.

Symbol table record structure

The symbol table is a flat array of fixed-size records. I was worried about doing flat sized records because it can waste a lot of space but as it turns out we can do 100 or so in just 2k and that is enough for bootstrapping. Each record occupies 20 bytes with the following layout:

Offset	Size	Field	Description
0–15	16	Name	Null-terminated, padded to 16 bytes (max 15 chars)
16	1	Type	0=int, 1=int*, 2=char, 3=char*
17	1	Kind	0=local, 1=param, 2=function, 3=global
18–19	2	Value	16-bit: stack offset (locals/params), code address (functions), or memory address (globals)

Note: name size was increased to 15 from 7 because some function names like get_thing() were larger than 7 characters. If we run out of space change this first.

How it works

The big reveal is that it doesn't store variables in the table like NVAR does, it stores handles (pointers) to the variables, and the variables are placed onto the stack. That is so we can do scoping (see below).

Three operations manage the table, like NVAR_SET, NVAR_GET and NVAR_CLEAR, here we use SVAR_SET (sym_add), SVAR_GET (sym_find) and SVAR_CLEAR (sym_clear).

- SVAR_SET/sym_add()
  - Appends a new entry. Needs name, and value like NVAR but additionally needs type (ex int, char) and kind (ex function, global, param).
- SVAR_GET/sym_find()
  - Performs a linear scan from the first entry, comparing each stored name against the search string using `strcmp`. Faster than NVAR because it's fixed length. Second, it returns the *address* of that entry, not the value. This is so we can do pointers.
- SVAR_CLEAR/sym_clear()
  - resets the count to zero. Just write a zero at the start to show no variables.

Scoping using the stack

The compiler implements function-level scoping by saving the values on the stack and only keeping a handle to them in the symbol table. When parsing a function definition, the current symbol count is saved into `CC_SYM_SCOPE`. Parameters and local variables are added during parsing. When the function's closing brace is reached, the count is restored to the saved value, effectively discarding all locals and parameters while preserving function-level entries (other function names, globals) that were added before. Kind of like “FORGET” in Forth.

Type vs Kind

- Kind 0 = Local Variable
  - A positive integer representing the distance below the frame pointer (GLZ).
- Kind 1 = Params
  - A positive integer representing the distance above the frame pointer. Accessed as `[GLZ + offset]`.
- Kind 2 = Functions
  - The 16-bit code address where the function's compiled body begins. Used by `emit_call` to generate `CALL addr` instructions.
- Kind 3 = Globals
  - A fixed memory address in bank 0, allocated from a bump pointer starting at `$D000`. This can't go on the stack for some reason I forgot. (CHECKME)

Compiling ''int a = 5;''

1. CC_FRAME_OFFSET goes from 0 to 2 (each int is 2 bytes)
2. cc_sym_add is called with name="a", type=0, kind=0, value=2
3. It computes the slot address: CC_SYMTAB + (count * 12) = $00F040
4. Writes "a\0\0\0\0\0\0" (name padded to 8), then 0x00 (type), 0x00 (kind), 0x02 0x00 (value = 2, little-endian)
5. Increments CC_SYM_COUNT to 1

Next for `int b = 3;`

1. CC_FRAME_OFFSET goes from 2 to 4
2. Slot address: $00F040 + (1 * 12) = $00F04C
3. Writes "b\0..." with value=4

When `return a;` needs the variable:

1. . cc_sym_find does a linear scan from $00F040
2. . Compares each record's name against CC_TOK_BUF using strcmp
3. . Finds "a" at record 0, returns a pointer (usu. BLY or LLZ) pointing to that record
4. . The caller reads the value field at BLY+10, gets 2
5. . Emits LDA [GLZ-2] (load from frame pointer minus 2)

When a new `.compile` starts:

cc_sym_clear sets CC_SYM_COUNT to 0. Like calling NVAR_CLEAR on RUN in BASIC. The old data is still in memory, but invisible since nothing scans past count.

The table is purely compile-time. The compiled program has no idea it exists. Variable names are resolved to numeric stack offsets during compilation and baked into the emitted instructions.

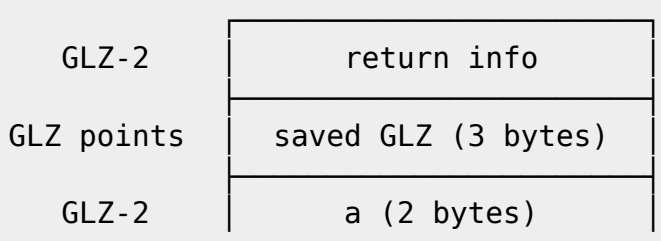
But why stack frames?

The stack is used because local variables need to appear when a function is entered and disappear when it returns, and the stack naturally provides that lifetime.

When `int main(void) { int a = 5; int b = 3; return a + b; }` is compiled, the emitted code does this:

```
PUSH GLZ          ; save old frame pointer (3 bytes on stack)
MOV GLZ, SP       ; GLZ = current stack top. This is our reference point
SUB SP, #2        ; reserve 2 bytes for 'a'. SP moves down
SUB SP, #2        ; reserve 2 bytes for 'b'. SP moves down again
```

The stack now looks like this (growing downward):



```
                ┌──────────────────────────┐
                │                          │
    GLZ-4       │      b (2 bytes)         │
                │                          │
  SP points     │      (free space)        │
                └──────────────────────────┘
```

a is always at GLZ-2. b is always at GLZ-4. These offsets are fixed at compile time and baked into the instructions. STA [GLZ-2] stores into a. LDA [GLZ-4] reads from b.

But why male models?

(But why on the stack?) It boils down to a memory management issue. You could assign a to address $1000 and b to $1002. For a single function that's called once, it would work. The problem is if a second function uses the name of a local variable in a function it calls. Recursion is another example of this:

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

If n lives at a fixed address, the second call to `factorial` (the second use of n) overwrites the first n. Then when the inner call returns and the outer call tries to read it's own n, the value is gone.

The solution is, every call needs its own private copy of n. If we place variables on the stack when we call a function, then we can just pop them off when we leave. Boom, instant memory management with scoping on top.

Working through if-else

After the symbol table, everything else started to fall into place quickly; if was next, and I already had the concept from the expression parser in BASIC.

By this time the language structure was starting to become established so I will write down how I added if as a guide for other keywords.

To get a new language feature like `if-else`, we have to follow the same pattern as in BASIC. This means:

1. Add it to the lexer (match keyword).
2. Parse the statement (done in the executor in BASIC, but a separate step here).
3. Emitter – This is the big diff, since we don't do it 'now' but we have to do it 'later' (at runtime).

The plan is the exact same as assembly (labels); back-patching, technically a two-pass, but done in the moment so we say it's one pass.

1. Emit the condition (cc_parse_expr, result in A)
2. Emit CMP A, #0
3. Emit JZ with a placeholder address. Save CC_HERE before writing it
4. Parse the then-block (emits its code)

5. Go back and patch the JZ address to point here (past the block)

for if-else:

1. Emit condition (same)
2. Emit CMP A, #0 (same)
3. JZ placeholder (same)
4. Parse then-block (same)
5. Emit JMP placeholder (skip over else)
6. Patch placeholder to here
7. Parse else-block
8. Patch placeholder to here

Documenting the process

Just like adding a keyword in BASIC.

1. Add the tokens.

```
    .equ TOK_IF             17
    .equ TOK_ELSE           18
```

2. Add the strings.

```
    cc_kw_if:
        .bytes "if", 0
    cc_kw_else:
        .bytes "else", 0
```

3. Add the keyword checks in the keyword checks section.

```
    ; Check "if"
    LDELM CC_TOK_BUF
    LDFLD @cc_kw_if
    LDAH $02
    INT $12
    JZ @cc_nt_kw_if
    ; Check "else"
    LDELM CC_TOK_BUF
    LDFLD @cc_kw_else
    LDAH $02
    INT $12
    JZ @cc_nt_kw_else
```

And the handlers,

```
    cc_nt_kw_if:
        LDAL TOK_IF
        STAL [@CC_TOKEN_TYPE]
        POP I
        RET
    cc_nt_kw_else:
        LDAL TOK_ELSE
        STAL [@CC_TOKEN_TYPE]
        POP I
        RET
```

We have to add it to the parser:

```
    cc_parse_stmt:
        LDAL [@CC_TOKEN_TYPE]
        CMP AL, TOK_RETURN
        JZ @cc_parse_return
        CMP AL, TOK_INT
        JZ @cc_parse_vardecl
        CMP AL, TOK_IDENT
        JZ @cc_parse_assign
        CMP AL, TOK_IF  ; Added here at the end
        JZ @cc_parse_if
        ; Unknown statement
        LDAL #1
        STAL [@CC_ERROR]
        LDELM @cc_str_expect_stmt
        LDAH $18
        INT $10
        RET
```

Finally here is the if parser. Note three important changes from BASIC.

1. Instead of manually checking for things like) the system uses a TOKEN table (TOK_LPAREN, etc). This generalized approach, I think, adds to thinkability and expandability.
2. You don't see int05_skip_space everywhere. Instead we have a cc_skip_ws which is included in cc_next_token. This isn't as 'inlined' as basic, but it doesn't need to be since it's a compiler. Keeping things like skip_space out of the way aids code understandability.
3. It's an emitter not an interpreter. Instead of doing it “now” we need to have a plan to do it 'later'.

```
; cc_parse_if
; 'if' '(' expr ')' block ('else' block)?
;
; Emitted code shape (if only):
;   <expr>            ; result in A
;   CMP_IMM A, #0     ; test condition
;   JZ patch1         ; skip then-block if false
```

```
;   <then-block>
;   patch1:           ; JZ lands here
;
; Emitted code shape (if/else):
;   <expr>
;   CMP_IMM A, #0
;   JZ patch1         ; skip to else if false
;   <then-block>
;   JMP patch2        ; skip over else
;   patch1:           ; else starts here
;   <else-block>
;   patch2:           ; both paths rejoin here
;
cc_parse_if:
    ; Consume 'if'
    CALL @cc_next_token

    ; Expect '('
    LDAL TOK_LPAREN
    CALL @cc_expect
    LDAL [@CC_ERROR]
    CMP AL, #0
    JNZ @cc_pi_done

    ; Parse condition expression (result in A)
    CALL @cc_parse_expr
    LDAL [@CC_ERROR]
    CMP AL, #0
    JNZ @cc_pi_done

    ; Expect ')'
    LDAL TOK_RPAREN
    CALL @cc_expect
    LDAL [@CC_ERROR]
    CMP AL, #0
    JNZ @cc_pi_done

    ; Emit: CMP_IMM A, #0 (opcode=91, reg=0, imm16=0,0)
    LDAL #91
    CALL @cc_emit_byte
    LDAL #0                         ; REG_A
    CALL @cc_emit_byte
    LDAL #0                         ; imm16 lo = 0
    CALL @cc_emit_byte
    LDAL #0                         ; imm16 hi = 0
    CALL @cc_emit_byte

    ; Emit: JZ <placeholder> — save address of the operand for patching
    LDAL #101                       ; JZ opcode
    CALL @cc_emit_byte
```

```
    ; Save CC_HERE — this is where the 3-byte address will go
    LDBLX [@CC_HERE]
    PUSH BLX                        ; patch1 address saved on stack
    ; Emit 3 placeholder bytes
    LDAL #0
    CALL @cc_emit_byte
    CALL @cc_emit_byte
    CALL @cc_emit_byte

    ; Parse then-block
    CALL @cc_parse_block
    LDAL [@CC_ERROR]
    CMP AL, #0
    JNZ @cc_pi_bail1

    ; Check for 'else'
    LDAL [@CC_TOKEN_TYPE]
    CMP AL, TOK_ELSE
    JZ @cc_pi_else

    ; No else = patch1 = CC_HERE (jump past then-block)
    POP BLX                         ; BLX = patch1 location
    CALL @cc_backpatch
    JMP @cc_pi_done

cc_pi_else:
    ; Consume 'else'
    CALL @cc_next_token

    ; Emit: JMP <placeholder> at end of then-block
    LDAL #100                       ; JMP opcode
    CALL @cc_emit_byte
    LDBLY [@CC_HERE]
    PUSH BLY                        ; patch2 address saved on stack
    LDAL #0
    CALL @cc_emit_byte
    CALL @cc_emit_byte
    CALL @cc_emit_byte

    ; We need patch1 now. Swap:
    POP BLY                         ; BLY = patch2
    POP BLX                         ; BLX = patch1
    PUSH BLY                        ; re-push patch2

    CALL @cc_backpatch              ; patch1 = current CC_HERE

    ; Parse else-block
    CALL @cc_parse_block
    LDAL [@CC_ERROR]
    CMP AL, #0
    JNZ @cc_pi_bail1
```

```
    ; Patch2 = CC_HERE (JMP lands past else-block)
    POP BLX                         ; BLX = patch2 location
    CALL @cc_backpatch

cc_pi_done:
    RET

cc_pi_bail1:
    POP BLX                         ; discard patch address
    RET
```

For a backpatch helper (in case we need this elsewhere) we just copy HERE:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; backpatch helper
cc_backpatch:
    PUSH ELM
    LDELM [@CC_HERE]
    STELM [FLD]
    POP ELM
    RET
```

while is easy

Once we have backpatching and the expression machinery, every new control flow construct is just a new arrangement of jumps. `for` would be the same. The mantra is “All loops are possible with goto” (See: |more loops from the SD-8516 User's Guide).

Functions and Pointers and Etc

To make a long story short I started to feel the pain of a 16 bit symbol table as I had already done a significant amount of work and I was in no mood to rewrite all the code to make it 24 or 32 bits. I was having significant issues because the word size is 16 bits but pointers are 24 bits.

So I came up with the brilliant idea of just making everything in bank 0.

Final Boss: Strings

Strings were the final boss. I am still working on them, kind of. Just fixed some errors this morning.

Adding strings was easy after char *. Same process: add TOK_CHAR. cc_kw_char. etc. It was the usual suspects for bugs again too. The usual pointer clobber bugs.

Jumping past strings

The key is to jump past the string. The data lives inside the code.

```
  char *s = "Hello";
```

When we emit the code here we will add a jmp, insert the string, then set s to the address of the string. This will allow us to reference the string like an array (s[6]) and so forth. Buffer overflows! Ahh so that's where this stuff comes from.

```
  s[6] = 133; call opcode; now we're in business!
```

Moving the Stack

I eventually had to move the stack because the stack is in bank 1 and this means `&local_var` is broken. We can not get the address of a local variable since it is technically stored in bank 1. This is a problem for strings too. This, for example, returns 40 (not 101):

```
  int main(void) {
      char *msg = "Hello";
      return msg[1];
  }
```

The solution is to move the SP inside the C REPL only. While we are inside we will do something like:

```
  MOV GLZ, SP
  MOV SP, $FFFF
  PUSH GLZ
  <program code here>
  POP SP
  RET
```

Built-in functions

Once strings worked I added putchar() as a built-in function. All I did was add the emitter to cc_do_compile. Unfortunately I added it before the symbol table was cleared so it didn't work. The I moved it. Here's the emitter:

```
; Built-in function emitter.
; Emits machine code for built-in functions and registers them in the symbol
table.
cc_emit_builtins:
    CALL @cc_builtin_putchar
    RET
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; putchar(c)
; Prints character c to terminal via INT $10 AH=$17
cc_builtin_putchar:
    ; Register in symbol table
    LDA [@CC_HERE]
    MOV B, A
    LDELM @cc_bi_putchar

    LDAL #0    ; type = 0 (int)
    LDAH #2    ; kind = 2 (function)
    CALL @cc_sym_add

    ; Emit prologue
    CALL @cc_emit_push_glz
    CALL @cc_emit_mov_glz_sp

    ; Emit: LDA [GLZ+6] (first arg)
    LDA #210
    CALL @cc_emit_byte
    LDA #0
    CALL @cc_emit_byte
    LDA #94
    CALL @cc_emit_byte
    LDA #6
    CALL @cc_emit_byte

    ; Emit: LDAH $17 (teletype putchar)
    LDA #0
    CALL @cc_emit_byte
    LDA #40
    CALL @cc_emit_byte
    LDA $17
    CALL @cc_emit_byte

    ; Emit: INT $10
    LDA #134
    CALL @cc_emit_byte
    LDA $10
    CALL @cc_emit_byte

    ; Emit epilogue
    CALL @cc_emit_mov_sp_glz
    CALL @cc_emit_pop_glz
    CALL @cc_emit_ret
    RET


; --- Name strings ---
cc_bi_putchar:
```

```
    .bytes "putchar", 0
```

After this, I added getchar(), halt() and yield(), and then things started getting really easy, for example print() is just:

```
int print(char *s) {
    int i = 0;
    while (s[i] != 0) {
        putchar(s[i]);
        i = i + 1;
    }
    return 0;
}
```

Wow, so we're writing our library in C now. Things are starting to go well from here!

- List of TinyC built-in functions

Evil Stack Bugs Strike Again

While not technically a bug, there's a 'feature' of this TinyC that lies in wait for unsuspecting victims...

```
  while (condition) {
      ....
      int c = msg[i];
      ....
  }
```

This emits SUB SP, #2 every iteration. On a bigger loop it will destroy the stack. The solution is found in K&R C. You must declare your variables at the start of the function. I mean, I always believed that was good programming style, but it's interesting that there are historical and technical reasons for it too. I never knew.

break/continue notes

1. PUSH outer break count (2 bytes)
2. PUSH outer loop top (3 bytes)
3. PUSH loop_top for JMP back (3 bytes)
4. PUSH patch_exit (3 bytes)
5. POP patch_exit for JMP emit, re-push
6. POP loop_top into ILY
7. POP patch_exit for backpatch
8. POP outer loop top, restore
9. POP outer break count, restore

1)

3 * 4) / 5