Part 59: Why Doesn't It Work, part 1

We've reached the **WDIW** stage: why doesn't it work? In this first part of this stage, I find a few easy to find bugs and fix them. That means there are some more subtle bugs yet to be uncovered.

Bad Code Generation for `*argv[i]`

I'm using `cwj` (the Gnu C compiled version) to build `cwj0`. The assembly code in `cwj0` is *our* assembly code, not the assembly code generated by Gnu C. So, when we run `cwj0`, any errors are because our assembly code isn't correct.

The first bug I noticed was that `*argv[i]` seemed to be generating code as if it was `(*argv)[i]`, i.e. always the *i'th* character of `*argv`, not the first character at `argv[i]`.

I first thought this was a parsing error, but no. It turned out that we were not setting `argv[i]` as an rvalue before we dereferenced it. I worked this out by dumping the AST trees with both `cwj` and `cwj0` and observing the differences between them. What we need to do is mark the expression *after* the `*` token as an rvalue. This is now done in `prefix()` in `expr.c`:

```
static struct ASTnode *prefix(int ptp) {
  struct ASTnode *tree;
  switch (Token.token) {
  ...
  case T_STAR:
```

```
    // Get the next token and parse it
    // recursively as a prefix expression.
    // Make it an rvalue
    scan(&Token);
    tree = prefix(ptp);
    tree->rvalue= 1;
```

Externs are Also Globals

This one is going to keep biting me, I'm sure. I found another place where I wasn't treating an extern symbol as a global symbol. This was in `genAST()` in `gen.c` where we generate assignment assembly code. The fix is:

```
    // Now into the assignment code
    // Are we assigning to an identifier or through a pointer?
    switch (n->right->op) {
      case A_IDENT:
        if (n->right->sym->class == C_GLOBAL ||
            n->right->sym->class == C_EXTERN ||
            n->right->sym->class == C_STATIC)
          return (cgstorglob(leftreg, n->right->sym));
        else
          return (cgstorlocal(leftreg, n->right->sym));
```

Scanning is Working

At this point, the `cwj0` compiler is reading source code input but not generating any output. Here are the new `Makefile` rules to do this:

```
# Try to do the triple test
triple: cwj1

cwj1: cwj0 $(SRCS) $(HSRCS)
        ./cwj0 -o cwj1 $(SRCS)

cwj0: install $(SRCS) $(HSRCS)
        ./cwj -o cwj0 $(SRCS)
```

So, a `$ make triple` will build `cwj` with Gnu C, then build `cwj0` with `cwj`, and finally build `cwj1` with `cwj0`. I'll talk about this below.

Right now, `cwj1` can't be created as there is no assembly output! The question is, where is the compiler getting to? To find out, I added a `printf()` to the bottom of `scan()`:

```
  // We found a token
  t->tokstr = Tstring[t->token];
  printf("Scanned %d\n", t->token);
  return (1);
```

With this added, I saw that both `cwj` and `cwj0` scan 50,404 tokens and the resulting token streams are identical. Thus, we can conclude that, up to `scan()`, things are working OK.

However, the output of `./cwj0 -S -T cg.c` show no AST trees. If I run `gdb cwj0`, set a breakpoint in `dumpAST()` and run it with the `-S -T cg.c` arguments, then we exit before we break at `dumpAST()`. We also don't get to `function_declaration()`. So, why doesn't it work?

Ah, I spotted a memory access to `0(%rbp)`. This should never happen as all locals are at negative locations relative to the frame pointer. In `cgaddress()` in `cg.c`, we have another missed external test. We now have:

```
int cgaddress(struct symtable *sym) {
  int r = alloc_register();

  if (sym->class == C_GLOBAL ||
      sym->class == C_EXTERN ||
      sym->class == C_STATIC)
    fprintf(Outfile, "\tleaq\t%s(%%rip), %s\n", sym->name, reglist[r]);
  else
    fprintf(Outfile, "\tleaq\t%d(%%rbp), %s\n", sym->st_posn, reglist[r]);
  return (r);
}
```

Damn these extern problems! Well, it's all my fault, so I need to take the blame here.

Bad Comparisons

With the above change added, we are now failing with:

```
$ ./cwj0 -S tests/input001.c
invalid digit in integer literal:e on line 1 of tests/input001.c
```

This turned out to be caused by this loop in `scanint()` in `scan.c()`:

```
static int scanint(int c) {
  int k;
  ...
  // Convert each character into an int value
  while ((k = chrpos("0123456789abcdef", tolower(c))) >= 0) {
```

What is happening is that the `k=` assignment is not only storing a result in memory, but it is being used as an expression. In this case the `k` result is being compared, i.e. `k >= 0`. Now, `k` is of type `int`, and we are performing this store to memory for its assignment:

```
    movl    %r10d, -8(%rbp)
```

When `chrpos()` returns `-1`, this gets truncated down to 32 bits (`0xffffffff`) and stored in `-8(%rbp)`, i.e. in `k`. But in the following comparison:

```
    movslq  -8(%rbp), %r10    # Load value back from k
    movq    $0, %r11          # Load zero
    cmpq    %r11, %r10        # Compare k's value against zero
```

we load the *32-bit* value of `k` into `%r10`, and now do a *64-bit* comparison. Well, as a 64-bit value, `0xffffffff` is a positive number, the loop comparison remains true and we don't leave the loop when we should.

What we should do is use a different `cmp` instruction based on the size of the operands in the comparison. I've made this change to `cgcompare_and_set()` in `cg.c`:

```
int cgcompare_and_set(int ASTop, int r1, int r2, int type) {
  int size = cgprimsize(type);
  ...
  switch (size) {
  case 1:
    fprintf(Outfile, "\tcmpb\t%s, %s\n", breglist[r2], breglist[r1]);
    break;
  case 4:
    fprintf(Outfile, "\tcmpl\t%s, %s\n", dreglist[r2], dreglist[r1]);
    break;
  default:
    fprintf(Outfile, "\tcmpq\t%s, %s\n", reglist[r2], reglist[r1]);
  }
```

```
    ...
}
```

Now the correct comparison instruction is being used. There is a similar function, `cgcompare_and_jump()`, and at some stage I should refactor and merge the two functions.

Now, So Close!

We are so close to passing what is informally known as the **triple test**. In the triple test, we use an existing compiler to build our compiler from source code (stage 1). Then we use this compiler to build itself (stage 2). Now, to prove that the compiler is self-compiling, we use the stage 2 compiler to build itself, resulting in the stage 3 compiler.

We can now:

- build `cwj` using the Gnu C compiler (stage 1)
- build `cwj0` using the `cwj` compiler (stage 2)
- build `cwj1` using the `cwj0` compiler (stage 3)

However, the binary sizes for `cwj0` and `cwj1` don't match:

```
$ size cwj[01]
   text    data     bss     dec     hex filename
 109636    3028      48  112712   1b848 cwj0
 109476    3028      48  112552   1b7a8 cwj1
```

and they should match *exactly*. Only when the compiler can compile itself multiple times in a row and produce the same result do we know that it is self-compiling properly.

Until the results match exactly, there is some subtle behaviour difference between stages 2 and 3, and so the compiler isn't compiling itself consistently.

Conclusion and What's Next

I didn't think I'd get to the point where I can build `cwj`, `cwj0` and `cwj1` in a single step of this journey. I expected we would have a whole pile of bugs to fix before we got to this point.

The next problem is to work out why the stage 2 and stage 3 compilers are different sizes. Looking at the `size` output, the data and bss sections are the same, but the amount of assembly code is different between the two compilers.

In the next part of our compiler writing journey, we will try to do a side-by-side comparison of the assembly output between the different stages, and try to work out what is causing the difference.

> P.S. In this part of the journey, I also started to add some assembly output which would allow `gdb` to see the source line number that we are stopped on. It isn't working yet, but in case you look, you will see a new function `cglinenum()` in `cg.c`. When I get it working, I'll write up some commentary on it. Next step