Part 27: Regression Testing and a Nice Surprise

We've had a few large-ish steps recently in our compiler writing journey, so I thought we should have a bit of a breather in this step. We can slow down a bit and review our progress so far.

In the last step I noticed that we didn't have a way to confirm that our syntax and semantic error checking was working correctly. So I've just rewritten the scripts in the `tests/` folder to do this.

I've been using Unix since the late 1980s, so my go-to automation tools are shell scripts and Makefiles or, if I need more complex tools, scripts written in Python or Perl (yes, I'm that old).

So let's quickly look at the `runtest` script in the `tests/` directory. Even though I said I'd been using Unix scripts forever, I'm definitely not an uber script writer.

The `runtest` Script

The job of this script is to take a set of input programs, get our compiler to compile them, run the executable and compare its output against known-good output. If they match, the test is a success. If not, it's a failure.

I've just extended it so that, if there is an "error" file associated with an input, we run our compiler and capture its error output. If this error output matches the expected error output, the test is a success as the compiler correctly detected the bad input.

So let's look at the sections of the `runtest` script in stages.

```
# Build our compiler if needed
if [ ! -f ../comp1 ]
then (cd ..; make)
fi
```

I'm using the '(...)' syntax here to create a *sub-shell*. This can change its working directory without affecting the original shell, so we can move up a directory and rebuild our compiler.

```
# Try to use each input source file
for i in input*
# We can't do anything if there's no file to test against
do if [ ! -f "out.$i" -a ! -f "err.$i" ]
   then echo "Can't run test on $i, no output file!"
```

The '[' thing is actually the external Unix tool, *test(1)*. Oh, if you've never seen this syntax before, *test(1)* means the manual page for *test* is in Section One of the man pages, and you can do:

```
$ man 1 test
```

to read the manual for *test* in Section One of the man pages. The `/usr/bin/[` executable is usually linked to `/usr/bin/test`, so that when you use '[' in a shell script, it's the same as running the *test* command.

We can read the line `[ ! -f "out.$i" -a ! -f "err.$i" ]` as saying: test if there is no file "out.$i" and no file "err.$i". If both don't exist, we can give the error message.

```
   # Output file: compile the source, run it and
   # capture the output, and compare it against
   # the known-good output
   else if [ -f "out.$i" ]
        then
          # Print the test name, compile it
          # with our compiler
          echo -n $i
          ../comp1 $i

          # Assemble the output, run it
          # and get the output in trial.$i
          cc -o out out.s ../lib/printint.c
          ./out > trial.$i

          # Compare this agains the correct output
          cmp -s "out.$i" "trial.$i"

          # If different, announce failure
          # and print out the difference
          if [ "$?" -eq "1" ]
          then echo ": failed"
            diff -c "out.$i" "trial.$i"
            echo

          # No failure, so announce success
          else echo ": OK"
          fi
```

This is the bulk of the script. I think the comments explain what is going on, but perhaps there are some subtleties to flesh out. `cmp -s` compares two text files; the `-s` flag means produce no output but set the exit value that `cmp` gives when it exits to:

> 0 if inputs are the same, 1 if different, 2 if trouble. (from the man page)

The line `if [ "$?" -eq "1" ]` says: if the exit value of the last command is equal to the number 1. So, if the compiler's output is different to the known-good output, we announce this and use the `diff` tool to show the differences between the two files.

```
   # Error file: compile the source and
   # capture the error messages. Compare
   # against the known-bad output. Same
   # mechanism as before
   else if [ -f "err.$i" ]
        then
          echo -n $i
          ../comp1 $i 2> "trial.$i"
          cmp -s "err.$i" "trial.$i"
          ...
```

This section gets executed when there is an error document, "err.$i". This time, we use the shell syntax `2>` to capture our compiler's standard error output to the file "trial.$i" and compare that against the correct error output. The logic after this is the same as before.

What We Are Doing: Regression Testing

I haven't talked much before about testing, but now's the time. I've taught software development in the past so it would be remiss of me not to cover testing at some point.

What we are doing here is regression testing. Wikipedia gives this definition:

> Regression testing is the action of re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change.

As our compiler is changing at each step, we have to ensure that each new change doesn't break the functionality (and the error checking) of the previous steps. So each time I introduce a change, I add one or more tests to a) prove that it works and b) re-run this test on future changes. As long as all the tests pass, I'm sure that the new code hasn't broken the old code.

Functional Tests

The `runtests` script looks for files with the `out` prefix to do the functional testing. Right now, we have:

```
tests/out.input01.c  tests/out.input12.c   tests/out.input22.c
tests/out.input02.c  tests/out.input13.c   tests/out.input23.c
tests/out.input03.c  tests/out.input14.c   tests/out.input24.c
tests/out.input04.c  tests/out.input15.c   tests/out.input25.c
tests/out.input05.c  tests/out.input16.c   tests/out.input26.c
tests/out.input06.c  tests/out.input17.c   tests/out.input27.c
tests/out.input07.c  tests/out.input18a.c  tests/out.input28.c
tests/out.input08.c  tests/out.input18.c   tests/out.input29.c
tests/out.input09.c  tests/out.input19.c   tests/out.input30.c
tests/out.input10.c  tests/out.input20.c   tests/out.input53.c
tests/out.input11.c  tests/out.input21.c   tests/out.input54.c
```

That's 33 separate tests of the compiler's functionality. Right now, I know for a fact that our compiler is a bit fragile. None of these tests really stress the compiler in any way: they are simple tests of a few lines each. Later on, we will start to add some nasty stress tests to help strengthen the compiler and make it more resilient.

Non-Functional Tests

The `runtests` script looks for files with the `err` prefix to do the functional testing. Right now, we have:

```
tests/err.input31.c  tests/err.input39.c  tests/err.input47.c
tests/err.input32.c  tests/err.input40.c  tests/err.input48.c
tests/err.input33.c  tests/err.input41.c  tests/err.input49.c
tests/err.input34.c  tests/err.input42.c  tests/err.input50.c
tests/err.input35.c  tests/err.input43.c  tests/err.input51.c
tests/err.input36.c  tests/err.input44.c  tests/err.input52.c
tests/err.input37.c  tests/err.input45.c
tests/err.input38.c  tests/err.input46.c
```

I created these 22 tests of the compiler's error checking in this step of our journey by looking for `fatal()` calls in the compiler. For each one, I've tried to write a small input file which would trigger it. Have a read of the matching source files and see if you can work out what syntax or semantic error each one triggers.

Other Forms of Testing

This isn't a course on software development methodologies, so I won't give too much more coverage on testing. But I'll give you links to a few more thing that I would highly recommend that you look at:

- Unit testing
- Test-driven development
- Continuous integration
- Version control

I haven't done any unit testing with our compiler. The main reason here is that the code is very fluid in terms of the APIs for the functions. I'm not using a traditional waterfall model of development, so I'd be spending too much time rewriting my unit tests to match the latest APIs of all the functions. So, in some sense I am living dangerously here: there will be a number of latent bugs in the code which we haven't detected yet.

However, there are guaranteed to be *many* more bugs where the compiler looks like it accepts the C language, but of course this isn't true. The compiler is failing the principle of least astonishment. We will need to spend some time adding in functionality that a "normal" C programmer expects to see.

And a Nice Surprise

Finally, we have a nice functional surprise with the compiler as it stands. A while back, I purposefully left out the code to test that the number and type of arguments to a function call matches the function's prototype (in `expr.c`):

```
    // XXX Check type of each argument against the function's prototype
```

I left this out as I didn't want to add too much new code in one of our steps.

Now that we have prototypes, I've wanted to finally add support for `printf()` so that we can ditch our homegrown `printint()` and `printchar()` functions. But we can't do this just yet, because `printf()` is a variadic function: it can accept a variable number of parameters. And, right now, our compiler only allows a function declaration with a fixed number of parameters.

However (and this is the nice surprise), because we don't check the number of arguments in a function call, we can pass *any* number of arguments to `printf()` as long as we have given it an existing prototype. So, at present, this code (`tests/input53.c`) works:

```
int printf(char *fmt);

int main()
{
  printf("Hello world, %d\n", 23);
  return(0);
}
```

And that's a nice thing!

There is a gotcha. With the given `printf()` prototype, the cleanup code in `cgcall()` won't adjust the stack pointer when the function returns, as there are less than six parameters in the prototype. But we could call `printf()` with ten arguments: we'd push four of them on the stack, but `cgcall()` wouldn't clean up these four arguments when `printf()` returns.

Conclusion and What's Next

There is no new compiler code in this step, but we are now testing the error checking capability of the compiler, and we now have 54 regression tests to help ensure we don't break the compiler when we add new functionality. And, fortuitously, we can now use `printf()` as well as the other external fixed parameter count functions.

In the next part of our compiler writing journey, I think I'll try to:

- add support for an external pre-processor
- allow the compiler to compile multiple files named on the command line

- add the `-o`, `-c` and `-S` flags to the compiler to make it feel more like a "normal" C compiler Next step