



CHALMERS

Compiler construction

Lecture 2: Software Engineering for Compilers

Thomas Sewell

Spring 2020

Chalmers University of Technology — Gothenburg University

Good software engineering for this course and “real life”

- Structuring the project
- Is the compiler correct?
- Compiler bootstrapping
- Writing Makefiles
- Managing state in compilers
- Using the assignment testsuite

Structuring the project

Passes

- Lexer
- Parser
- Type checker
- Return checking¹
- Code generator

Structuring passes

- In functional languages, a pass correspond to a function
- In OO style, a pass corresponds to a visitor method
 - OO languages can also use functional style!

¹Can be done as a separate pass or as part of the type checker

- Generate a lexer and parser using BNFC, you will have to change the BNFC file for JAVALETTE that we provide for you
 - <https://bnfc.digitalgrammars.com/>
- Write typechecker
- Write code generator (parts B & C)
- Write a `main` function which connects the above pieces together, and executes LLVM (parts B & C)

- It is highly recommended that you use version control software; using version control software is an essential practice when developing code
- For example: git, darcs, subversion, mercurial, ...
- However, do not put your code in a public repository, where others can see your code
- Use educational account for GitHub or BitBucket
- Alternative: use a Dropbox folder as a git remote (create a bare repo)

Is the compiler correct?

Bugs

When something goes wrong with our program, we go to great lengths to find bugs in our own code.

- It's extremely disappointing if the problem is with the compiler.
- Programmers trust the compiler to generate correct code
- The most important task of the compiler is to generate correct code

Options

- Proving the correctness of a compiler (too complicated?)
- Testing

Testing compilers

- Most compilers use unit testing
- They have a big collection of example programs which are used for testing the compiler
- For each program the expected output is stored in the test suite
- Whenever a new bug is found, a new example program is added to the test suite; this is known as regression testing
 - (not the same as statistical regression testing)

Instead of hand-writing all tests, we can use random values.

Challenge: we don't know the correct output.

A/B testing: Compare similar things

- Test program A against known good program B
- e.g. during a major upgrade

Fuzz testing

- Simple random testing with nonsense “fuzz”
- Check that the program succeeds
- e.g. for a compiler, check the compiler terminates
 - Reporting an error is OK
 - Crashing, looping or raising an exception is not OK

Property-based testing

- Specify (semi-formal) properties that software should have
- Generate random inputs and check the properties

Example

```
propReverse :: [Int] -> [Int] -> Bool
propReverse xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs
```

```
Prelude Test.QuickCheck> quickCheck propReverse
+++ OK, passed 100 tests.
```

- Writing good random generators for a source language is very difficult
- Different parts of the compiler might need different generators
 - The parser needs random strings, but they need to be skewed towards syntactically correct programs in order to be useful
 - The type checker needs a generator which can generate type correct programs (with high probability)
- How do we evaluate an output program? Run it? What if it takes a very long time?
- Using random testing for compilers is difficult and a lot of work
 - This is an active research area

Remember to test your compiler!

- Use the provided test suite!
- Write your own tests!

Hmm ...

If systematic testing of compilers is so difficult, why not look at the other option:

Proving the correctness of compilers!

There will be a lecture on this topic later in the course.

Can't wait? Talk to us:

Compiler verification is our research topic, in particular for non-pure functional programming languages.

Check out: <https://cakeml.org>

MSc thesis topic?

Compiler Bootstrapping

Some people say:

A programming language isn't real until it has a self-hosting compiler.

Why a self-hosting compiler?

If you've designed an awesome programming language, surely you want to program in it.

In particular, you would want to write the compiler in this language.

If we want to write a compiler for the language X in the language X, how does the first compiler get written?

Solutions

- Write an interpreter for language X in language Y
- Write another compiler for language X in language Y
- Write the compiler in a subset of X which is possible to compile with an existing compiler
- Hand-compile the first compiler

Building on a new architecture

How to first build a compiler on a new hardware architecture?

Solution: cross-compilation

- The compiler can emit code for architecture A while running on architecture B.

Tricky build processes

GCC, for example, builds and then rebuilds itself

- Necessary because many optional components are written using GCC-specific C extensions.

Writing Makefiles

The build automation tool `make` is handy for compiling large projects. It keeps track of which files need to be recompiled.

A Makefile consists of rules which specify:

- Which target file will be generated
- How these files are generated

General structure of rules

```
target : dependencies ...  
    shell commands specifying how to generate target
```

Concrete example

```
compiler : parser.o typechecker.o  
    gcc -o compiler parser.o typechecker.o  
parser.o : parser.c  
    gcc -c parser.c -o parser.o
```

Pattern rules

- Build rules tend to be repetitive
 - e.g. each source file builds an object file
- Then pattern rules come in handy

```
%.o : %.c  
    gcc -c $< -o $@
```

Warning

- The space before the shell commands needs to be a tab stop!
- If you just use spaces then the commands will not execute

Invoking `make`

- `make` with no arguments will make the first target in the Makefile
- `make X` will try to build `X` and all of its dependencies as needed

Using `PHONY` rules

- Sometimes it is convenient to have targets which do not produce files
- A common example is `clean` which removes all generated files
- These targets should be declared as `PHONY`

```
.PHONY clean
```

```
clean:
```

```
rm -f *.o
```

- There are many more features to `make`, but these basics will get you fairly far
- `make` (and the shell) have been with us a long time. They have many flaws, but it is (still) good to know how to use them well

Project

- You are expected to use `make` in the project
- In the project you automatically get a Makefile from the BNFC tool
- Don't forget to `make clean` before packaging your solution for submission
- It can be very convenient to have a target which automatically makes a package for submission

Managing state in the compiler

- When writing the type checker and code generator, the compiler needs to store symbol tables with information about e.g. the type of a variable
- This is handled differently when implementing the compiler in an object-oriented language or a functional language

Object-oriented

In OO languages it is easy to manage state, simply by using a local variable which is updated, or an object field.

Functional

In pure functional languages it can be tiresome to carry around state. Here a state monad can conveniently deal with state.

The state monad provides a convenient way to carrying around state in Haskell.

```
data CompileState = ...
```

```
type CompileMonad a = State CompileState a
```

For debugging purposes it is often convenient to use the state monad transformer on top of the IO monad.

This allows for easily printing debug-information.

```
data CompileState = ...
```

```
type CompileMonad a = StateT CompileState IO a
```

Submission Guidelines

Oskar (our course TA) kindly implemented a test suite last year.

Get the test suite from the course repo:

`https://github.com/myreen/tda283/tree/master/tester`

The test suite is still fairly new, so please report bugs or issues. We may accept pull requests on github.

The first component of the test suite: `testing.py`

This checks that your submission tarball has the correct format and contents, and passes some simple tests for each assignment component.

The `README.md` explains the submission format.

Make sure your submission passes these tests.

The test suite also contains a docker image. We will test your assignments within that environment.

Use the image to check your submission builds and runs in our environment.

We can add more packages to the environment if you have a good reason to use them and they're easy to install.

Instruction on docker in the repo and at <http://www.docker.com/>

Hint: Windows users etc might want to run docker within a VM.

Submission will be on Fire, which will be set up by next lecture.

Check your submission runs in the Docker environment.

BNFC 2.8.3 is installed in the image.

Haskell users: stack is installed.

- includes various default packages
- we recommend you don't use the Haskell LLVM package (not compatible with recent LLVM)

Java users: OpenJDK 11.0.6 is installed, also JLEX and CUP.

- includes the JDK standard library

C++ users: flex, bison and the STL will be available.

- we recommend you only attempt the course in C++ if you have good experience with C++ already
- contact us and tell us what other packages you need

It looks like most people have found a lab partner by now.

Part A is due next week. Get started soon!

The Fire submission system will be set up soon.

Good luck with it! Contact us if you're having trouble with any of the tools.