Universitetet i Oslo
Institutt for Informatikk
Reliable systems
Martin Steffen

INF 5110: Compiler construction

Spring 2024 — **Oblig 2** — 2. 04. 2024

Issued: 2. 04. 2024

1 Official

The **deadline/frist** for the second oblig is

Wednesday, 15. May 2024

In this handout, I refer to information in files and directories. Part of the information is contained in the **git-repository** that you have cloned for the first oblig already (though it was not relevant at that time, and I adapted the documentation like this handout and the slides to the 2024 round in the meantime).

I refer to the root of the repos as

`<COMPILAROOT>`.

On the net, it corresponds to the browsable version in the repository hosted by github under

`https://github.uio.no/compilerconstruction-inf5110/compila`

In your own **working directory,** it will probably carry the name `compila00-<xx>`, unless you have chosen to rename it.

2 What and how to hand in

2.1 Git

You will continue with your group's git-repos you used in the first oblig. Basically, you continue with your previous code, add the new functionality, push a solution before the deadline, and inform me when it's done so that I can pull your solution via git. It's important to tell me, as I don't want to repeatedly update in the hope that it's done.

In case your previous code of oblig 1 was not fully functional, you of course get first the parser in a workable state.

If a change in arrangement is needed (merge of groups, or a split of groups), you need to ask for that re-arrangement (not just announce on the day of the deadline that there is now a new group ...).

See also the *Readme* of the *"patch"* under

`<COMPILAROOT>/oblig2patch/oblig2patch/Readme.org`

2.2 What to include into a solution

As before, it should be an appropriately commented repos, solving the tasks of oblig 2. In particular needed is (basically as before)

- A top-level *Readme-file*[1] containing
  - containing names and emails of the authors (as before)
  - instructions how to build the compiler and how to run it (as before).
  - test-output for running the compiler on `compila.cmp` as input
  - of course, all code needed to run your solution
  - the Java-classes for the syntax-tree
  - the build-script `build-oblig2-inspiration.xml` (adapted)

Of course, the old code (for lex and yacc-based parsing) is still needed. It's not needed that both versions of the grammar, required for oblig 1, are still supported, one working version is enough.

3 Purpose and goal

The goal of the task is to collect more practical experience implementing a compiler, in particular, a taste of phases after parsing. It's only a taste, as we don't have the time to get a full-scale compiler on its feet. The language we are compiling is (as before) described in the **compila 24 language specification.** This time, also the later sections about type checking etc, that were irrelevant for oblig 1, specify the scope of the task as far as the language features are concerned.

Testing becomes more important than in oblig 1. It's *necessary* that a solution is equipped

with "automatic test-cases"

That can be done (as before) via ant targets. Those tests have to be executable on the RHEL linux pool at the university.[2]

4 Tools

The tools are basically the same as for the previous oblig, and typically you will continue anyway with the previous set-up.

5 Task more specifically: Type checking and code generation

The task is to extend the parser and AST generation with type checking and code generation. The rules governing the type checking and other restrictions are described in the language specification already (in the later sections). The "semantics" is *not* specified, but the language is so simple that it should basically be clear what a compila program is supposed to do.

The target "platform" is described in a separate document (which was already made available as part of the git-repos). It's also browsable under

[1] Many did a `Readme.md` which is a good format.

[2] That should actually not be a big restriction, as Java (and the task) is to a big extent platform independent ("write once, run everywhere" ...). Nonetheless: Based on experience with the earlier years (this year actually no real problems occured on my side in oblig 1, though one group reported that the initial starting point initially did not work at their side): it's advised to make this "test" setup early on (not after the deadline), to design the code *with the goal that it runs also at a different place than one's own platform* and to test that this goal is actually met. The reason for that "testability" requirement is that correction will again not be based on reading much code from my side, but in first approximation: running the tests. In that sense, it's also not of primary importance, whether it's `ant` or perhaps *make* or some script. Important is, that I can execute it by invoking a simple command. I don't have the time to figure out how one particular solution is configured, started, etc. And I don't want to look around and try whether I find a `main` method somewhere...

`<COMPILAROOT>/doc/bytecodeinterpreter/bytecodeinterpreter.pdf`

Tests

The tests that need to be successfully run for oblig 2 are

1. testing the type checker resp. semantic analysis
2. testing the code coge generator

The tests are located as follows relative to the `COMPILAROOT`-directory

`./src/tests/semanticanalysis/`

`./src/tests/fullprograms/runme.cmp`

They are already contained in **your git-repos** at the descibed places (unless you have reorganized it).

Patch

Obtain the patch (as zip-archive) under:

`https://github.uio.no/compilerconstruction-inf5110/compila/tree/master/oblig2patch/oblig2patch.zip`

or via an *updated* clone of the course repos. Read also the Readme.org there, there is more info about how to start. Note: when you cloned or downloaded the repository in perhaps Februrary there had been already some *oblig2patch* as part of the repos. Also that should be more or less usable, but it reflects the **2024 version**. Not much has changed since then (mostly the documentation like this **handout** here and the *slides*, the actual "content" is basically the same.

Why is it called patch?

It's called patch, because your compiler needs not just *new classes* for instance for type checking, some already existing files also need adaptation and changes. For oblig 1, I provided for example a file `Compiler.java`, most took it as starting point (but may have adapted it as well). Since I have no control what individual groups did with that file I cannot hand out an **actual patch** in the technical sense. Instead, I give a *new version* of `Compiler.java` as part of the patch-package, that should be used as **inpiration** of what probably needs to be adapted in your version. The files `Tester.java` and `FileEndingsFilter.java` are meant to be used for testing, in particular to **automate** the tests provided. Again, since I don't control how you solved oblig 1, it's possible that they require adapdation. At least they need to be integrated to your implemantation, i.e., moved out from the silly directory *oblig2patch* to a better suited place (and integrated into your build-set-up).