Universitetet i Oslo
Institutt for Informatikk

Reliable system
Martin Steffen

INF 5110: Compiler construction

Spring 2020

Oblig 1

03. 02. 2020

Issued: 03. 02. 2020

1 Official grading guidelines

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

Friday, 06. 03. 2020, 23:59

Requirements:

- the test program parses 100% ok (see below)
- the printed AST reflects the correctly parsed structure (especially, the correct associativity and precedence). The exact same formatting as in the provided illustration is not required; reasonable deviations are fine.
- the instructions under "What and how to hand-in" need to be followed
- the solution needs to compile and run *at the UiO pool*.[1] Test it!

---

2 What and how to hand in

2.1 Git

As the previous two times, we use

git

to hand in the obligs.

In earlier years it was mostly via email, 2017 via devilry, but since it's about a a piece of software, something like git is more appropriate. Likewise: oblig 2 will extend your oblig 1, so the code will evolve. Finally, we strongly encourage working in groups, so git allows to share things.

I assume that most are in principle familiar with `git`, if not, ask me (or bring yourself up to speed otherwise). I suggest, that we use the uio-internal git server[2]

`github.uio.no`

In order to hand-in via git, each group has to do the following steps.

[1] Statements like "but on my laptop it worked, I can show you" don't cut it ....

[2] Actually, it's the UiO enterprise github server. Github is just one particular "web-interface" on top of git and there are also alternatives on the market, but for uniformity, you may as stick to the UiO git / github server.

1. everyone: if not already: create yourself a account at `github.uio.no`. Your UiO login allows you to do that.

2. per group: *create* a new *project*. If your are in group number n,[3]

   call your project `compila<n>`

   The "parentheses" `<` and `>` are *not* part of the name! If the group contains more than one person, the creator has to add the partner as *collaborator.*

3. If your project is "private": add me as collaborator (login `msteffen`). I don't need to contibute as collaborator, but I need access.

4. Send me an *email* with the link, mentioning the names (and login) of the members of the group as confirmation. That needs to be done *before the deadline.*The names of the member of the group should alsp feature *prominently* on the top-level of the repository, as in the top-level `Readme`.

2.2 What to include into a solution

- A top-level *Readme-file* containing
  - containing names and emails of the authors
  - instructions how to build the compiler and how to run it.

  The top-level Readme should describe in a consise manner instructions for building, installation, running, and testing your compiler in a manner useful for an interested *user*. The target audience is a master-level computer science student or someone how is not afraid to git-clone a repository (or download a jar-file or similar) and following a few clearly installation steps. The user is not expected to figure out himself or herself how to install, run, and test it. The user is also /not/ expected to be a student of this compiler course and familiar with the specification of the compila-language and the oblig-documentation.

  It might be nice to use some *markdown* format (like a `Readme.md`. It's some form of poor-man's markup like HTML, and is typically rendered nicely by browsers. The current top-level Compila-Readme is not in the `md` format, but `Readme.org` (which is a similar format and also rendered nicely by github).

- Additional information should be provided, which is not relevant for that mentioned "interested user", but that's insider information relevant for the oblig and the technical realization. This it's better not in the top level readme but at a different place or file.
  - test-output for running the compiler on `compila.cmp` as input
  - of course, the /code/ needed to run your package. That includes
    - *JFlex*-code for the scanner
    - *CUP*-cpde for the 2 variants of the syntax
    - the Java-classes for the syntax-tree
    - the build-script `build.xml`[4]

3 Purpose and goal

The goal of the task is to gather practical experience of the following tools and technies.

- use scanner/lexer and parser tools. In this case `JFLex` and `CUP`.

- rewrite and massage a grammar given in one form into another one so that it's accepted by the tools. In our case, the language is given in some EBNF, which has to be adequately rewritten so that it can be fit into the lexer and parser tools

[3] The project names need to be different so that, for correction, one can distinguish them by their "name" (not that all projects are called "compila")

[4] Alternatively, you can use a makefile if you prefer that. In that case, remove the `build.xml`-file.

- handle associativity and precedence of syntactic contructs in two possible ways
  - formulate a (unambiguous) grammar that embodies the correct precendences and associativies
  - work with an ambiguous grammar, but instruct the parser tool (like CUP) to result in an appropriate parser.
- design and implement an suitable AST data structure. Use the parser to output your AST (in case of a successful parse).
- do a "*pretty printer*" in the following sense: implement some functionality that *outputs* and AST in a "useful" manner. In particular, the parenthetic tree structure must be visible from the output (i.e., one can see whether the associativity and the precedence is correctly implemented).

4 Tools

The platform is *Java*, together with the auxiliary tools

- *JFlex* (scanner generator in the (f)lex family)
- *CUP* (parser generator in the yacc family)
- *ant* (a kind of "make" tool specialized for Java)

The tool ant is available at the RHEL pool at IFI, for other platforms I don't know, but it's freely available. *JFlex* and *CUP* are provided.

If, for some reason, you plan to deviate from the suggested tools, you

1. **MUST** discuss that first with the lecturer/
2. it **must** be a platform which is freely available at the university RHEL pool resp. is platform independent. Proprietary tools or tools I don't have easy[5] access to cannot be used. If using Java, it must compile and run without support of specific development environments or "frameworks" besides the ones mentioned (*JFlex*, *CUP*, *ant*).

5 Task more specifically: Syntax check and parsing

The overal task is to

> implement a parser for the *Compila 20* language.

The language specification is given in a separate document. Oblig 1 is concerned with checking *syntactic correctness*, which means, not all of the language specification is relevant right now: semantic correctness, type checking etc. will become relevant only later for the second oblig.

5.1 Syntax tree

The result of a successful parse is an *abstract syntax tree*. That data structure needs to be appropriately "designed". In a Java implementation, that involves the definition of appropriately chosen classes arranged in some class hierarchy. Make also use if *abstract classes*. In the lecture, there had been some "design guidelines" that may be helpful. Carefully chosen names for classes will help in a conceptually clear implementation. A definitely *non-recommended* way is to have one single class `Node` lumping together all kinds of nodes and syntactic categories in the syntax tree.

[5] I *mean* easy and in the sense that it does not cost time to install the required environment or to figure out how it all hangs together. *Not* "easy" as in "it's really not hard after you read some manuals and with the help from the fine folks on stack-exchange" ...

5.2 Print out of the AST

The AST should be "printed". The easiest and recommeded form of printout is in *prefix form*. Under `material/sample-compila-ast`, there is an example compila input file and a corresping file containing a possible output. The two files are called

- `complexaddition.cmp`
- `complexaddition.ast`

Note: the two files are meant as *inspiration.* Each year the syntax of *compila* slightly changes (wrt. keywords, associativity etc). So the syntax is 100% in accordance with the 2020 version (but pretty comparable).

The one that should be used for this task (AST-printing) is the following

```
./src/tests/fullprograms/complexaddition.cmp
```

and it should be consistent with this year's gammar (fingers crossed).

It's allowed (but not necessary) to print it in other forms than prefix form. But the output must indicate the AST in readable form ("readable" as in human-readble that is ...). Note, the task is not that the output is a syntactically correct *compila* program again (that might be a formatting tool), we just need a way to look at the syntax tree, which comes in handy for debugging,

5.3 Two grammars

As mentioned shortly, the task requires 2 grammars, representing 2 ways dealing with precendence and associativity.

1. an *unambiguuous* grammar resolving precedence and associativity by "baking it in" directly into the grammar. The grammar is in plain BNF (in the form required by the tools)
2. the second grammar is ambiguous and relies on *CUP* to resolve the associativity and precendence. This second grammar will probably look nicer and will be shorter. It's therefore probably best to take that one as *default* (for instance for oblig 2).

Comparison and discussion

Investigate and characterise *conflicts* of the *original* grammar. How many states do the 2 generated CUP grammars have? That requires a look into the CUP-generated code. Discuss also whether the choice of the two grammars influences the generation of the AST: is one of the two approaches easier to work with when it comes to generate an AST (resp. your chosen AST data structure.

Note: It's not required to provide code to build *two* versions of AST-generation, one is enough. In other words, for one of the two grammars, you don't need "action code" in the grammar to produce an AST, plain *checking* is sufficient.

5.4 Lexical analysis

As mentioned, *JFLex* is the tool of choice for lexical analysis. It delivers a token to the parser via the method `next_token()`.

As far the the "theoretical" task concerning *compila 20* is concerned, the lexer is responsible for ignoring comment, white-space etc, find keywords and the like.

Besides that, one has to make the parser and the lexer "work together" hand in hand. Information about that can be found in the corresponding manual. There should also be examples for inspiration. A crucial ingredient is the interface `java_cup.runtime.Scanner` which needs to be implemented by the actual scanner. The scanner will hand over tokens of the type `Symbol` and one can use `Symbol.value` to pass "text" or other objects from the lexer to the parser.

5.5 Error handling

Error handling can be done simple: When hitting an error, parsing should stop (as opposed to try to continue and give back an avalanche of subsequent errors). Some meaningful error message (at least wrt. which syntactic class caused the error) would be welcome, as opposed to a plain "sorry, bad program". It's not required to give back line numbers referrring to the original source code or positions in the original file. In practice that's definitely useful (and not very hard either), but not required for the oblig.

5.6 Tests

For testing, there is a bunch of files under

./src/tests

They are supposed to contain syntactically correct programs for this year's version, with exception of the ones under

`./src/tests/errors`

For oblig one, you are requested to generate the

`./src/tests/fullprograms/complexadditions.cmp`

as part of the oblig as mentioned, but you may of course use the other test program to see how robust your implementation is. Especially the error-programs later will become relevant for oblig2, when we do type checking; there, some syntactically ok programs should be flagged as erroneous by the type checker.

6 Resources

The web-page (the git-one) will contain also links to *JFlex CUP* and corresponding manuals.