Universitetet i Oslo
Institutt for Informatikk
PMA
Martin Steffen

INF 5110: Compiler construction

Spring 2018

Oblig 1

26. 02. 2018

Issued: 26. 02. 2018

1 Official grading guidelines

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

Friday, 23. 03. 2018, 23:59

Requirements:

- the testprogram 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

This year, we try whether handing it in via

git

is a good way. In previous years it was mostly via email, 2017 via devilry, but since it's about a a piece of software, let's try git this time. 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]

[1] statements like "but on my laptop it works, 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, let's just stick to the UiO github server.

`github.uio.no`

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

1. everyone: if not already: create yourself a account at `github.uio.no`. You 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. Assuming that the 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 at latest *before the deadline.*The names of the member of the group should als 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.
  - test-output for running the compiler on `compila.cmp` as input
  - of course, all 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`

3 Purpose and goal

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

- 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
- 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.

[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")

- 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 teacher
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[4] 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 18* 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.

[4] 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*. In the provided "starting point", there is an example compila input file and a corresping file containing a possible output. The two files are called

- `compila.cmp`
- `compila.ast`

Note: the two files are meant as inspiration. Each year the syntax of *compila* slightly changes (wrt. keywords, associativity etc). So it might not 100% in accordance with the 2018 version.

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 18* 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 meaningfull 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.

6 Resources

The course web-page contais an html file `resources.html` which collects clickable links to *JFlex* *CUP* and corresponding manuals.