02141 Computer Science Modelling
Practical Assignment 2022

1 Overview

The overall goal of the practical assignment is to build a tool for running and analysing programs written in a variant of the Guarded Command Language (GCL). The tool you will develop can be seen as a basic version of `formalmethods.dk/fm4fun` with a text-based user interface.

The assignment is divided into seven tasks. Each task is devoted to a module of the tool (a parser, a compiler, an interpreter, a verifier, and several analysers). Each module should be runnable as a standalone program that takes as input a GCL program (and perhaps some additional input required by the individual analyses) and produces a result.

The overall structure of the assignment is illustrated below; you may want to use it as a guideline for structuring your implementation.

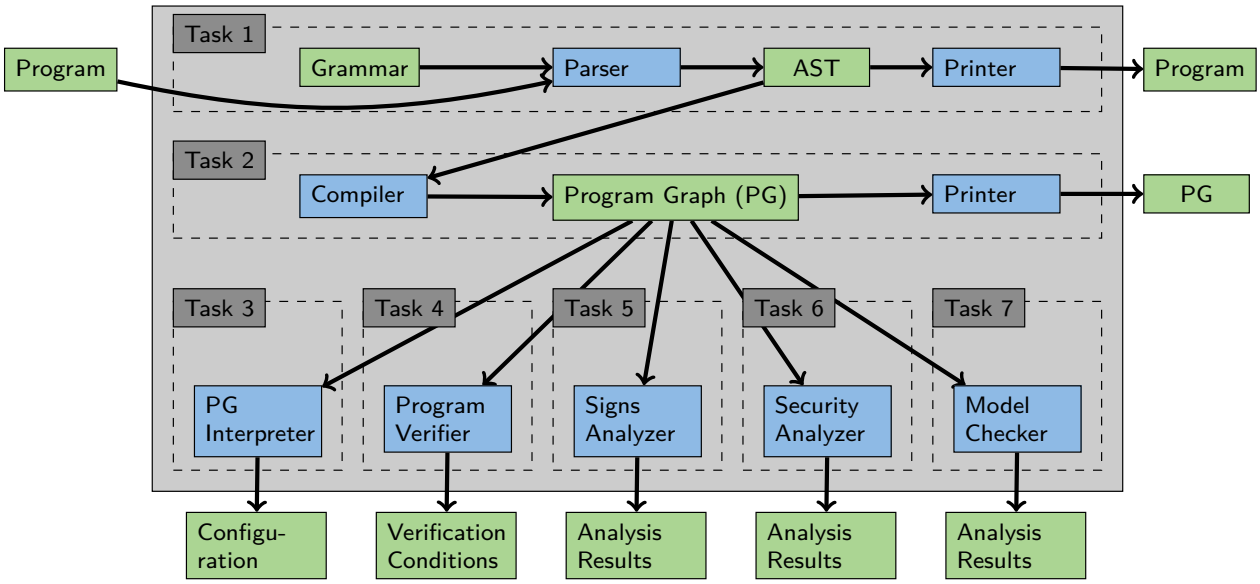

Figure 1: Structure of the practical assignment.

This document first describes the guidelines for working on the practical assignment. After that, it describes the programming language that we will work with, followed by a discussion of each task. Finally, we describe several challenges, which are optional but allow you to continue working on aspects that you may find particularly exciting.

2 Guidelines for the Practical Assignment

Please carefully read the following general guidelines for working on the practical assignment.

Requirements

1. The project must be done in teams of size 3.
2. There is no constraint on specific programming languages or parser generators to be used. However, the TAs and the teachers will provide support on F#/FSLexYacc only.
3. There is no requirement on how input and output should be handled. One option is to use the standard input and ouput. Another option is to use text files.
4. You have to implement the techniques presented in the teaching material.

Software installations and guidelines. Guidelines for installing and using parser generators are at `https://gitlab.gbar.dtu.dk/02141`. Installation of software is your responsibility. Inquiry the TAs only after having invested enough time trying to find a solution, and after having asked other teams for help.

Feedback and testing We will *not* publish solutions of any practical assignment tasks. Instead, we ask you to proactively seek feedback by testing your solution as follows

For each task prepare a set of (manually or automatically generated) test cases. Use the test cases to test your project against `formalmethods.dk/fm4fun`. Do you obtain the same results? If not, reflect on which of the two tools is providing the correct result. Use the test cases to test your project against the project of another team. Do you obtain the same results? If not, reflect on which of the two tools is providing the correct result.

Challenge other teams to "hack" your project: can they find cases where your project provides wrong results? Once one or more other teams agree your solution seems to be correct, ask the TAs or the teacher for feedback. For every task, the teachers will keep a scoreboard covering the progress of individual teams; participation is voluntary.

Submission & Evaluation

- Submission is through DTU Learn. For each task, we ask you to submit your solution on DTU Learn within the specified deadline. You have to submit well-commented code and instructions (e.g. a README file) explaining which are the key files and how to install and run the project, including a description of how to provide input and interpret output.
- We recommend submitting a reference to a specific version of a software repository (githb and similar); this way, you can also share your project with other teams to obtain feedback.
- You can continue working on the module of a task after the corresponding deadline has passed. For example, if you discover bugs in the parser while doing the interpreter.
- Practical assignments do not contribute to the grade. Recall the course description: "practical assignments provide a good background for learning the methodology of the course as will be tested at the exam."

3 Programming Language: Yet Another GCL Variant

The variant of GCL that you have to consider throughout the tasks is the following subset of the language used in `formalmethods.dk/fm4fun`:

```
 C ::= x := a | A[a] := a | skip | C ; C | if GC fi | do GC od
GC ::= b -> C | GC [] GC
 a ::= n | x | A[a] | a + a | a - a | a * a | a / a | - a |  a ^ a | (a)
 b ::= true | false | b & b | b | b | b && b | b || b | !b
     | a = a | a != a | a > a | a >= a | a < a | a <= a | (b)
```

This GCL variant can be seen as a language in between the one presented in [Formal Methods, Definition 2.3] and the one used in `formalmethods.dk/fm4fun`.

The syntax of variables and numbers, and the associativity and precedence of operators must be the same as in `formalmethods.dk/fm4fun`, which can be read by clicking on the question mark besides "Examples". We reproduce part of them here for your convenience:

- Variables `x` and arrays `A` are strings matching the regular expression `[a-zA-Z][a-zA-Z\d_]*` and cannot be any of the keywords.
- Numbers `n` match the regular expression `\d+`.
- A whitespace matches the regular expression `[\u00A0 \n \r \t]`, with a mandatory whitespace after `if`, `do`, and before `fi`, `od`. Whitespaces are ignored anywhere else.
- Precedence and associativity rules:
  - In arithmetic expressions, precedence is highest for `-` (unary minus), then `^`, then `*` and `/`, and lowest for `+` and `-` (binary minus).
  - In boolean expressions, precedence is highest for `!`, then `&` and `&&`, and lowest for `|` and `||`.
  - Operators `*`, `/`, `+`, `-`, `&`, `|`, `&&`, and `||` are left-associative.
  - Operators `^`, `[]`, and `;` are right associative.

In the rest of the document GCL refers to the above language.

4 Tasks

Task 1: A parser for GCL. The goal of this task is to implement a parser for GCL that accepts or rejects programs and builds AST for them, thus working like the syntax checker of `formalmethods.dk/fm4fun`. The parser must take as input a string intended to describe a GCL program and must build an AST for it. In addition, the program must produce compilation results: it should return whether the input is a program accepted by the GCL grammar specified above. You should also implement a "Pretty Printer" module that prints the AST so you can easily check your solution.

Hints: Use a parser generator as seen in class. Start with the grammar as given above and adapt it to your parser generator. You may need to specify precedence/associativity of some operators in the parser generator language, or by applying some of the grammar transformations seen in class. Your parser needs to generate abstract syntax, which you will need in task 2.

Submission deadline: March 10, 23:59

Task 2: A compiler for GCL. The goal of this task is to implement a compiler of GCL programs into Program Graphs (PGs) similar to results you obtain under "Program Graph" in `formalmethods.dk/fm4fun`. The program must take a GCL program as input. In addition, your compiler must include a way to specify a flag to construct either deterministic PGs or a non-deterministic PGs (emulating the corresponding buttons on `formalmethods.dk/fm4fun`). The compiler must produce a PG in the output. The compiler must produce a PG as output in the textual graphviz format used by the export feature on `formalmethods.dk/fm4fun`.

Hints: Enrich the parser developed in Task 1 so that it exploits the abstract syntax for GCL programs. Follow [Formal Methods, Chapter 2.2] to construct a PG for a GCL program. You can use graphviz tools to visualize the PGs that your compiler produces.

Submission deadline: March 14, 23:59

Task 3: An interpreter for GCL. The goal of this task is to implement an interpreter for GCL programs that works similarly to the environment "Step-wise Execution" on `formalmethods.dk/fm4fun`. The interpreter must take a GCL program as input. In addition, you will need a way to specify initial values of the variables (e.g. reading them from a file or from the console input). The interpreter must output the status of the program (terminated/stuck) and the configuration (node and memory) when the program stops. The format for the output can be like in this example:

```
status: terminated
Node: qFinal
x: 13
y: 7
```

i.e. a line with with the actual status (`terminated`/`stuck`), and a line for the current node in the program graph, followed by a line for each (defined) variable in the program, with the name and value of the variable.

Hints: Follow [Formal Methods, Chapter 1.2] and [Formal Methods, Chapter 2.3] to build an interpreter based on the semantics of GCL programs and their PGs. You can start considering deterministic PGs first. If you have time you can extend your interpreter to deal with non-deterministic PGs.

Submission deadline: March 24, 23:59

Task 4: Program Verification. Implement support for program verification as suggested in [Formal Methods, Appendix B]; your implementation should extract the relevant proof obligations as supported by `www.formalmethods.dk/fm4fun/`. Moreover, your tool should generate *logical formulas* (instead of just sequences of actions as on FM4FUN) as proof obligations, similarly to what has been discussed in the lecture and in [Formal Methods, Teaser 3.12].

Give at least one example (a GCL program) including at least one non-trivial loop, where you also managed to (manually) discharge the generated proof obligations.

Hint: You are free to choose the details of your proof obligation's format but you may want to have a look at existing formats for logical formulas, such as the one described in Challenge 5.

Submission deadline: April 4, 23:59

Task 5: A sign analyser for GCL. The goal of this task is to implement a tool for sign analysis of GCL programs that works like the one available under environment "Detection of Signs Analysis" on `formalmethods.dk/fm4fun`. The sign analysis must follow the approach in [Formal Methods, Chapter 4]. The tool must take a GCL program as input. In addition, you will need a way to specify/input (multiple) abstract initial values of the variables in the tool (e.g. reading them from a file or from the console input). The tool must output the result of a sign analysis for the variables. The variables and their signs must be printed in the same order as `formalmethods.dk/fm4fun` does. An example of the output of an analysis that produces three abstract memories should look like this:

```
x y z
+ - -
- + -
0 + 0
```

Hints: Enrich the parser as you did in Task 3 and follow [Formal Methods, Chapter 4] for implementing the sign analysis.

Submission deadline: April 21, 23:59

Task 6: A security analyser for GCL. The goal of this task is to implement a tool for security analysis of GCL programs, that works as a simplified version of the environment "Security Analysis" on `formalmethods.dk/fm4fun`. The security analysis must follow the approach in [Formal methods, Chapter 5.3-5.4]. The tool must take a GCL program as input. In addition, you will need a way to specify the "Security lattice" and the "Security Classification for Variable" (e.g. reading them from a file or from the console input). The security analysis tool must print/output the result of the security analysis i.e. print lines with the following information:

- Actual flows
- Allowed flows
- Violations
- Result (`secure`/`not secure`).

Hints: Enrich the parser as you did in Task 3 and follow [Formal Methods, Chapter 5.4] for implementing security analysis. Base your analysis on deterministic PG.

Submission deadline: May 5, 23:59

Task 7: A model checker for GCL. The goal of this task is to implement a simple model checker for GCL programs. The tool must take a GCL program and an initial configuration as input. The model checker must then explore its transition system to detect reachable stuck states (i.e. stuck or terminated configurations).

The tool must print all reachable stuck states in the transition system with the same format as in task 3. The tool must support non-deterministic programs.

Hints: Base your solution on the definition of transition systems and stuck states in [Formal Methods, Chapter 6.1], on the construction of transition systems for PGs described in the first page of [Formal Methods, Chapter 6.4], and on the following pseudo-algorithm:

Visited = $\emptyset$;
ToExplore = { initial state };
while ToExplore $\neq \emptyset$ **do**
    remove some state s from ToExplore;
    if s $\in$ Visited **continue**;
    Visited := Visited $\cup$ {s};
    if $\mathbf{Reach}_1$(s) = $\emptyset$ then report stuck state s and **continue**;
    for each state s' $\in$ $\mathbf{Reach}_1$(s) **do**
        add s' to ToExplore;

Also, be careful of undesired side effects, e.g. avoid using shared variable/structures between states.
NOTE: You do not need to consider CTL, atomic propositions and the labelling function.

Submission deadline: May 6, 23:59

5 Challenges

If you have completed the tasks of the practical assignment you may consider the following challenges, which are aimed at further reinforcing and broadening your knowledge and skills. Note that some of the challenges go beyond the teaching material of the course, and also note that they demand additional effort and you should prioritize first the main activities and learning objectives of the course.

1. **More GCL**. Extend your compiler and interpreter to cover richer variants of GCL. Some options are:

   - control structures in the bonus material [Formal Methods, Chapter 2].
   - concurrency, as covered in [Formal Methods, Chapter 8] and supported by `formalmethods.dk/par4fun`.
   - procedures, as covered in [Formal Methods, Chapter 9] and supported by `formalmethods.dk/rec4fun`.
   - additional data types.

2. **Model Checking CTL.** Go beyond the deadlock checker implemented in **Task 6** and implement a model checker for CTL properties [Formal Methods, Chapter 6]. You may start with support for simple one-step operators (**EX**ϕ, **AX**ϕ), move to basic temporal operators (**EF**ϕ, **AF**ϕ, **EG**ϕ, **EG**ϕ), and last consider the until operators (**E**ϕ_1**U**ϕ_2 , **A**ϕ_1**U**ϕ_2).

3. **Smart Deadlock Detection.** Your solution to **Task 6** can be used to determine if a GCL program is deadlock-free. If you have extended your tool to support concurrency (see challenge 1) you will discover that it takes some time to fix deadlocks and achieve deadlock freeness. Finding deadlocks fast and providing short traces leading to deadlocks is useful in this process. Consider implementing smart strategies to improve the deadlock-finding capabilities of your tool. For example, the step "remove some state s from ToExplore" of the provided pseudo-algorithm could be based on some smart heuristic for selecting the next state to explore.

4. **Lexing and Parsing.** Implement your own lexer and parser instead of using a parser generator. Lexers are essentially based on the material in the RL part of the course (tokens can be described by regular expressions, for which we can build token-recognizing DFAs). For building a parser, you may consider a top-down approach, e.g. based on a recursive-descent parser (partly covered in the CFL part of the course).

5. **More Program Verification.** Extend the implementation of your program verifier to output proof obligations in a standardized format for first-order predicate logic called SMT-lib, see `https://smtlib.cs.uiowa.edu` for a detailed description of the SMT-lib language. You can then attempt to automatically check your generated proof obligations by feeding them to an automated solver, such as Z3 (`https://github.com/Z3Prover/z3/wiki`).