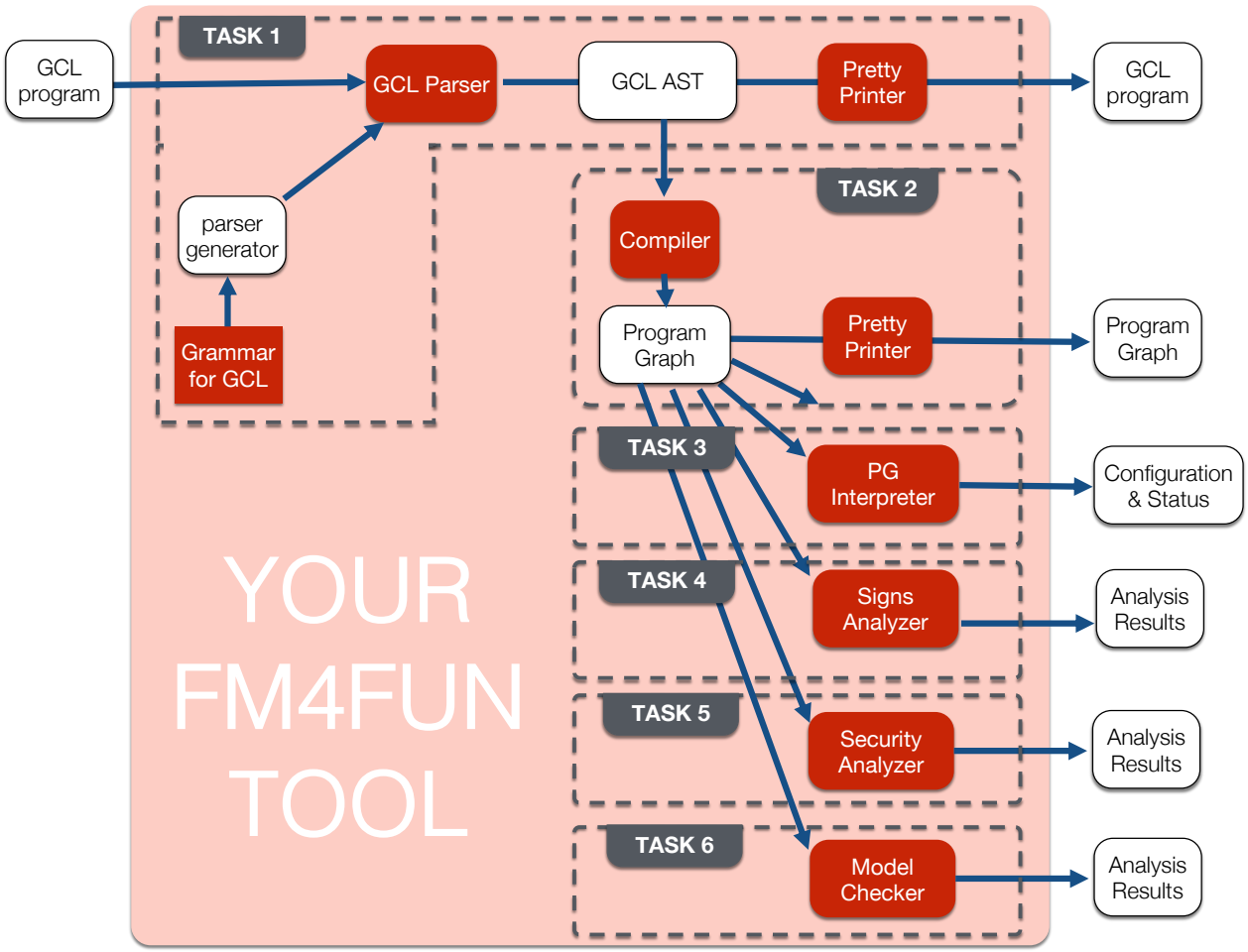
02141 Computer Science Modelling
Mandatory Assignment

Overview

The overall goal of the 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 interface.

The assignment is divided in tasks. Each task is devoted to a module of the tool (a parser, a compiler, an interpreter and several analysers). Each module should be runnable as a standalone program that takes as input a GCL program and produces a result.

The overall structure of the assignment is illustrated below and it can be used as a guideline for structuring the implementation.



Yet Another GCL variant

The variant of GCL that you have to consider 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)
 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` 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+`. Negative numbers are expressed using the unary minus operator `-n`.
- A whitespace matches the regular expression `[\u00A0\f\n\r\t\v]`, 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.

Tasks

Task 1: A parser for GCL. The goal of this task is to implement a parser for GCL that accepts or rejects programs, thus working like the syntax checker of `formalmethods.dk/fm4fun` but in a simplified way. The parser must take as input a string intended to describe a GCL program and must produce compilation results: whether the input is a program accepted by the GCL grammar specified above, and if not some hints to correct the program.

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 12, 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 and must produce a PG in the output. The format of the output must be the same textual graphviz format used by the export feature of `formalmethods.dk/fm4fun`. Your compiler must include a flag to construct either a deterministic PG or a non-deterministic PG (emulating the corresponding buttons in `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 19, 23:59

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

```
status: terminated
Node: qFinal
x: 13
y: 7
z: 0
```

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

Hints: Enrich the parser developed in Task 2 so that it produces abstract syntax for PGs. 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.

Submission deadline: April 2, 23:59

Task 4: 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" in `formalmethods.dk/fm4fun`. The tool must take a GCL program as input and must print the result of a sign analysis for the variables at the end of the computation. The sign analysis must follow the approach in [Formal methods, Chapter 4]. You will need a way to specify the abstract initial values in the tool (e.g. reading them from a file or from the console input). 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 23, 23:59

Task 5: 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" in `formalmethods.dk/fm4fun`. The tool must take a GCL program as input and must print the result of the security analysis:

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

The security analysis must follow the approach in [Formal methods, Chapter 5.4]. 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).

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 4, 23:59

Task 6: 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 and must 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;

NOTE: You do not need to consider CTL, atomic propositions and the labelling function.

Submission deadline: May 11, 23:59

Additional Information

Guidelines and software installations 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.

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. Submission is through CampusNet. For each task you need to do a submission on CampusNet within the specified deadline.
5. 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.
6. You have to implement the techniques presented in the teaching material. If you implement alternative techniques you have to provide a report describing them in detail. For example, if in Task 4 you decide to implement your own analysis technique technique instead of the one descriped in [Formal methods, Chapter 4], you have then to provide in a report a formal description of your technique, including its proof of correctness.
7. 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. You can also submit a reference to a specific version of a software repository (github and similar). You don't need to submit a report unless you deviate from the teaching material (cf. above).

Feedback and testing Proactively seek feedback by testing your solution:

- For each task prepare a set of (manually or automatically generated) test cases (GCL programs).
- 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?
- In case of disagreement on a result, ask the TAs or the teacher for feedback.

Evaluation Mandatory assignments do not contribute to the grade. Recall the course description: "Mandatory assignments provide a good background for learning the methodology of the course as will be tested at the exam."