EECS4302 Winter 2020
Project
(30% of your grade)
Design and Implementation of a Compiler to Support Program Verification

Chen-Wei Wang

Due: **11:59pm, Monday, April 6**

- **Check the Amendments section of this document regularly for changes, fixes, and clarifications.**
- **Ask questions on the course forum on the moodle site.**

Policies

- You are allowed to form **a tema of one member, two members, or three members** for this assignment.
  - When you submit your lab, you claim that it is **solely** your team's work. Therefore, it is considered as **an violation of academic integrity** if you copy or share **any** parts of your Java code, in any way, shape, or form, during **any** stages of your development.
  - When assessing your submission, the instructor and TA will examine your code, and **suspicious submissions will be reported immediately to Lassonde for further, official exploratory meetings**. **We do not tolerate academic dishonesty**, so please obey this policy strictly.
- **Your (submitted or un-submitted) solution to this assignment (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., a non-private Github repository) in any way, shape, or form. The department reserves the right to take necessary actions upon found violations of this policy.**
- You are entirely responsible for making your submission in time.
  - You may submit **multiple times** prior to the deadline: only the last submission before the deadline will be graded.
  - Practice submitting your project early **even before it is in its final form**.
  - No excuses will be accepted for failing to submit shortly before the deadline.
  - Back up your work **periodically**, so as to minimize the damage should any sort of computer failures occur. Follow this tutorial series on setting up a **private** Github repository for your Java projects.
  - The deadline is **strict** with no excuses: you receive **0** for not making your electronic submission in time.
  - Emailing your solutions to the instruction or TAs will not be acceptable.
- You are free to work on this lab on your own machine(s), but you are responsible for testing your code at a Prims lab machine before the submission.

Contents

1 Amendments

This project specification document was initially written in a way that intends to give you the maximum freedom to design and implement your compiler. However, additional requirements and clarifications may be added here.

2 Background Readings

– Tutorial videos on the ANTLR4 parser generator: `https://www.youtube.com/playlist?list=PL5dxAmCmjv_4FGYtGzcvBeoS-BobRTJLq`

3 Problem: Program Verification

Our society is increasingly dependent upon the behaviour of complex software systems. Errors in the design and implementation of these systems can have significant consequences. For examples (and many more recent examples can still be found):

– In August 2012, a fairly major bug in the trading software used by Knight Capital Group lost that firm $461m in 45 minutes[1].

– A software glitch in the anti-lock braking system caused Toyota to recall more than 400,000 vehicles in 2010[2]; the total cost to the company of this and other software-related recalls in the same period is estimated at $3bn.

– In October 2008, 103 people were injured, 12 of them seriously, when a Qantas airliner[3] dived repeatedly as the fly-by-wire software responded inappropriately to data from inertial reference sensors.

– A modern car contains the product of over 100 million lines of source code[4], and in the aerospace industry, it has been claimed that "the current development process is reaching the limit of affordability of building safe aircraft"[5].

One approach to improving confidence with software correctness is via *program verification*, which uses discrete mathematics (e.g., propositions, predicates, proof by induction) to prove or disprove the correctness of intended system behaviour, which is specified contractually and formally as preconditions, postconditions, and invariants (recall what you learned in the previous software design course). Formal verification can be useful in proving the correctness of systems such as source code written in a programming language.

To verify a program, we first need to construct (manually, or automatically if there is a "compiler" for it) a formal, mathematical model of the system/program under verification, and then prove or disprove (on paper or via a tool) that the mathematical model entails certain properties. For example, the Hoare Triple and weakest precondition[6] (which you learned in the previous software design course) formalizes the input program as predicates that can either be proved or disproved. There are verification tools available to facilitate the process of formalizing and proving properties of programs. To use such tools, you must *rewrite* (or *formalize*) the source program in therm of the input (specification) language of the verification tool in question.

[1] A. Massoudi (2012): Knight Capital glitch loss hits $461m. Financial Times.

[2] M. Williams (2010): Toyota to recall Prius hybrids over ABS software. Computerworld.

[3] Australian Transport Safety Bureau (2011): In-flight upset 154km West of Learmouth, WA, VH-QPA, Airbus A330-303. Aviation Safety Investigations and Reports AO-2008-070.

[4] R. N. Charette (2009): This car runs on code. IEEE Spectrum. Available at `http://www.spectrum.ieee.org/feb09/7649`.

[5] P.H. Feiler (2010): Model-based validation of safety-critical embedded systems. In: Aerospace Conference, IEEE, pp. 1 10.

[6] See: `https://www.eecs.yorku.ca/~jackie/teaching/lectures/2019/F/EECS3311/slides/13-Program-Correctness.pdf` for a review.

4 Your Task: Building a Compiler to Support Program Verification

You are required to build a compiler to support the verification of some programming langage, using the ANTLR4 parser generator. More precisely, your implemented compiler must tranform a programming language (conforming to a syntax of your design) into the input specification language of a verification tool (of your choice) for verification. It is required that your compiler takes as input a plaint text file, where lines are written in accordance with the syntax of the source programming language. Figure 1 summarizes the problem you are asked to solve for this project:

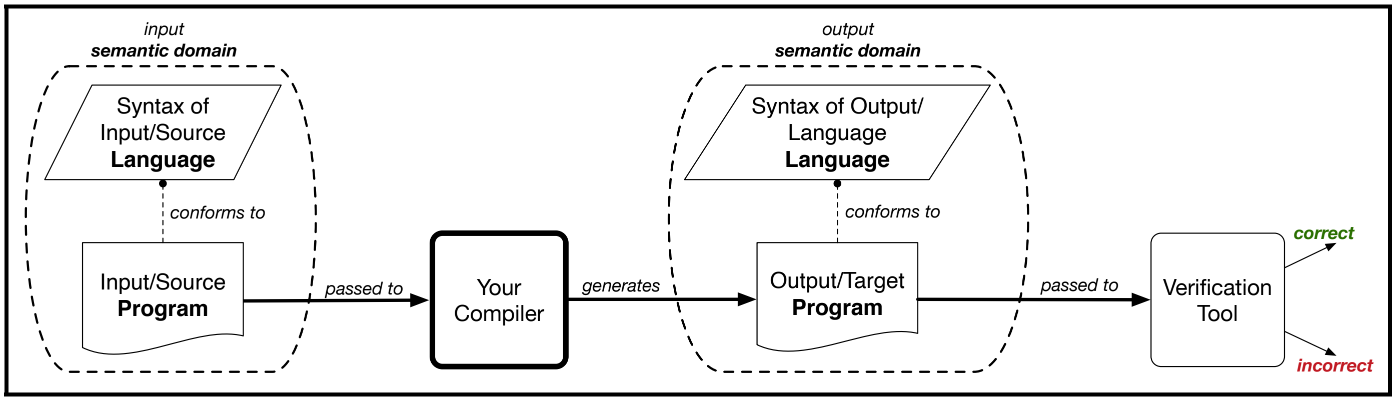

Figure 1: Problem: Building a Compiler to Support Program Verification

1. $\boxed{\text{Design the Syntax}}$ of Input/Source Language.
   - You will specify your design of tokens and rules in a `.g4` file for the ANTLR4 tool to generate the lexer and parser. You are free to choose to work with either the `-visitor` option or the `-no-listener` version.
   - What programming features should be supported in your source language for verification are completely up to your design. You are encouraged to incorporate features of your favorite programming languages. However, there is a list of minimum programming features that your compiler must support in order to receive a **passing mark**. See Section 8 for details.
   - Recall in your previous software design course that correctness of code relative to its specification. Consequently, in order for the source programming language to be supported for formal verification, part of its syntax must support some kind of specification (e.g., contracts).

2. $\boxed{\text{Implement semantic analysis}}$ on the source language such as type checking (e.g., variables are declared before they are used, potential danger of dereferencing of null pointers). As covered in the tutorial videos, some of these checks may be performed while the parse trees are being built by ANTLR, whereas some may be performed on (using the visitor design pattern) the tree-structured model object.

3. $\boxed{\text{Implement a transformation}}$ from the source input, if type-correct, into a semantically-equivalent output, written using the specification language of a verification tool that you choose.
   - In order to familiarize yourself with the input specification language of the verification tool you choose, it it not necessary for to study its formal grammar (which may or may not be available). Instead, the most effective way (just like how you learn any other new language) is to study the examples made available by the tool documentation site, or by contributors who also use the same tool.
   - You are given a fixed list of verification tools which you can choose. See Section 5.
   - Depending on the verification tool you choose:
     - If the target tool supports automated verification, then your compiler must directly call it[7].
     - If the target tool only supports interactive verification, then it is acceptable for your compiler to generate artifacts that are then manually fed into the tool and then expect the user perform manual verification.

[7]If you wish to justify for an exception to this, speak to Jackie early to confirm.

In both cases, it is important for you to justify, in the final user manual, why the source-to-target transformation your compiler performs is semantics-preserving.

4. $\boxed{\text{Document critical components of your compiler}}$, including:
   - Syntax of the source programming language
     - Simply copying and pasting the `.g4` ANTLR file is not acceptable.
     - You should present the syntax of your programming language in a way such that beginners of your tool (e.g., TA, Jackie) can easily try out new examples on their own.
   - Patterns of the source-to-target transformation and their (informal) justification
   - **10 working examples representative of your compiler's capability**
     - These examples should not be contrived; rather, they should correspond to some (simplified) real-world scenarios (e.g., bank, student management system, bridge control, mutual exclusion algorithm).
     - These 10 examples are meant to help TA/Jackie determine what the "best" it is that your compiler is capable of verifying.
   - Any known issues of your compiler (you would not want TA or Jackie to be surprised by your compiler crashing on new examples)

5 Requirements

- You must use the ANTLR4 Parser Generator (for Java) to build your compiler.
- For the target verification tool, you must choose from one of the following (and confirm by the due date; see Section 7), where a suggested starting point is provided for each tool:
  - PVS `https://pvs.csl.sri.com/`
    - ◇ This tool is available in Prism and used by *EECS4312 Software Engineering Requirements*.
    - ◇ More info here: `https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:pvs:start`
  - Coq `https://coq.inria.fr/`
  - Isabelle `https://isabelle.in.tum.de/`
  - Alloy `https://alloytools.org/`
  - PAT `https://pat.comp.nus.edu.sg/`
  - Spin `http://spinroot.com/spin/whatispin.html`
  - Z3 `https://rise4fun.com/z3/tutorial`

  Nonetheless, if there is a particular verification tool which you prefer to working with but it is not in the above list, speak to Jackie by the due date of submitting the `team.txt` file (See Section 7 for the due date).

6 Suggested Workflow of the Project

1. For the first week to 10 days, determine the target verification tool, and try out as many examples as possible to know the specification features that can be used to support the verification. This is why there is a milestone in early March to check if you are familiar enough with the verification tool.

   It is expected that members of your team conduct enough research on the tool chosen in order to understand how the tool works. Feel free to speak to Jackie (early) if you are stuck with certain specification concepts; getting in touch late in the timeline would not help.

2. Once you know about the specification features available in the chosen verification tools, go over the list of programming features that you wish to support: for each programming feature, can it be translated/formalized accordingly? If so, then include it. If not, then either drop it or speak to Jackie for advice. Then, design the syntax of your language accordingly.

That is, study examples of the chosen verification tool, and work **backwards** to see what programming features are feasible to be verified and thus may be supported by the source language.

3. You may **not** want to implement components of your compiler sequentially: design the full syntax of your input program, then implement the transformation from each programming feature to the specification feature, then test the output on the verification tool shortly before the project due date.

   Instead, get the verification done (including syntax, transformation, and testing) for one programming feature at a time. The advantage of this is that you always have a working compiler ready for submission. This is also why we have the milestone in late March to check for a working comiler on basic features.

7 Milestones of the Project

1. Confirm Team Member(s) and the Target Verification Tool

   By the end of **Tuesday, March 3**, submit a plain text file `team.txt` for your team via the Prism account of a memer:

   ```
   submit 4302 Project team.txt
   ```

   Only one submission is needed for each team. The file should contain names, student numbers, and Prism login names of members, as well as the verification tool that your team chooses to work with. For example, the `team.txt` file should look like:

   ```
   Last Name   First Name   Prism Login   Student Number
   =========   ==========   ===========   ==============
   Arthur      Fleck        afleck        123456789
   Bruce       Wayne        brucew        987654321

   choice of verification tool: Coq
   ```

2. Demonstrate Proficiency with the Chosen Target Verification Tool [**3%**]

   – On **Thursday March 5 or Friday March 6** (about 2 weeks after the project is released), your team is required to meet with Jackie and demonstrate that you are familiar with the verification tool (and its specification language) you choose.

   – During the meeting, you must demonstrate (using your computer) **5 non-trivial examples** (ones that show the various target language features that are relevant to your compilation) of verification. For each example, you will demonstrate how to verify it using the tool.

   – **In this meeting, Jackie may suggest specific tasks that your team should complete and will be included in the evaluation of the second milstone.**

3. Demonstrate Satisfactory Progress on the Compiler [**5%**]

   – On **Thursday March 19 or Friday March 20** (about 1 month after the project is released), your team is required to meet with Jackie and demonstrate a working version of your compiler on the following basic features of the source language features (of syntax of your own design), including:

   - variable declarations
   - variable assignments
   - variable references (i.e., referring to declared variables in expressions)
   - arithmetic, relational, and logical expressions
   - conditionals
   - specification (e.g., preconditions, postconditions, invariants, property assertions) in input programs that guide the target verification

  - During the meeting, you must demonstrate (using your computer) **5 non-trivial examples** (ones that show the above source language features) of verification. For each example, you will demonstrate how to verify it using your compiler (e.g., given an input file, your tool will compile it into another file which can be taken as input by the target verification tool).
  - **In this meeting, Jackie may suggest specific tasks that your team should complete and will be included in the evaluation of the final project in April.**

   These two milestones are meant to make sure that you are on the right track. Based on your demo, Jackie will give you feedback.

4. User Manual and Source Code of the Compiler [**22%**]
  - These are to be submitted by the project due date in April.
  - The user manual must provide sufficient details so that TA/Jackie can easily follow (e.g., installation of special software, how to build your compiler from scratch, how to run your tool from Eclipse and from a terminal) in order to run using your tool: **1)** examples which you provide; and **2)** **new** examples which TA/Jackie comes up with.
  - **Required format of the user manual and required organization of the source will be made available to you, no later than four weeks before the project due date.**

8 Grading Criteria of the Project

It is understandable that you have only six weeks to (attempt to) solve this open-ended problem. Therefore, what we expect from you is not a compiler that supports the verification of a rich source language. Instead, you will receive a **passing mark** if your compiler can at least support the verification of the basic features, listed in the above second milstone. To receive a higher grade, we then judge on:

- How similar is the source programming language to the target specification language (the more similar they are, the less challenge it is to transform from one to another)
- Additional language features that your compiler is able to verify, such as:
  - tabular expressions

    This is a recommended feature for you to explore and support in your compiler (specifically, the verification of completness and disjointness). As a starting point, see:

    Parnas, D.L., Madey, J.: *Functional documentation for computer systems.* Science of Computer Programming. 25, 4161 (1995)
  - iterations (with invariants and variants)
  - data structures (e.g., lists, arrays, maps)
  - function calls
  - anonymous functions
  - object orientations
  - client-supplier relations
  - generic class, generic methods
  - inheritance
  - concurrency
  - any other language features you find interesting for your compiler to support
- Justification on the correctness of your source-to-target transformation (i.e., the way in which you formalize the input programming constructs, using the output specification constructs, is semantics-preserving)
- Clarity and completeness of the user manual