EECS4302 Compilers and Interpreters
Winter 2020
Project Report Template

Revised: March 30, 2020

- Sections **4.4** *Document critical components of your compiler* and **7.4** *User Manual and Source Code of the Compiler* of the Project instructions also addresses the expectation of this user manual.
- This first page is the title page.
- Feel free to design a title page of this user manual of your compiler (e.g., with a informative title such as "*A Compiler for the Verification of a C-Like Programming Language: User Manual*")
- This template lists sections which your final report **must at least** cover. Feel free to add more sections if you think they help readers learn about your compiler.
- There is **no** need for your to explain the design of Java classes in the `model` package.
- The **minimum** number of working examples you must submit is **10**. However, you can supply as many more working examples as you see fit, if they help readers learn about the capability of your compiler.
- Keep the **Table of Contents** up to date when you submit this report.
- At least include each team member's:
  - Name (Last name, First name)
  - Student Number
  - Prism login name

Table of Contents

1 Input Programming Language

1.1 Overall Structure of a Program

This section is closely related to **Section 2**.

In this section, outline and explain how a typical input program should look like.

For example, there might be sections of an input program (e.g., variable declarations, functions, assertions):

```
module M1
  /* Section: Variable Declarations */
  x: INTEGER

  /* Section: Functions */
  increment (v: INTEGER)
    pre x + v < 10          /* Specification: Precondition */
    imp x := x + v
    post x = old x + v      /* Specification: Postcondition */

  /* Specification: Assertion */
  assert x < 8 /* This assertion, when loaded onto the verification, should fail. */
end
```

Figure 1: Structure of Input Program

Notes. Each of ten examples you submit may focus on a different perspective of your programming language. On the other hand, the above example program, should demonstrate as many **programming** features as possible that are supported by your compiler; **it can just be symbolic** (e.g., using x, y, f1, a1), but it must be syntactically and type correct.

1.2 How a Specification is Written

The notion of program correctness is *relative* to its specification, which can take the form of assertions, preconditions, postconditions, invariants, a combination of these, *etc*. For example, in Eiffel, in addition to the programming constructs (e.g., assignments, if-statements, from-until loops), contracts (i.e., require, ensure, invariant, variant) are supported so that you can verify the correctness of implementation against their contract at runtime.

In this section, explain clearly **all** kinds of specification that are supported in your input programming language. You may include at least (and this is *not* an exhaustive list):

1. Keywords supported for specification (e.g., assert)
2. Where can specification constructs occur in the input program? (See **Section 1.1**)
3. What's the form of each specification construct (e.g., proposition, predicate)?
4. Does your specification support the notion of pre-state vs. post-state (e.g., the use of old keyword in Eiffel)? If so, **how**? If no, **why not**?
5. Code snippets illustrating how complex each specification construct can be (e.g., compound propositions, nested universal/existential quantifications). In addition, in applicable, cross-reference to the working examples you submit.

1.3 List of Advanced Programming Features

You already implemented the basic programming features from Mileston-3. In this section, there is no need to address them again. Instead, focus on those advanced features which you have (partially or completely) implemented.

1.3.1 Feature 1: ??

For each advanced feature you support (for verification), **at least**:

1. Explain its syntax
2. Give examples (by **presenting code snippets**), where:
    a. Some input programs are **correct** (so that the generated code, when loaded and run on the verification tool, will indicate a **success**)
    b. Some input programs are **incorrect** (so that the generated code, when loaded and run on the verification tool, will trigger some **counterexample**).
    c. If some of these examples correspond to the working examples you submit, also make reference(s) to them.
3. Explain how it is **transformed** into the specification constructs for the verification tool:
    a. Justify why you believe the transformation is ***semantics-preserving***
    b. Your justification is more convincing when the working examples you submit show how this advanced feature can be verified, as well as how the verification may succeed (when some programs are correct) and fail (when other programs are incorrect).

1.3.2 Feature 2: ??

2 Output Specification Language

Corresponding to **Section 1.1**, outline and explain how a typical output program (which is both syntactically correct and type correct for the verification tool) generated by your compiler looks like. For example, show how ***Figure 1: Structure of Input Program*** (of course, in your case, the structure of the input language may look different) can be transformed into the semantics-equivalent structure written in the specification language of the verification tool.

Note. Each of ten example input program you submit may be compiled into an output program (to be loaded onto the verification tool) that looks different. On the other hand, your outline and explanation above, should demonstrate as many **specification** features as possible that are used from the target verification tool language; **it can just be symbolic** (e.g., using x, y, f1, a1), but it must be syntactically and type correct.

3 Examples

- You are required to submit (at least) **10 working examples** representative of your compiler's capability.
    - Some of these should demonstrate **how verification can succeed** (when the input program is ***correct***) and **how verification can fail** (when the input program is ***incorrect***, where a counterexample is generated by the verification tool).
    - 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 examples are meant to help TA/Jackie determine **what the "best" it is that your compiler is capable of verifying**.
- **This section is not meant to be long**. It should be similar to `doc-1.pdf` of your Milestone-3 submission. Summarize (in bullet points) what each of the submitted example demonstrates.

3.1 Example Input
3.2 Example input
3.3 Example input
3.4 Example input
3.5 Example input
3.6 Example input
3.7 Example input
3.8 Example input
3.9 Example input
3.10 Example input

If you decide to submit more than 10 examples, just create more sub-sections.

4 Miscellaneous Features

The main purpose of this project is program verification. Any additional features supported by your compiler such as type checking, error reporting, handling of multiple input files, *etc.* should be listed here. For each feature:

1. Describe how it works
2. Give examples (or refer to some of the examples you submit), screenshots, *etc.*

5 Limitations

- For each programming feature, do you support it fully? Or there is certain scenario that's not supported, e.g., simple loops rather than nested loops?
- List any other known limitations (e.g., certain input programs, although can be compiled to generate outputs, cannot be verified by the target tool).