COMP3161/9164 24T3 Assignment 0
Proofs

Marks : 15% of overall marks for the course.
A mark of x (out of 100) on this assignment will translate to .15x course marks.

Due date: **Tuesday, 7th of October 2024, 23:59:59 PM (Sydney Time)**

1 Task

In this assignment you will formally model a language of boolean computations using a variety of semantic techniques, including its syntax and semantics, and its compilation to various target languages.

Prepare your answers in one PDF file, preferably using LaTeX, where all prose is typeset. Figures may be drawn, but make sure they are legible. Please ensure all mathematics is formatted correctly. Some guidance will be posted on the course website.

Submit your PDF using the CSE give system, by typing the command

```
give cs3161 Proofs Proofs.pdf
```

or by using the CSE give web interface.

Part A: Syntax (25 marks)

Consider the language of boolean expressions $\mathcal{P}$ containing just literals (`True`, `False`), parentheses, if-and-only-if ($\leftrightarrow$) and negation ($\neg$):

$$\mathcal{P} = \{\texttt{True}, \texttt{False}, \neg\texttt{True}, \neg\texttt{False}, \texttt{True} \leftrightarrow \texttt{False}, \neg(\texttt{True} \leftrightarrow \texttt{False}), \dots\}$$

1. Write down a set of inference rules that define the set $\mathcal{P}$. The rules may be ambiguous. *(5 marks)*

2. The operator $\neg$ has the highest precedence, and if-and-only-if is right-associative. Define a set of simultaneous judgements to define the language without any ambiguity. *(5 marks)*

3. Here is an *abstract syntax* $\mathcal{B}$ for the same language:

$$\mathcal{B} ::= \mathsf{Not}\ \mathcal{B} \mid \mathsf{Iff}\ \mathcal{B}\ \mathcal{B} \mid \mathsf{True} \mid \mathsf{False}$$

Write an inductive definition for the *parsing* relation connecting your unambiguous judgements to this abstract syntax. *(5 marks)*

4. Here is a big-step semantics for the language $\mathcal{B}$

$$\frac{x \Downarrow \mathsf{True}}{\mathsf{Not}\ x \Downarrow \mathsf{False}}(\mathrm{N}_1) \quad \frac{x \Downarrow \mathsf{False}}{\mathsf{Not}\ x \Downarrow \mathsf{True}}(\mathrm{N}_2) \quad \frac{}{\mathsf{True} \Downarrow \mathsf{True}}(\mathrm{N}_3) \quad \frac{}{\mathsf{False} \Downarrow \mathsf{False}}(\mathrm{N}_4)$$

$$\frac{x \Downarrow \mathsf{False} \quad \mathsf{Not}\ y \Downarrow v}{\mathsf{Iff}\ x\ y \Downarrow v}(\mathrm{N}_5) \quad \frac{x \Downarrow \mathsf{True} \quad y \Downarrow v}{\mathsf{Iff}\ x\ y \Downarrow v}(\mathrm{N}_6)$$

a) Show the evaluation of $\mathsf{Iff}\ (\mathsf{Not}\ (\mathsf{Iff}\ \mathsf{True}\ \mathsf{False}))\ \mathsf{False}$ with a derivation tree. *(5 marks)*

b) Consider the following inference rule:

$$\frac{x \Downarrow v}{\mathsf{Not}\ x \Downarrow v^{-1}}$$

where we understand v^{-1} to be defined by the following equations:

$$\mathsf{True}^{-1} = \mathsf{False}$$
$$\mathsf{False}^{-1} = \mathsf{True}$$

Is this rule derivable? Is it admissible? Justify your answers. *(5 marks)*

Part B: Semantics (20 marks)

Here is a small-step semantics for a language $\mathcal{L}$ with `True`, `False` and `if` expressions:

$$\frac{c \mapsto c'}{(\texttt{If}\ c\ t\ e) \mapsto (\texttt{If}\ c'\ t\ e)}(\mathrm{L}_1) \quad \frac{}{(\texttt{If True}\ t\ e) \mapsto t}(\mathrm{L}_2) \quad \frac{}{(\texttt{If False}\ t\ e) \mapsto e}(\mathrm{L}_3)$$

1. Show the full evaluation of the term (`If True` (`If True True False`) `False`). *(5 marks)*

2. Define an equivalent *big-step* semantics for $\mathcal{L}$. *(5 marks)*

3. Prove that if $e \Downarrow v$ then $e \overset{\star}{\mapsto} v$, where $\Downarrow$ is the big-step semantics you defined in the previous question, and $\overset{\star}{\mapsto}$ is the reflexive and transitive closure of $\mapsto$. Use rule induction on $e \Downarrow v$. *(10 marks)*

Part C: Compilation (15 marks)

1. Define a recursive *compilation function* $c : \mathcal{B} \to \mathcal{L}$ which converts expressions in $\mathcal{B}$ to expressions in $\mathcal{L}$. *(5 marks)*

   It might help to consider writing a Haskell function from $\mathcal{B}$ (its abstract syntax encoded as a Haskell datatype) to $\mathcal{L}$ (likewise encoded). However you may use any format you like to define your function as long as its meaning is clear, you do not need to use Haskell syntax.

   We do expect your compilation function to be able to produce `If` expressions in its output. An alternative would be to simplify the input expression in $\mathcal{B}$ down to True or False (it is currently possible to do this for every expression in $\mathcal{B}$) and then trivially compile to $\mathcal{L}$. Don't do that.

2. Prove that for all e, $\mathsf{e} \Downarrow \nu$ implies $c(\mathsf{e}) \Downarrow \nu$, by rule induction on the assumption that $\mathsf{e} \Downarrow \nu$. *(10 marks)*

Part D: Lambda Calculus (25 marks)

1. Here is a term in λ-calculus:

   $$(\lambda \mathsf{g}.\ \lambda \mathsf{f}.\ \lambda \mathsf{x}.\ (\mathsf{g}\ \mathsf{f}\ (\mathsf{f}\ \mathsf{x})))\ (\lambda \mathsf{f}.\ \lambda \mathsf{x}.\ \mathsf{f}\ \mathsf{f}\ \mathsf{x})$$

   a) Fully β-reduce the above λ-term. Show all intermediate beta reduction steps. *(5 marks)*

   b) Identify an η-reducible expression in the above (unreduced) term. *(5 marks)*

2. Recall that in λ-calculus, booleans can be encoded as binary functions that return one of their arguments:

   $$\mathbf{T} \equiv (\lambda \mathsf{x}.\ \lambda \mathsf{y}.\ \mathsf{x})$$

   $$\mathbf{F} \equiv (\lambda \mathsf{x}.\ \lambda \mathsf{y}.\ \mathsf{y})$$

   Either via $\mathcal{L}$ or directly, define a function $\mathsf{d} : \mathcal{B} \to \lambda$ which converts expressions in $\mathcal{B}$ to λ-calculus. *(5 marks)*

3. Prove that for all e such that $\mathsf{e} \Downarrow \nu$ it holds that $d(\mathsf{e}) \equiv_{\alpha\beta\eta} \nu'$, where ν' is the λ-calculus encoding of ν. *(10 marks)*

Part E: Let-Bound Local Functions (15 marks)

Suppose we added *unary local function definitions* to our language $\mathcal{P}$. Here's an example in concrete syntax:

```
let
  g(x) = ¬x
in
  g(True)
end
```

We limit ourselves to non-recursive bindings (meaning functions can't call themselves), and first-order functions (meaning functions require boolean arguments).

1. Extend the abstract syntax for $\mathcal{B}$ from question A.3 so that it supports the features used in the above example. Use first-order abstract syntax with explicit strings. You don't have to extend the parsing relation. *(5 marks)*

2. Define a scope-checking judgement, similar to the **ok** judgement from the lectures. It should check (a) that all names of variables and functions are used only within their scopes; and (b) that names used in variable (or function) position are indeed the names of variables (or functions). Hence, the following expressions should all be rejected:

```
let                 let                 let
  f(x) = ¬x           f(x) = ¬x           f(x) = x(True)
in                  in                  in
  f(x)                f(f)                f(False)
end                 end                 end
```

   The following are examples of things that should be accepted: nested definitions, and shadowed definitions.

```
let                          let
  f(x) =                       f(x) = x
    let                      in
      g(y) = ¬x ↔ y            let
    in                           f(x) = f(x)
      g(x) ↔ ¬g(x)           in
    end                        f(True)
in                           end
  f(False)                 end
end
```

   Note that the latter example is *not* a recursive call.

   This question is a bit more open-ended than the previous ones, and will require you to make some design choices. Here's one: if the same string is used to denote both a function name and a variable in the same scope, should that be rejected or accepted by your judgement? Explicitly state (in English) which choice you made, and define things accordingly. *(10 marks)*

2 Late Penalty

You may submit up to five days (120 hours) late. Each day of lateness corresponds to a 5% reduction of your total mark. For example, if your assignment is worth 88% and you submit it two days late, you get 78%. If you submit it more than five days late, you get 0%.

Course staff cannot grant assignment extensions—if you need an extension, you have to apply for special consideration through the standard procedure. More information here: `https://www.student.unsw.edu.au/special-consideration`

3 Plagiarism and AI

Many students do not appear to understand what is regarded as plagiarism. This is no defense. Before submitting any work you should read and understand the UNSW plagiarism policy `https://student.unsw.edu.au/plagiarism`.

All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. In this course submission of any work derived from another person, or solely or jointly written by and or with someone else, without clear and explicit acknowledgement, will be severely punished and may result in automatic failure for the course and a mark of zero for the course. Note this includes including unreferenced work from books, the internet, etc.

Do not provide or show your assessable work to any other person. Allowing another student to copy from you will, at the very least, result in zero for that assessment. If you knowingly provide or show your assessment work to another person for any reason, and work derived from it is subsequently submitted, you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep your work private until submissions have closed.

If you are unsure about whether certain activities would constitute plagiarism ask us before engaging in them!

You are *not* allowed to use AI tools (such as ChatGPT or GitHub Copilot) to help you with technical content, or to develop definitions and proofs. You *are* allowed to use AI tools for non-technical purposes; for example, to help polish your grammar in essay questions.