Feedback for part 1 of G53CMP (Compilers) coursework
Friday 10th November 2017
Martin Handley, psxmah@nottingham.ac.uk

General comments

- Overall, reports were largely well written and adequately detailed. A number of students in particularly produced reports that were of a standard that would reflect a good dissertation format. Despite this, it was clear that on many occasions students failed to read the questions thoroughly, which repeatedly stated that additions to the lexical, context-free and abstract grammars should be specified precisely. Unfortunately, this led to deductions in correctness marks.
- For those students who submitted dark code screen-shots (i.e., light text on dark background), I would advise against this in the future (e.g., for dissertations). If a screen-shot has to be used, then the code should be dark and the background light in order to match the white paper. This makes the code much easier to read when printed, and as such, improves the quality of the report. In addition, some code screen-shots were too small and/or out of focus. This should also be avoided. My general advice would to invest some time in learning how to typeset code and grammars in Latex. This presentation is not only clearer, but appeals more to the reader.

Question specific feedback

Part 1.1

- This was by far the most well answered question and almost all students scored full marks for both correctness and style.
- A number of students failed to specify the additions to the grammars (lexical, context-free and abstract), but I didn't penalise (as instructed by Henrik).
- Some student's grammars also appeared to confuse the notion of non-terminals and terminals, with their production rules using italic font for the terminals **repeat** and **until**. I didn't penalise this or any other instances of this throughout the coursework, however. Nevertheless, students may wish to revisit these notions for future assessment.

Part 1.2

- Overall, this question was well answered, with the majority of students scoring full marks for correctness, and all students scoring full marks for style.
- A significant number of students lost correctness marks for failing to specify their additions to the grammars, as with part 1.2. In some cases, students simply specified one set of production rules that had a 'hybrid' style syntax somewhere between context-free and abstract. I felt that this demonstrated a lack of appreciation for the distinction between context-free and abstract grammars, and consequently marks were deducted.
- Correctness marks were also deducted for failing to consider the precedence of the conditional expression in the `Parser.y` file, i.e., omitting `%right '?' ':'`.
- As a general style comment, very few students took initiative with their naming conventions in the abstract syntax for this expression. A common definition was something akin to:

```
ExpCond
 { ecCond    :: Expression
 , ecExp1    :: Expression
 , ecExp2    :: Expression
 , expSrcPos :: SrcPos
 }
```

Although, strictly speaking, this is a valid definition, in my opinion it would be more

constructive for the names `ecExp1` and `ecExp2` to be reflective of the expression's semantics, i.e., named `ecTrue`, `ecFalse` or something similar. Style marks were not deducted for this, however.

Part 1.3

- Overall, students scored well on this part, and many students were awarded full marks for both correctness and style.
- As with part 1.1 and part 1.2, a number of students still failed to specify their additions to the grammars, and again, some specified a 'hybrid' style set of production rules. Marks were deducted in both instances.
- The majority of marks lost were for three other reasons in particular:
  -- Too many production rules (four in most cases) added to the `Parser.y` file for the `if` command, leading to Happy reduce/reduce conflicts.
  -- Abusing the notion of a command by defining an `ElsIf` constructor for the corresponding abstract syntax type (i.e., `Command`). This was to allow for a recursive abstract definition for the if command, whereby each **elsif** branch would be the 'parent' node of subsequent branches and the optional **else** branch. In this instance, the corresponding pretty printing functions were unnecessarily complex. Nevertheless, in most cases this approach did function correctly, which meant that only style marks were deducted.
  -- Failing to appreciate the non-deterministic nature of lists and defining the abstract syntax for **elsif** branches as something of type `Maybe [(Expression, Command)]` cf. `[(Expression, Command)]`. Again, this was more of a style issue rather than an issue of correctness.
- As a general style comment, for the students who defined `if`'s abstract syntax as such:

```
CmdIf
 { ciCond    :: Expression
 , ciThen    :: Command
 , ciElsIfs  :: [(Expression, Command)]
 , ciOptElse :: Maybe Command
 , cmdSrcPos :: SrcPos
 }
```

I think it would have been good for them to group `(ciCond, ciThen)` with `ciElsIfs`:

```
CmdIf
 { ciCondsCmds :: [(Expression, Command)]
 , ciOptElse   :: Maybe Command
 , cmdSrcPos   :: SrcPos
 }
```

This allows for a more natural encoding of this command's conditionally branching semantics, which can then be uniformly handled during e.g., type checking and compilation. Marks were not deducted for the above.

Part 1.4

- This was the lowest scoring task overall, with only a small number of students scoring full marks for correctness, and fewer for style.
- Again, a significant number of students failed to specify their additions to the grammars.
- Aside that, most correctness marks were lost for the following reasons:
  -- Failing to check graphic characters were valid, i.e., between `' '` and `'~'` (ASCII 32-126) and not `'\''` and not `'\\'`.
  -- Failing to check escape characters were valid. In this case, a number of

  students' solutions would convert an invalid escape character into a potentially valid graphic character, e.g., `'\\b'` -> `'b'`.
  -- Failing to handle the case where `'\''` appears on its own by omitting a pattern for the remainder of the input string being empty in the corresponding `scanLitChar` function. This would lead to a Haskell runtime error.
- Most style points were lost for having a catch-all pattern in the corresponding `scanLitChar` function, which would disregard the initial `'\''` without any thought of what appears next in the input. In this case, it is better to provide precise error messages. For example, “`\'a"` could be consumed by the lexer and an error message stating that `'\''` was not terminated could be returned.