AP - Assignment 3

Nicolas Gram Dyhrman — lzg210
Mikkel Bilgrav Madsen — tdg783

September 30, 2022

Design and implementation choices

We have chosen to use the parsec library since it was mentioned being a good choice as it will be present during our exam. This choice did eventually lead to us having issues with precedens, as the other READ parser library had some nice functions for determining precedens whereas in the parsec library we had an issue finding /implementing said functions.

Our eqaulity operators $<, >, ==$ parses as expected, but the related $a <= b$ parses by saying that $a <= b = a < b + 1$ and $a >= b = a + 1 > b$ thereby using only the two operators $<, >$ interally to handle the cases. Again this is tested in our testsuite.

Assessment

Our parser is overall rather functional, we can take multiple statements, and parse most expressions in an (almost) correct form. Our major problem is with regards to precedens. We had a hard time manually fixing the precedense of parantheses when other precedense ruels were applied (f.x. for times and div).

Completeness

The precedens rules for having paranthesis combined with higher precedens operators (e.g. Times and Div) does not work as intended. The issue is showcased by f.x. $parseString$"$(1+2)*3$" giving an answer equal to $1+2*3$, meaning that for the higher order precedens operators they somehow overwrite our paranthesis. Given more time this problem should be fixable.
Another issue is regarding escape characters in strings, we have issues with certain escape characters in strings not being passed as wanted. Another issue is variables and keywords, we having f.x. a variable called Nonee escapes with an error, where it should be a variable called "Nonee" this has to do with how we implemented keywords and probably needs more cases for having letters after a

keyword. Finally we had issues with the maximum of three equality operators in an expressions (e.g. 2==2==2) which should fail, but where we simply parse the first two and ignore the rest. We did not find a way to run through the rest of the expression to search for another of these operators. The same issue is relevant in regards to the other equality operators (¡, ¿, ¿=, ¡=). Now this could be fixed by some way of checking if one has appears, does another one appear in the remaining expression, but we did not have time to find a solution.

Otherwise we do parse all the operators, we pass strings, list, list comprehensions, variables, constants and it is overall close to being complete in regards to the specifications.

Correctness

Our test suite is run by calling $\$stacktest$ while being in the src directory of part2.
All of our mathematical operators $+, -, *, //, \%$ are parsing correctly as we intend, and as is described in the assignment text. Which is thoroughly tested in our testsuite.
Our test suite should be rather fulfilling, we have tested for both when it fails with error messages, and all possible succeeding calls. The tests we fail are described in completeness as being lacking, and due to extensive testing we have located most (hopefully) of our faulty implementations in regards to the specification.

Efficiency

We have focused our efforts on making the implementation work, and smaller optimizations has been left out due to lack of time.

Robustness

As the main question of robustness in this case is internal, namely a parser sending to an interpreter, it is somehow to a high degree the interpreters responsibility to errorcheck most inputs. In regards to parsing things that should not parsed, we have issues. At time we will parse string that maybe should have been discarded and rejected. This is due to our parser is overly eager to pass things that might not be correct.

Maintainability

Our code is fully commented which should increase the readability and maintainability of our code to some degree.

Our code is rather messy, and even though we tried cleaning it up with auxiliary functions to increase maintainability and readability through modularity, the code is still spaghetti-like.