CS325 - Compiler Design

1403116

February 2017

1 Design and Implementation

This implementation makes use of the shlex library to separate out tokens based on the bash equivalent lexical method, which matches up quite well to the lua tokenisation routine. In order to return useful error information with line number, the data was split by line and then passed into a shlex constructor. The recursive technique used was predictive parsing, with requests to both view and pop the leftmost token (in terms of context free grammars), which can be used in constructions to test expected matching values.

By using predictive parsing, continuing through the program becomes much simpler, since an unmatched token can immediately be treated as an error. A method to return the latest token was implemented for when an error occurs, as it was intended to allow the possibility that the failed token was meant to begin a new construction. However since the construction needed may not have been necessarily known, this was later removed and as such the error recovery may not be as elegant as it could be.

Error handling is done on a per token basis and entirely within focus of a construction. This allows reported errors to be precise to an individual problem, specifying the expected token and the encountered token. Token comparisons make use of both regular expressions and the simpler equivalence expression. Another approach to the parsing could have been to make use of collections of regular expressions, however some functionality could have been lost between lines, as well as readability problems occurring. Regular expressions do have the benefit of excellent forward checking[1], but consuming of tokens may become a little more difficult.

2 Challenges

Long strings are one of the few non-bash recognised tokens required to be parsed in the lua language[3]. Instead of developing a separate lexical tokenizer, instead the assumption was made that there are no multiple line spanning strings. By assuming this, long strings can be replaced by stepping through line by line and using regular expressions to find occurrences, although existing quotation marks have to be escaped in this case. After this the lexical tokenizing can take place as normal. To improve the system, this initial parse through could span multiple lines and comma separate the strings between lines, to allow for line counts to be maintained. It is also possible to simply replace the entire long string with an empty string for the purposes of the syntax analysis since it is not actually required to compile.

One limit of the system currently in place is the lexical parsers inability to recognise some binary operators such as '<=', '>=' or '=='. This is because shlex treats each of the component characters as individual tokens[2]. The current workaround solution is to forward check to the next token

when one of these is met to try and create a binary operator and return this if it matches. One problem with this method is that the tokenizer strips whitespace characters, so the two tokens may not necessarily be next to each other, making this erroneous acceptance. This could be fixed by making a new lexical analyzer which accepts these operators as expressions and can predict when to accept singular or paired values.

Some constructions of the lua grammar are heavily contained within others and as such, several of the grammars simulated in the program are contained within another. This was a problem within the prefixexp construction which was heavily self-recursive. In order to keep track of which production was created, Python's ability to return different types was used with calls to prefixexp in order to handle some of the harder parsing challenges. Despite this some of the possible prefixexp constructions aren't possible to discern without further forward checking, so a desired improvement would be to look further down the parse list[4], although this would require a lot more checks.

3 Testing

Development of this project was incremental, allowing each construction to be tested as it was produced. As the complexity increased however, there were many cases where the incorrect production was taken, so edits had to be made at several points. Initially the dataset was passed through each method, before a decision was made to have singular methods to request tokens, which helped with reaching across lines to get the next token.

One remaining possible problem is what the correct response to a given error actually is. Often an error simply fails out of a construction back to the current block, however it might sometimes be best to try and build a new statement on the token that is failed on. The selection for this is very complex however and would require a lot of consideration into what was intended to occur up until the point of the error. As such, one error may lead to several more before the program recovers, as it expects new statements to begin. Improvements to the system could involve repeating the program with certain erroneous constructions removed.

4 Conclusion

Overall, the program is quite good at handling relatively simple programs although it struggles when a depth of recursive constructions appear. An element which would be quite easy to implement is function checking, which are saved as the program builds and these can be checked on funccall constructions. The next step for this program would be constructing a new lexical analyser and incorporating a more robust forward checking system.

References

[1] Python regular expressions. `https://docs.python.org/2/library/re.html`.

[2] shlex - simple lexical analysis. `https://docs.python.org/2/library/shlex.html`.

[3] Lua.org. Lua 5.1 manual. `http://www.lua.org/manual/5.1/manual.html`.

[4] Gihan Mudalige. Predictive parsing and follow sets. `http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs325/cs325-3-syntaxanalysis.pdf`.