Compiler Design - CS325 - Report

Introduction

This coursework involved creating a recursive descent parser to parse a Lua 5.1 program and be able to identify and recover from errors found. This solution utilises a changed grammar that most importantly has no left recursion and a top down predictive recursive descent parser that may lookahead by up to 2 - however in most cases a lookahead of only 1 is required.

Design decisions and difficulties

Initially attempts were made to change the grammar into one that satisfies the LL1 property [2]. Whilst removing direct and indirect left recursion was possible, fully left factoring the grammar to remove ambiguity was unsuccessful. The property of disjoint augmented first sets was therefore not satisfiable using this grammar and thus a lookahead of 2 is required in cases where there is ambiguity.

The design choice was taken early on that a predictive rather than a backtracking parser would be created. Contributing factors to this included predictive parsing tending to be faster than backtracking potentially a large number of steps in the grammar "tree", easier error checking and a predictive parsing not being seen as significantly more complex for this project.

Very few external resources were used in the creation of this program. This was due to a philosophy of building the program from the ground up (rather than top down) by solving simple problems and adding complexity and refactoring. Thus the program began with simply matching terminals, then being able to match a nonterminal that contained those terminals, then the ability to match zero or more nonterminals, and finally nonterminals being able to match other nonterminals.

The implementation frequently utilises partially applied or curried functions, most notably in the implementation of the "star" operator. The corresponding function called "star" that implements this looks ahead by n (where n is 1 or 2 in this program) and if it can find n matches then executes a list of other functions passed to it as parameters that are other nonterminals.

In addition to the "star" operator the "?" or optional operator is also implemented similarly except without a loop. Implicit use is made of the system stack as the program runs and functions call into other functions in a recursive manner. Mirroring the grammar layout is achieved through this.

Program structure like this ensures that once a function that corresponds to a grammar rule has been entered, a matching path can always be found by checking the current token and possibly the next, given a syntactically correct program.

Syntax errors in a program being parsed will therefore either result in an inability to match the current token to a viable path while inside a grammar function. There is another case. Consider a grammar nonterminal "statement" of form 'do block end' where a block is a nonterminal that is not a string and do and end are terminals, and an erroneous example of this of the form 'do "stringy" end'. Program flow will enter the statement function and match the 'do' keyword as a terminal. It will then be unable to find a match for the string, as it was expecting a block. This case is different from the previously mentioned one as the lookahead will succeed due to it only considering the 'do' keyword and looking ahead only 1. However another case of failure is then needed to identify that the string should not be there.

In this second case, the matching function will throw an error, as in all cases in a syntactically correct program, the match succeeds.

The tokenizer creates a list of tokens by using regular expressions to create lexemes and attach meaning by attaching a type such as “Number” or “Keyword”. The program is then able to perform matches on keywords or variable names for instance by checking the type in the tuple and calling the appropriate match function (to allow for a variable name to be matched base on if it has a type “Name” but an “=” operator only be matched if it is exactly an “=” operator).

As the program runs, an index (i) into the token list is used to keep track of where the program is. Consuming a token is equivalent to incrementing i.

Matching named and anonymous functions and their parameters is done by logging the indices which contain them (from i to i+some_amount) and their parameter lists at the appropriate places in the grammar where they may legally be created as the program runs.

Testing

A note to testing the program on files - whilst this program does support Lua’s long string of the form [==[bla]==] and single line comments, it does not support the ‘require’ keyword that is not in Lua 5.1 nor does it support multi line comments. Therefore these were removed if present from test cases.

Testing was done during program development, with thorough testing being done on cases early in development to ensure that building blocks for the program were sound. Functions were tested as they were added, as well program behaviour re-tested to ensure it still behaved correctly.

Additionally, it was found a good source of test cases was simply to run the program on large Lua files found on the internet, isolate the case that was breaking the program and fix it.

The program is able to parse large files such as ones found in the addons directory at [1].

Limitations

Whilst this program conforms to the grammar, since the grammar is not LL1 a lookahead of 2 is infrequently required. Ideally, a lookahead of only 1 would ever be needed. This is only a very minor drawback in terms of performance however.

The largest limitation of the program is the ability to recover from multiple errors.

References

[1] Ui source code for world of warcraft. `https://github.com/tekkub/wow-ui-source`. Accessed: 10.02.17.

[2] Marc Moreno Maza. Ll(1) grammars. `http://www.csd.uwo.ca/~moreno/CS447/Lectures/Syntax.html/node14.html`. Accessed: 11.02.17.