Cy Scott (U12538058)

LL(k) Parsers: Parsing strings one K of tokens at a time

# Description of the Task

The purpose of this project was to create an parser. LL stands for left to right and leftmost derivation. parsers are designed to parse an input string using a given context free grammar (CFG). The way to make parsers possible is to use a parsing table where the columns represent the permutations of all the terminals symbols up to a length of and where the rows represent the permutations of non-terminals or variables up to a length of . The cells in the table represent the correct rule to follow given some string of terminals and some variable.

# Constructing a Parsing Table

Constructing a parsing table is the most complex part of this problem; it requires recursively exploring each rule to generate derivation trees for each variable. Rules can often branch into other rules and sometimes reference the same rule that is the root of the tree, therefore it is possible to create a tree of infinite size, it is also possible to create a tree that is so large that it is impossible for a given computer to store or generate the tree. Both of these kinds of trees will create a stack overflow error in the program when attempting to generate the derivation tree.

Due to the nature of a stack overflow error most programming languages do not allow the program to gracefully handle those kinds of errors. As such, it is important to keep track of how many recursive calls are made when creating the derivation tree, and to throw an appropriate error when such an error happens. Although some CFGs can be restructured so that the amount of recursive calls is reduced, not all CFGs can be adjusted in that way.

After the derivation trees for each variable are constructed, then a parsing table can be created based on those trees. Since there is one and only one tree for each variable, each row in the table will represent that variable. Using the derivation tree for each variable, every possible path of the tree must be explored in order to produce a string of terminals up to a length of and a rule that produces the string of terminals. The string of terminals will be used to place the rule in the appropriate column for the row.

# Parsing Algorithm

After an parsing table is constructed, then the following algorithm is used to parse the input:

1. A stack () is created and the starting variable for the CFG is push onto the stack.
2. If the input stream is empty then go to step 8.
3. A string of symbols is peeked from the input stream up to a length of and stored in a variable called (peeking means that the data is read from the stream, but is not removed from the stream).
4. A symbol is read from the and stored in a variable called .
5. If is a variable then:
   1. Find the cell in the parsing table using as the row index and as the column index.
   2. If no rule exists then return an error else push the rule on to the.
   3. Go to step 2.
6. If is a terminal then:
   1. If matches the first value in then remove one value from the input stream and go to step 2 else return an error.
7. If the is empty then return all the rules that were used to parse the input and return them in the order in which they were used. However, if is not empty then return an error.

It should be noted that the special case of an empty input string will not work with this algorithm. In that special case you will need to check the starting variable and make sure that it supports an empty string and return the appropriate response.

# Summary of the Results

A few examples were tested against this program to make sure that it would function as expected. For example, the following CFG was tested:

The test setup was and the input was . The following results were reproduced:

A total of 7 rules processed.

Rules Followed:

S, S, S, S, S, S, S

Leftmost Derivations:

S -> (S+S) -> ((S+S)+S) -> ((a+S)+S) -> ((a+a)+S) -> ((a+a)+(S+S)) -> ((a+a)+(a+S)) -> ((a+a)+(a+a))

Another example:

The test setup was and the input was . The following results were reproduced:

A total of 4 rules processed.

Rules Followed:

S, A, A, B

Leftmost Derivations:

S -> AcB -> aAbcB -> aabbcB -> aabbcacb

# Surprising Discoveries

Converting this algorithm into a program did present some challenges. For example: solving the stack overflow problem when creating derivation trees. The stack overflow problem was solved by ending the building process for a branch once a string of terminals of size would be produced following the path of the branch. Furthermore, a tree level index value that correlated to the minimum length between the current node and the root node was used to make sure that the tree did not grow taller than a fixed sized. If the tree did grow to the fixed size and the longest branches of the tree could not produce a string of terminals of length then an error would be returned. This was important because if this did happen then it was not possible for the computer to parse input for the provided CFG.

Another discovery was figuring out that the algorithm could be changed in a way such that the time complexity analysis would go from: to: . This is because the data structure for the parsing table could be changed to a hash table. The parsing table has a search time of: while a hashing table has a search time of: . Because the input string is read in increments of then the analysis is when a hash table is used. As such, large values of can produce a faster processing time when using a hash table versus using a parsing table.

# Conclusions

parsers are a simple way of parsing most CFGs and the complexity of the CFG can be high enough to describe some high level programming languages and math equations. In addition, parsers work the same way a human reads a string (left to right following rules as the string is read). As a result of these rules, parsers are very intuitive.