Theory of Computation (CS - 561)
Project Report: On CFG and PDA

Sandesh Bhusal

December 10, 2023

Contents

1 Introduction

Context-free grammars (CFGs) provide a formal framework for describing the syntax of programming languages and other formal languages. They consist of production rules that define the hierarchical structure of strings using non-terminal and terminal symbols. Pushdown automata (PDAs) complement CFGs by serving as theoretical machines equipped with a stack that can process and recognize languages generated by context-free grammars. Together, CFGs and PDAs are intimately connected through the concept of language equivalence, where a language is generated by a CFG if and only if it is recognized by a PDA, highlighting their symbiotic relationship in the study of formal languages and computational theory.

In this project, **CFGDeriver**, we explore the methods to parse a given language grammar and check to see if a given string can be parsed using the given grammar or not.

2 Implementation

2.1 Tools used

2.1.1 Programming Language

The project was implemented in Rust. I chose rust for its ease of use, powerful pattern-matching abilities (not on string, rather on code and objects), and a simplified way of doing functional programming (with currying, which reduces the codebase size and makes it more legible. There are multiple files for each structure in the codebase, primarily for printing purposes. Rust also has very ergonomic logging crate, env_logger, which can be used during debug print. Apart from this, the 'clap' crate was used to parse arguments, and 'anyhow' crate for error handling.

`Bash` and `Python3` scripts were used for benchmarking and counting the number of variable derivations in cases where an output was produced. The scripts `runall.sh` are used on testcases, `bench.py` is used to get the output across benchmark suites. The binary produces logs on every stack manipulation and variable expansion, the expansion logs were `grep`ped to find the number of derivations. The benchmark script takes in `n`, as a first parameter, where `n` means the id of the eval suite to execute.

2.1.2 Validation

The provided codebase contained three test and benchmark suites. The tests each contained 5 strings each for validation, and the eval suites contained 3 strings each for benchmarking. The results of benchmark were validated using Stanford's CFG Developer which uses a Early parser. To validate results for the string membership in the language for G1 through G3 for eval suites, the CNF

forms were used and checked online if the answer found through **CFGDeriver** was correct or not, since the normal bound of 100 and stp_b cannot be used for it. The results obtained from the tool and CFGDeriver were tallied and found to be correct in the output. However, for the first grammar, `CFGDeriver` went into an infinite loop, and testing with a BFS-based implementation produced similar results. This might be due to the indirectly-derivable left-production in the grammar on `S->ASA` and `A->S` rules present in the grammar.

Bounds calculation

Each instance of program execution produces a result - "yes" or "no". However, within the given bounds, the given PDA may or may not be able to generate the given string. There are three types of bounds in use for the project:

1. $stp_c = 100$: This bound gives each instance of the PDA, a maximum possible derivations of 100 variables. If the PDA can produce results within these 100 derivations, the result is returned as "yes", otherwise as "no".

2. $stp_b = \frac{b^h - 1}{b - 1}$: This bound was specially interesting as we are trying to check if we can derive the given string within the minimum possible number of derivations. For a grammar that has a maximum number of derivable symbols to the RHS of the current variable as b, there are stp_b steps to derive the given string in the best case for a string of length n, where $n \leq b^h$. Therefore, n was substituted in the above formula, to get $stp_b = \frac{n-1}{b-1}$ and to do the simulation.

   The simulation using the stp_b bound produced almost all "no"s (except one timeout) as answer because the bounds themselves were very small. For a given string of length, say, $n = 45$ and branching factor of, say $b = 3$, we get the bounds number as $stp_b = 22$ derivations. This is not surprising,

2.2 Process

The project was completed in the following stages:

1. Create a parser type for the CFG string.
2. Create a PDA type from the parser type above.
3. Test on provided testsuite.
4. Create a simplified PDA type without states using only stack duplication for generation of multiple instances.
5. Test on provided testsuite.
6. Run benchmarks on provided eval suite.
7. Validate benchmark results.

3 Benchmarks and Analysis

Results from CFG Developer

Although this is not the target of this project, i.e. to produce efficient parsers that are more efficient than a normal PDA implementation, the CFG Developer tool was used as a reference point to generate the results for the PDA checks on the eval suite for membership of strings. This was very helpful specially with the second eval suite, because the third eval suite was a common "not pallindrome" grammar, the membership for which could easily be analysed by hand. The following observations were made from the results of CFGDeveloper[1]:

Table 1: Results from CFG Developer for different languages - values denote number of steps taken by the Early parsing algorithm to reach the conclusion. Multiple values means ambiguous derivation.

Language in Consideration	**Eval 1**	**Eval 2**	**Eval 3**
L1Gb	Yes (119/124)	Yes (199/204)	Yes (240/240)
L1cnf	Yes (89/89)	Yes (129/129)	Yes (169/169)
L2Gb	Yes (81)	Yes (123)	No
L2cnf	Yes (89)	Yes (129)	No
L3Gb	No	Yes (34)	Yes (85)
L3cnf	No	Yes (127)	Yes (167)

3.1 Observed results

G^{type}	$eval\_1$		$eval\_2$		$eval\_3$	
	$S \Rightarrow *w_1$	#deriv	$S \Rightarrow *w_2$	#deriv	S $\Rightarrow *w_3$	#deriv
G_b^c	Timeout	N/A	Timeout	N/A	Timeout	N/A
G_b^b	No	230702	No	1611043	Timeout	N/A
G_{cnf}^{cnf}	Timeout	N/A	Timeout	N/A	Timeout	N/A

Table 2: Results for eval #1

3.2 Analysis

3.2.1 Results from CFGDeriver

The results from CFGDeriver matched all cases on the results of CFG Developer, for the CNF form of grammar which assures some level of accuracy. Also

[1]Even though the parser from CFGDeveloper derived all strings pretty quickly, it took around 4-10 seconds each for deriving the strings 'eval1_2' and 'eval1_3' which was an interesting fact to note.

G^{type}	$eval\_1$		$eval\_2$		$eval\_3$	
	$S \Rightarrow *w_1$	#deriv	$S \Rightarrow *w_2$	#deriv	S $\Rightarrow *w_3$	#deriv
G_b^c	Yes	85	No	102	No	229
G_b^b	No	38	No	24	No	93
G_{cnf}^{cnf}	Yes	110	Yes	153	No	589

Table 3: Results for eval #2

G^{type}	$eval\_1$		$eval\_2$		$eval\_3$	
	$S \Rightarrow *w_1$	#deriv	$S \Rightarrow *w_2$	#deriv	S $\Rightarrow *w_3$	#deriv
G_b^c	No	1081	Yes	660	Yes	3741
G_b^b	No	276	No	561	No	946
G_{cnf}^{cnf}	No	3247	Yes	861	Yes	11749

Table 4: Results for eval #3

for the case when $stp_c = 100$, the derivation is as expected, i.e. for cases where derivation was done within 100 steps, CFGDeriver gives the correct answer.

The CFGDeriver looped and reached timeout for the first eval suite, except when using a bound which was derived from stp_b and was quite low. As a matter of fact, even with stp_b, we can see that CFGDeriver looped quite a while before giving up and concluding that deriving the given string will not be possible within the given number of steps. This could be because of the fact that the first grammar contains an indirect left production through $S \rightarrow A \rightarrow S$ and the first derivation was $S \rightarrow ASA$.

We can also see that for all other terminating cases, G_{cnf} takes quite a lot more steps than the other derivations, primarily because the variable expansion happens much more in a CNF form, because of the restriction on the number of production tokens each production can produce (which is 2). This also leads it to check for more productions of each variable across all dead-end paths which increases the total number of variable expansions.

Apart from this another interesting thing to be noted is that no grammar could be derived by CFGDeriver with stp_b bounds. This can be attributed to the low value of bounds that we calculate for each string given its length and the length of longest production in the respective grammar.

4 Conclusion

The project successfully achieved its goals in parsing CFGs and buidling PDAs, allowing for the validation of string membership in CFG languages. The results were compared with Stanford's CFG Developer tool, providing a reference point for the correctness of CFGDeriver.

Observations from the benchmarks revealed that the CFGDeriver performed well within the specified bounds ($stp_c = 100$) and for grammars in CNF form. However, challenges were encountered, particularly in handling indirect left productions, leading to timeouts in specific cases with eval suite 1.

The use of stp_b bounds presented interesting insights, showing that CFGDeriver struggled to derive strings efficiently within the low bounds calculated. This highlights the complexity of deriving strings within a limited number of steps, especially in non CNF forms where variable expansion is more loose.