An Implementation of a Fast Threaded Nondeterministic LL(*) Parser Generator

Amr M. AbdelLatif
Faculty of Computing and Informatics, Zagazig University, Egypt

Amr Kamel
Faculty of Computers and Information, Cairo University, Egypt

Reem Bahgat
Faculty of Computers and Information, Cairo University, Egypt

ABSTRACT
Parsers are used in many applications such as compilers, NLP and other applications. Parsers that are developed by hand are a complex task and require a generator to automatically generate the parser. The generator reads a grammar and generates a fully working parser.

This paper proposes separating the semantic actions' execution from the parsing phase. The parser generates a queue of semantic actions attached with grammar rules to be visited in case of successful parsing. By this separation, the execution time of the parsing phase can be enhanced. More importantly, this will avoid the incorrect execution of semantic actions when dealing with non-deterministic grammars. Investigating an implementation for the parallelization of the parsing phase for non-deterministic rules is also another contribution of the paper. A previous theoretical work of this paper was made in [1]. The experimental work shows that working with single threaded backtracking with storage of intermediate results, as well as following the Fork/Join parallel execution model without intermediate storage perform in most cases better than working with raw threads execution or by predicting rules as in ANTLR V4. The generator assumes that the grammar is left-recursive free.

General Terms
Compiler, parsers and parser generator

Keywords
Non-deterministic grammar, LL grammar and compilers

1. INTRODUCTION
Many applications use parsers as a primary phase. The applications range from browsers to display web pages, search engines to find web pages, checkers in word processors, to compilers. For these parsers to work, the scanned tokens must be well formed and described by specific grammars. These grammars are always varying and may be non-deterministic. If the developer's intension is to build a parser by hand, it would be a complex and costly task. Parser generators are used instead. Parsers generally fall into two types: bottom-up and top-down.

Existing parser generators such as YACC [2], Bison [3] and CUP [4] are types of LR parser generators but require the grammar to be LALR(1). They use a bottom-up approach and only handle deterministic grammars. To resolve non-determinism, the GLR [5] is an attempt to handle ε-grammars; an example of GLR parser generators is Elkhound and an example of Elkhound-based C++ parser is Elsa [6]. The generated parser by Elkhound can launch sub-parsers to walk on all available paths. If one failed, it should die. Surviving parsers result in a parse forest. Elkhound proved to have an efficient parsing time.

Non-deterministic LR grammars can also be parsed with backtracking. [7, 8, 9, 10] are some attempts to use backtracking. The backtracking approach suggests that when finding a non-deterministic rule, then a trial parsing is done to the first alternative. If the first alternative failed then backtrack to try another one. Parsing with backtracking is not free of errors when thinking of semantic actions' execution. Some actions would be executed during parsing the wrong rule. Merrill [9] suggests predicates in the form of conditional and unconditional actions. For example the unconditional actions would be executed in case of failure or success of parsing an alternative rule. This is shown in fig 1. Thurston [10] added an undo action so he is handling three action types namely, trial, undo and final action. The problem is that too much time is required doing a trial action and revert it by doing the undo action.

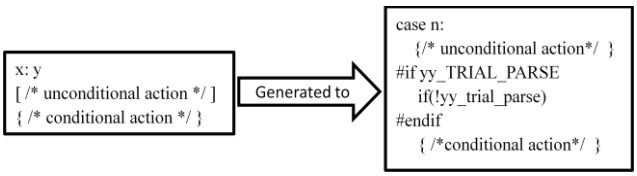

Fig 1: Using conditional and unconditional actions by Merrill [9]

Regarding top-down parsers, which are used to parse type of grammar called LL, a set of parser generators are used such as Coco/R [11], ANTLR 3 [12], ANTLR 4 [13], and JavaCC [14]. They require the grammar to be LL(k). JavaCC resolves non-determinism by increasing the lookahead and requires the programmer to do it explicitly. It also uses backtracking techniques without storing intermediate results. ANTLR 3 can parse LL(*) by analyzing and building a DFA from the specified grammar to avoid backtracking. If it can't build a DFA, it asks the user to specify the option {backtrack=true} explicitly.

Parsing expression grammars (PEGs) [15] introduces ambiguity in the first place. It uses a prioritized operator "/" to order the rules to be visited. PEGs also introduces some predicates for false and successful parsing. Ford had implemented parsers for PEGs like packrat and its generator Pappy [16, 17] which is written for Haskell. Rats and Mouse [18, 19] are other implementations and are written for java. Mouse uses a very simple recursive descent parser with backtracking but doesn't store intermediate results; this make the parser inefficient. (Rats!) memorizes all intermediate results to ensure linear-time performance.

1.1 Problems Of Using Backtracking

Although backtracking allows writing grammar without being concerned about the non-determinism and without limiting the number of lookahead symbols, it has a critical side effect. Not only backtracking causes exponential execution time in the worst case [15] as it has to visit all alternative paths to obtain the correct path, but also it has bad effects concerning executing false actions as well as ignoring pre-visited semantic actions.

Let us take a non-deterministic grammar like the one shown in fig 2 and the corresponding Nondeterministic Finite Automata (NFA) in fig 3. This grammar is non-deterministic. The actions are placed between {}. Two possible sets of tokens can be matched by this grammar, that is a set of a's followed by zx or zy, such as "aaazx" or "aaazy". If the input token is "aaazy", the decision which rule to select is delayed until matching the last symbol. When this grammar is tested with ANTLR 3, the execution tends to navigate the first alternative and successfully eating the set of a's and each time it eats a symbol a, it displays a message x. Upon discovering that the first alternative is the wrong one, it tries the second alternative, the successful one, which displays y. Clearly false actions are executed as well as true actions. The output is "xxxzyyyz"

```
s : x z 'x' | y z 'y' ;
z: 'z' {System.out.println("z");} ;
x : 'a' {System.out.println("x");} x | ;
y : 'a' {System.out.println("y");} y | ;
A : 'a';
X : 'x';
Y : 'y';
```

Fig 2: Example of a non-deterministic grammar augmented with semantic actions.

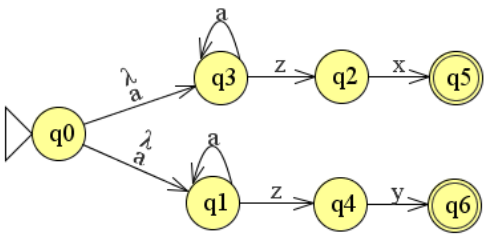

Fig 3: NFA of the grammar in Fig 1.

ANTLR 4 had solved the problem of non-determinism by predicting which rule to be investigated. It calls a special prediction function that returns the rule number. The function simulates the execution of rules by building an augmented recursive transition network (ATN), a diagram like a syntax diagram. The simulation is done using multithreading techniques. By this way, it avoids traversing false rules and hence executing false actions. The drawback of this way is the execution time as prediction and visiting the rules double the time required to parse the input tokens.

The same grammar was tested with the Rat parser generator after replacing | with priority operator / and after modifying the grammar to match the syntax of Rat. The output was "xxxzyyy" while it is supposed to display the output "yyyz". By analyzing the reason for that, the generated parser tries the first alternative and executes the actions and displays "xxxz". Intermediate results are stored so as not to visit it for a second time and achieve efficiency in the execution time. When the first alternative failed, the second alternative is tested and the rule *Z:'z'* is ignored and hence its action is not executed. That was another weakness of storing intermediate results to be skipped at later times.

The main problem facing parsers that handle the non-determinism either by backtracking or by parallel execution of sub-parsers is that they fall in the weakness of executing the semantic actions that are attached with the grammar rules. An undesired action may be executed during parsing false alternatives. This problem causes the parser generators to be not practically in use.

The main contribution of the paper is to generate parsers that separate the parsing phase from the semantic action execution. The job of the parser is to match the order of tokens against a predefined grammar and the job of the semantic action phase is to execute semantic actions. So the parsing phase will be split into two sub-phases. Another contribution of the paper is to avoid using backtracking and parallelizing the non-deterministic rule derivation.

The advantages of separating semantic actions execution from the parsing phase are:

- The readability of the grammar is achieved especially when the grammar is complex. Complex semantic action statements are placed in external modules.
- It the parser results in an error, then semantic actions would not be executed.
- Allow researchers to optimize the parsers without considering whether the optimization would contradict with action's execution.
- Make sure that the action attached with a rule is executed only if it is the correct rule for the parsed expression.
- The parser generator can name local and global variables with any names, and not to worry about a conflict with the generated names.

The remaining of the paper is organized as follows. Section 2 describes the grammar used by the generator and how semantic actions are added. Section 3 explains the process of generating the parser and semantic actions by the generator. Parallelizing the parser with multi-threaded and fork/join models is explored in section 4. In section 5, we discuss the way of storing intermediate results to avoid re-parsing the same part twice. The experimental work and our conclusions are presented in sections 6 and 7 respectively.

2. THE GENERATOR

The generator is used to read a grammar and generate a parser that can recognize tokens as well as execute semantic actions. The following is an example of a grammar. The grammar supports all EBNF features, and it is much like the one used by ANTLR. Non-terminals start with small letters. Terminals start with capital letters. Quoted terminals must have an explicit definition. Terminals are defined with regular expressions. Each production is terminated with a semicolon ';'.

```
options{ maxThreads='2';}
stmts : (stmt ';')+;
stmt  : ID '=' expr {stmt};
expr  :  Term '+' expr {add}
       | Term '-' expr {sub}
       | Term {term};
term  :  factor '*' term {mult}
       | factor '/' term {div}
       | factor {factor};
factor: ID | NUM | '(' expr ')';
ID    : [a-zA-Z][a-zA-Z0-9]*
NUM   : [1-9][0-9]*;
EQ    : '=';
ADD   : '+';
SUB   : '-';
MULT  : '*';
DIV   : '/';
skip WS : [ \r\t\n]+;
```

Fig 4: Sample grammar for arithmetic expressions.

The generator has three keywords. The keyword 'skip' means that this token is recognized by the lexer but not stored in the token stream. The keywords 'explicit' and 'implicit' are used with terminal rules. If implicit is used, this means that this part can only be used by other parts. For example the ID regular expression can be broken into smaller parts as shown in fig 5.

```
implicit LETTER: [a-zA-Z];
implicit DIGIT : [0-9];
implicit LETTERORDIGIT:  LETTER | DIGIT;
ID : LETTER LETTERORDIGIT*;
```

Fig 5: Breaking ID into sub-terminal rules.

Semantic actions are inserted between {}, similar to the Mouse parser generator [19]. A single word is placed between {} which is converted to a method call. But unlike the Mouse, the context information is passed to the methods. Semantic actions can be written at any part of the non-terminal production. The generated methods will be called later after the parsing phase has completely finished.

The grammar doesn't support direct nor indirect left-recursion. The grammar written is assumed to be LL(*). The first rule is assumed to be the start rule. The grammar is easy to write and understand by the programmer as it avoids the complexity of augmenting complex semantic actions. Non-determinism is allowed within the rule terms. For example the rule r: a (b c)? (b d)*;

is non-deterministic. The generated parser may try the first alternative (b c) and if it fails it backtracks to try the second alternative.

The *options* keyword allows the user to specify the threshold as the maximum number of threads (or sub-tasks) to be created. The user can specify the value 'auto' to equate the number of sub-tasks with the number of cores on the host machine.

Terminal rules are converted to a form recognized by JLex [20]; a lexical scanner for Java that is based on the famous Lex [22]. It can recognize the longest match and can give priority for tokens by their appearance in the specifications. Implicit tokens are converted to JLex macros. Non-terminal rules are converted to methods. Semantic actions are numbered with unique numbers. For instance, the above grammar has 7 semantic actions. Semantic actions are attached with information, the current, previous and next token. After the parsing phase has finished, all correct semantic actions that are to be executed in order are added to a general queue. The next phase is to iterate over all semantic actions stored in the queue and call them in the sequence that they appeared. For example if the input string is "x=1*2*3;" then the queue will contain [term, factor, mult, mult]

3. THE GENERATED CODE

Three files are generated, namely Parser.java, Lexer.java and SemanticAction.java. For the grammar written in Fig 4, seven semantic action methods are generated. These methods are [stmt, add, sub, mult, div, term and factor]. Fig 6 shows the *SemanticAction* class. The Context contains some information related to the rule such as current token associated with the action and can be used to store results of any computations to be passed to another called methods.

```
public class  SemanticAction{
    //(1) stmt  : ID '=' expr
    public void stmt(Context context)
    {
        //add code for stmt here
    }
    //(2) expr  :  Term '+' expr
    public void add(Context context)
    {
        // add code for
    }
    .
    .
    .
}
```

Fig 6: The generated SemanticAction class.

The generated parser is direct recursive descent and very easy to understand. Each deterministic rule generates a separate method. For a non-deterministic rule, each alternative is encapsulated in a separate method. Each method returns a Boolean value to represent the state of parsing either fail or success. The parsing method starts by first checking whether that part of input was parsed earlier (it may be parsed earlier and the result is stored in the intermediate storage, see section 4). If it was parsed earlier, then the method skips parsing this part and returns immediately. Otherwise it must start parsing. After finishing parsing, it may need to store the result obtained into the intermediate storage. Fig 7 shows a method generated.

```
        //stmt : ID '=' expr
      public boolean parseStmt(){
   if(isNodeInStore(forwardPointer,2))
                return true;
     int size = visitedRules.size();
      int startFrom =forwardPointer;
        visitedRules.addFirst(2);
            switch(tok.type){
                case ID:
    if(! Eat(TokenType.ID)) return false;
  if(! Eat(TokenType.EQUAL)) return false;
```

```
            if(! parseExpr()) return false;
                        break;
                    default:
                   return false;
                       }
             if(this.isAmbiguousThread)
                ids.addNode(startFrom,
                    2,getRulesUpToSize(size),
                        forwardPointer);
                  return true;
                       }
```

Fig 7: Method generated for rule "*stmt: ID '=' expr*"

The third file is generated using JLex and all tokens are loaded into memory before the parsing process is done.

4. PARALLEL PARSING

One way to avoid using backtracking is to allow parallelism on deriving non-deterministic rules. Parallel execution can reduce the parsing time as each thread executes on a core on a multiprocessor system. This section shows how parallelism is done using raw threads and using the Join/Fork framework.

4.1 Parallel Parsing with Threads

When the parser reaches a set of alternatives to be investigated and the selection decision is not known from the current lookahead, then other instances (threads) from the same parser are launched. Each sub-parser starts parsing from the point of non-determinism. The main one waits for the successful one to return back with its result. If one thread fails then the result from that thread is ignored and the other threads continue. If all sub-parsers or threads fail to parse the remaining tokens then the main thread returns with failure.

Each thread must maintain a set of information necessary for parsing, such as a marker to store the starting token to be parsed. The value of this marker is set by the main thread that created it. Also each thread has a queue to store all visited rules. Each thread stores its result in a queue as its output and if it parses successfully, its contents are copied to the parent thread queue. Since threads representing alternative rules are accessing the same set of tokens in the memory simultaneously, multiple instruction single data (MISD) is used and tokens are shared between all threads.

A single thread may branch more sub-parsers as it may encounter more alternative choices or it may select to work with backtracking. The selection is based on the coordinator and the branching threshold. If the number of currently active threads exceeds the threshold value, then no more sub-parsers are created. Creating more threads can be efficient especially on a device with multi-processors. Fig 8 shows three threads started as there are three alternatives for parsing the rule [`expr:Term'+'expr|Term'-'expr|Term;`].

Threads can coordinate their work with each other by means of storing intermediate results obtained in a shared data structure. A thread may not need to parse a set of tokens and skip them if it finds their parsing information memorized by other threads. Concurrent read on shared tokens creates no problem. But concurrent write on a shared memory raises a problem. One solution to this problem is to synchronize access to the shared memory and treating the shared memory as an atomic structure. By putting synchronized keyword on the shared memory methods, all threads competing for storing their information need to access the shared memory. The faster thread takes the lock first and all remaining threads wait for the lock to be released. The tricky part is to prevent storing the same information parsed by two threads more than once so a check must be done for the existence of information in the shared storage. Fig 9 shows an algorithm for thread creation.

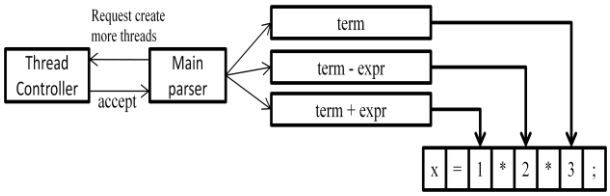

Fig 8: Three threads started from the main one for three alternatives.

```
Parsing Alternatives Algorithm
1 – IF there are non-deterministic
        alternatives THEN
2 –  FOR each alternative
3 –    Create sub-parser
4 –    Start sub-parser
5 –  ENDFOR
6 –  Wait for all sub-parsers to finish
7 –  Collect results
8 – FOR each sub-parser
9 –  IF result equals parsing successfully
                THEN
10 –    Add sub-parser result to main QUEUE
11 –    RETURN TRUE   %parse true
12 –   ENDIF
13 –  ENDFOR
14 –    RETURN FALSE  %parse error
15 –   ELSE
16 –   Continue parse remaining tokens
17 – ENDIF
18 – END
```

Fig 9: Parsing alternatives algorithm with threads

4.2 Parallel Parsing with Fork/Join Model

Another way to employ multiprocessing efficiently is by using the Fork/Join framework. Java 7 has recently added a Fork/Join framework in its library [21]. The Fork/Join is a multi-threaded programming style that works with divide-and-conquer approach. It allows the problem to be divided into smaller sub-problems; each sub-problem can be solved by the same or different way from the main problem. The process of division continues until reaching the atom. The atom is the smallest problem that cannot be divided and must be solved directly.

The benefits of using the java Fork/Join framework is that it can manage tasks in the same way the operating system manages threads. The difference is that it manages tasks in a light weight manner but the operating system manages threads in a heavy weight manner. Threads are created only one time and saved in a thread-pool area, thus avoiding thread allocation and re-allocation. Each main task and its sub-tasks that are held in a queue are scheduled to a thread. Threads can be created equal to the number of cores (by using java function Runtime.getRuntime().availableProcessors()) or as the programmer specifies. Another benefit is that if a main task and all its sub-tasks are idle (e.g. waiting for some event to occur), the system can steal tasks from other threads. Since the cost of constructing a new thread is greater than the parsing time, the Fork/Join model has greatly enhanced the parsing time.

The parser model can follow the Fork/Join model. When encountering a set of alternatives to be parsed and the lookahead cannot be used to decide which alternative to derive, and if the remaining tokens are large enough, then fork sub-tasks to start parsing. Each sub-task has the set of

information as stated in the multi-threaded model. The algorithm for Fork/Join is shown in fig 10.

```
Parsing Alternatives Algorithm Fork/Join
    1 - IF there are non-deterministic
            alternatives THEN
  2 -     IF remaining tokens are small or reach
            threshold THEN
    3 -       Continue parse remaining tokens
                4 -     ENDIF
    5 -     Fork sub-task for each alternative
     6 -    Wait for all sub-tasks to finish
              7 -     Collect results
             8 -     FOR each sub-parser
        9 -         IF result equals parsing
              successfully THEN
10 -           Add sub-tasks result to main QUEUE
     11 -          RETURN TRUE    %parse true
                  12 -        ENDIF
                   13 -     ENDFOR
       14 -     RETURN FALSE   %parse error
                    15 - ELSE
     16 -     Continue parse remaining tokens
                  17 - ENDIF
                   18 - END
```

Fig 10: Algorithm for parsing alternatives with multi-threaded Fork/Join Model.

5. STORING INTERMEDIATE RESULTS

By storing intermediate results, the parsing time with backtracking can be reduced from exponential time to linear time [16, 17]. Clearly storing information avoids re-parsing the same part more than once. The same concept can be used with the multi-thread model. Storing intermediate results can reduce the parsing time. Due to the fact that the final result is a sequence of semantic actions in the bridge queue, each node in the storage is attached with an internal queue of the semantic actions. The table has two entries. The first entry is the parsed token number and the second entry is the rule number. The table uses hashing to map entries which takes time efficiency O(1) for the first entry and also for the second entry. Each node stored in the table stores a number to indicate the end token position of parsing. For example fig 11 shows an example of a table data structure constructed by parsing the input string "x=1*2*3;". The first entry 2 means that the thread started parsing a token in position number 2. From token position 2 to token position 2, it can be reduced by rule 10 or rule 8. When it is reduced by rule 10, then it should execute semantic action $\Omega_{10,}$ and when it is reduced by rule 8 then it should execute actions { Ω_{10} Ω_6}. From token position 2 to token position 6, it can be reduced by rule 6 with semantic actions { $\Omega_{10}\Omega_8\Omega_{10}\Omega_8\Omega_{10}\Omega_8\Omega_6\Omega_6$}.

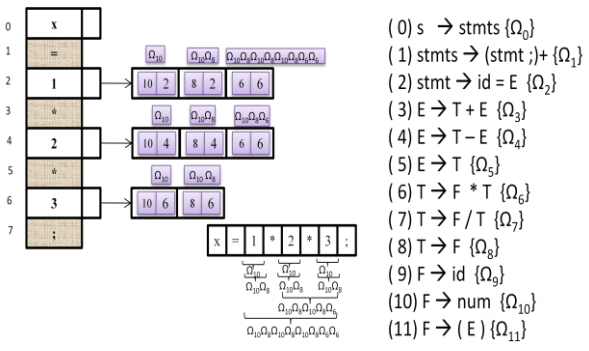

Fig 11: Construction of shared storage

To optimize the storage used by the parser, two enhancements are done to the data structure. The first is that it is not necessary to store each entry of the token number. In contrast with [16] which creates a storage of M(N+1) where M is the number of methods (rules) and N is the number of tokens in the input string in addition to the empty string. For example, tokens [=, * and;] are complementary parts to the rules and they are not used alone in reduction, so it is not necessary to store them. The second enhancement it that it is not necessary to store nodes in the table except in case there is non-determinism. This is because with deterministic rules there is only one path to follow and it is not possible to seek alternative paths as in the case of non-deterministic rules.

5.1 Storing Intermediate Results

When using a single thread, it will be the only thread that accesses the table and stores its results. When using multiple threads, they will all compete to access and store their results. The shared memory will be a bottleneck and must be protected against concurrent access by multiple threads. Concurrency control causes only one thread to be active and the others waiting to access the shared memory. Only one thread can write at a time. This will increase the execution time. We didn't use intermediate storage with Fork/Join model in order to avoid concurrent write operations. We used intermediate storage with single threaded model.

6. EXPERIMENTAL WORK

In this section we show the time and memory measurements for our experiments. We show the impact of different implementations of the parser, the effect of applying multithreading and the multi-threaded Fork/Join model. The experiments are taken from our previous research work done in [1]. The time is compared with ANTLR V4 and JavaCC [14]. We used syntactic lookahead with JavaCC to allow the resolution of non-determinism. The time measurements do not include semantic action execution, only the parse time is measured. Time is measured after the two executions so as to make sure that the Java Virtual Machine has its stable state. Moreover, the average of three consecutive measurements is recorded.

All the experiments are done on a machine with processor model of Intel® Core™ i5-2450M CPU @ 2.5GHz 2.5GHz. Memory is 4 GB. The operating system used 64-bit Windows 7. Java Development Kit version jdk1.8.0_05 is used as a compilation environment. Memory is measured according to the equation:

```
Runtime runtime = Runtime.getRuntime();
long memory = runtime.totalMemory() -
runtime.freeMemory();
```

We didn't compare the results with Rats as it executes some false semantic actions while preventing some semantic actions from execution.

6.1 Experiments with Planned Test

Working on the grammar listed in fig 4 and varying the input size, we notice that only two rules have non-determinism, namely:

```
expr:Term'+'expr|Term'-'expr|Term;
term:factor'*'term|factor'/'term|factor;
```

6.1.1 Planned Test 1

Two types of input strings are planned to be tested. The first type examines the depth in the first rule. For example "x=1*2*3*-------*n;" is a set of multiplication operators,

which is the worst case analysis as the first and the second alternatives of the first rule would fail and the third alternative would succeed. By testing this type of input, three threads will be created, one for each alternative of the first rule, and all of the three alternatives remain alive until reaching the token by which a thread can either succeed or fail. For the second rule, the decision would be quick by eating a factor and scanning the next input token '*'; the first thread keeps alive and the remaining alternatives die.

Fig 12 presents a graph of this planned test. The graph illustrates that working with raw threads (4 threads allocated as the test runs on 4 core processors) has the worst time analysis as it grows tremendously with small increase in the file size. The graph compares also the run time for multi-thread execution without storing intermediate results, which performs better than the case when storing intermediate results. Working with only one thread and working with the Fork/Join model is more efficient than ANTLR. JavaCC has the shortest time from all tested models. The tests were made on input file sizes up to 7000 bytes, as ANTLR V4 gave an exception message for the higher sizes. JavaCC, Fork/Join and single-thread models seem to be coinciding due to their small time measurements.

"Exception in thread "main" java.lang.StackOverflowError at org.antlr.v4.runtime.atn.ATNState.getNumberOfTransitions(ATNState.java:178)"

Another zoomed version of the graph without the multi-threaded model is shown in Fig 13. The fig shows that working with only one thread and working with Fork/Join model is more efficient than working with ANTLR V4. JavaCC and one-threaded are very close, but JavaCC is more efficient.

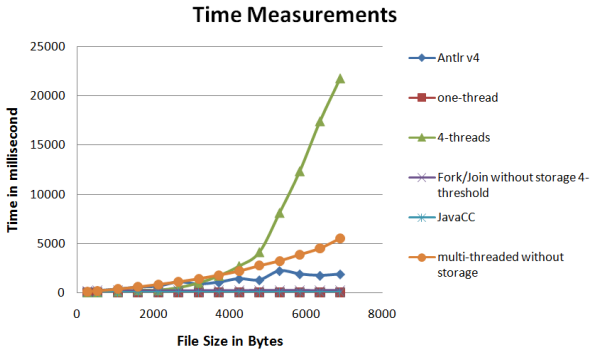

Fig 12:.Time Measurements comparison between Antlr v4, one-thread, multi-thread (4 threads), Fork/Join, JavaCC and multi-thread without storage (plan 1).

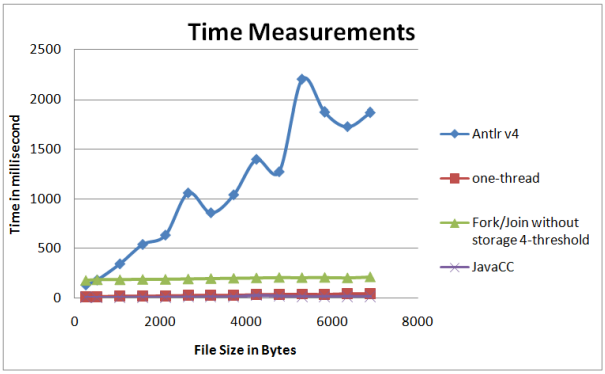

Fig 13: Time Measurements comparison between Antlr v4, one-thread, Fork/Join model and JavaCC (plan 1).

6.1.2 Memory Measurement of Planned Test 1

The memory comparison is shown in fig 14. The comparison shows that one-thread, Fork/Join and Javacc have the least memory consumption. Antlr V4 requires extra storage to store the NFA of the ATN simulator. The 4-thread model with share intermediate storage consumes high storage as each thread accesses the intermediate storage to store its information. The multi-thread model without sharing intermediate storage also consumes high memory as each thread constructed must have its own context that consumes an amount of memory.

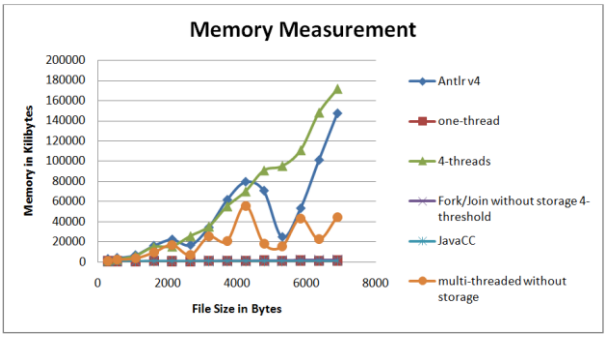

Fig 14: Memory measurements comparison between Antlr v4, one-threaded, multi-threaded (4 threads), Fork/Join and JavaCC modes (plan 1).

6.1.3 Analysis of Planned Test 1

In our opinion, there are two reasons behind the inefficient performance of the multi-thread model. The first reason is the frequent allocation and de-allocation of threads which can be solved by using a thread pool. The thread pool allows the creation of threads and allocation of resources only once and re-using threads many times as needed. The thread allocation time can be more costly than parsing the part allocated to that thread. The second reason is in the shared data structure in case of shared memory for storing intermediate results. Storing intermediate results is supposed to reduce the parsing time. Since the shared data structure allows many threads to access it at the same time, so synchronization must be done to prevent concurrent write problems. Synchronizing the shared data structure causes one thread to acquire the lock while other threads to be blocked waiting for the lock.

Both the multi-threaded Fork/Join model and working with only single thread seem to have linear graphs with very small slope. The multi-threaded Fork/Join model costs some extra time to allocate threads which is a constant time. The Fork/Join model graph is less than ANTLR V4 as it allocates threads only once and also avoids the problem of locking threads on the shared data structure. We didn't use intermediate storage to store intermediate results obtained from each thread in the Fork/Join model in order to avoid falling in the concurrency problem and also to reduce the memory usage. The worker stealer feature implemented in the Fork/Join model allows the idle thread to steal some tasks from other busy threads. This feature maximized the CPU utilization and the task throughput.

6.1.4 Planned Test 2

The other type of input string that is tested is the variation of depth in both rules, like "x=1*2*-----*n / 1*2*-----*n + 1*2*-----*n / 1*2*-----*n - 1*2*-----*n / 1*2*-----*n;". Fig 15 shows the comparison between ANTLR V4, one-thread model, multi-threaded model, multi-threaded Fork/Join and JavaCC. At first, it seems that using multi-threads is faster than

ANTLR V4. But for the same reasons stated in the analysis of plan 1, the graph tends to grow very fast by small increase in the input file size. Multi-threaded Fork/Join model is more efficient than ANTLR V4. A more detailed graph without multi-threaded model is shown in fig 16. The Fork/Join model and the one-threaded model are more efficient than ANTLR for the same reasons stated in the analysis of plan 1. JavaCC is the most efficient one.

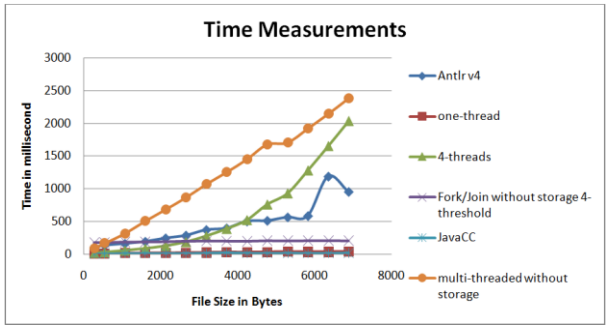

Fig 15: Time Measurements comparison with multi-threaded (plan 2)

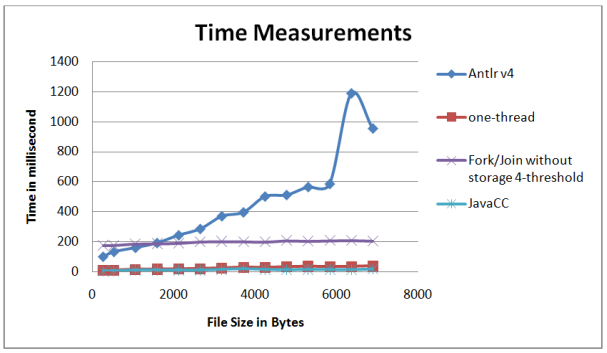

Fig 16: Time Measurements comparison without multi-threaded (plan 2)

Memory measurement of plan 2 is shown in Fig 17. JavaCC, Fork/Join and one-threaded models seem to be coinciding due to small values. The one-threaded model has greater memory as it needs to store intermediate values but the Fork/Join model has less memory as it doesn't need to store intermediate values.

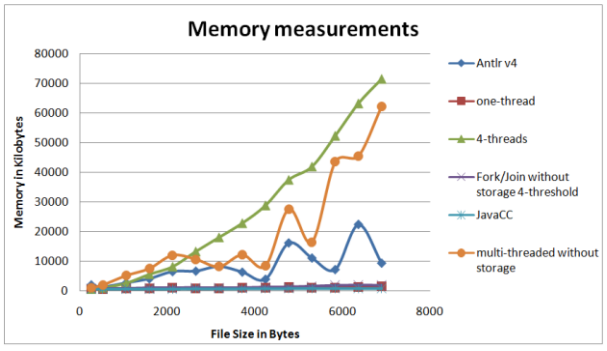

Fig 17: Memory Measurements comparison between Antlr v4, one-threaded, multi-threaded (4 threads), Fork/Join and JavaCC modes (plan 2).

6.2 Experiments with java Grammars

In this section, we test the parser generated from the Fork/Join model with threshold 4 on java source files. 7705 source files with size 81.6 MB from the JDK 1.8 are used in the experiment. The grammar was taken from Antlr V4 site (https://github.com/antlr/grammars-v4). Antrl provides two version of java grammar and we had selected the fastest one to test with it. The experiments show that Antlr has the fastest execution time, but the worst use of memory. JavaCC is the best in memory measurements as it doesn't load all tokens in memory. The Fork/Join without storage has time less than JavaCC by about 5.01% but more than Antlr V4 by about 20.39%. The Fork/Join without storage has memory consumption less than Antlr V4 by about 27.39% and more memory than JavaCC by about 32.35%.

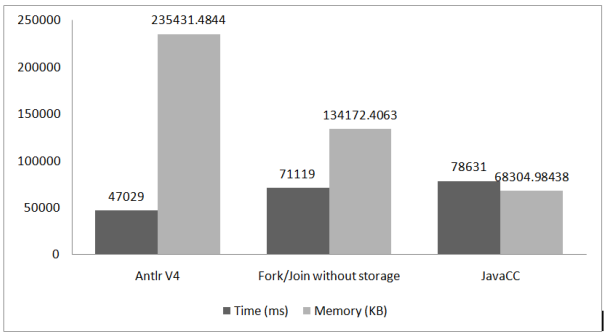

Fig 17: Time and memory measurements for Java source files.

7. CONCLUSION AND FUTURE WORK

The paper proposed separating semantic actions' execution from the parsing phase. This allowed us to avoid the incorrect execution of semantic actions and reduced the parsing phase execution time. It has also allowed for the smooth parallelization of the parsing phase. The non-determinism can be solved by launching multiple threads to parse the different alternative rules. A practical parser generator is done for it. The generator can generate a recursive descent parser with multi-threaded or Fork/Join model. The analysis shows that working with the multi-threaded Fork/Join model can be practically used within a time and memory accepted by users even for small PCs. Creating a parser that executes with single thread and backtracking has approximately the same parsing time as the Fork/Join model but due to the thread allocation constant time, the Fork/Join model takes a small extra constant time than the one-threaded model. The Fork/Join model consumes less memory than the single-threaded model.

By separating the semantic actions execution from the parsing phase, researchers are encouraged to find more methods to enhance the parsing time. We are seeking for more parsing improvements that can benefit from the parallelism and multi-core technologies which became available to all users.

8. REFERENCES

[1] Amr M., Amr K., Reem B.,"TGLL: A Fast Threaded Nondeterministic LL(*) Parsing". ARPN journal of systems and software, VOL. 5, NO. 2, August 2015

[2] Johnson, S.C., "YACC: Yet another compiler compiler". In *UNIX Programmer's Manual* (7th ed.), Volume 2B, 1979.

[3] Donnelly, C., Stallman, R.M., "Bison: the YACC-compatible Parser Generator". Bison Version 1.28. *Free Software Foundation*, 675 Mass Ave, Cambridge, MA 02139, 1999.

[4] Appel, A. W., Flannery, F., and Hudson, S. E., "CUP parser generator for Java".

http://www2.cs.tum.edu/projects/cup/,1999, Last visited on March 4th 2015.

[5] Nozohoor-Farshi, R., "GLR parsing for ε-grammars". In *Tomita, M., ed.: Generalized LR Parsing*. Kluwer, pp. 61-75**,** 1991.

[6] McPeak, S., and Necula, G. C., "Elkhound: A fast, practical GLR parser generator". In *Proc. Of 13th International Conference on Compiler Construction*, vol. 2985 of LNCS, Springer, pp. 73-88, 2004.

[7] Dodd, C., and Maslov, V., "Backtracking Yacc". http://www.siber.com/btyacc/, last visited on April 3rd 2015.

[8] Spencer, M., "Basil: A backtracking LR parser generator". http://www.lazycplusplus.com/basil/, last visited on April 1st 2015.

[9] Merrill, G. H., "Parsing non-LR(k) grammars with Yacc". *Software, Practice and Experience*, 23(8), pp. 829–850, 1993.

[10] Thurston, A. D., and Cordy, J. R, "A backtracking LR algorithm for parsing ambiguous context-dependent languages". In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, October 16-19, 2006, Toronto, Ontario, Canada.

[11] Mössenböck, H., Löberbauer, M., and Wöß, A., "The Compiler Generator Coco/R". http://www.ssw.uni-linz.ac.at/Coco/, last visited April 2nd, 2015.

[12] Parr, T., Fisher, K., "LL(*): the foundation of the ANTLR parser generator". In *Proceedings of PLDI 2011*, pp. 425-436, 2011.

[13] Parr, T., Harwell, S., and Fisher, K., "Adaptive LL(*) parsing: the power of dynamic analysis". In *Proceedings of OOPSLA 2014*, pp. 579-598, 2014.

[14] Viswanadha, S., Sankar, S., and Dunkan, R., "Java compiler compiler (JavaCC)-The java parser generator." *Java.net*, https://javacc.java.net/, last visited Aug 2 2015.

[15] Ford, B., "Parsing Expression Grammars: a Recognition-Based Syntactic Foundation". In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pp. 111–122, New York, NY, USA, 2004. ACM Press.

[16] Ford, B., "Packrat Parsing: a practical linear-time algorithm with backtracking". Master's thesis, *Massachusetts Institute of Technology*, September 2002.

[17] Ford, B., "Packrat parsing: Simple, powerful, lazy, linear time". In *Proceedings of the 2002 ACM International Conference on Functional Programming*, pp. 36–47, Pittsburgh, Pennsylvania, Oct. 2002.

[18] Grimm, R., "Practical packrat parsing." *New York University Technical Report, Dept. of Computer Science, TR2004-854*, 2004.

[19] Redziejowski, R. R., "Parsing expression grammar as a primitive recursive-descent parser with backtracking." *Fundamenta Informaticae* 79, no. 3, pp. 513-524, 2007.

[20] Berk, E. J., and Ananian, C. S., "JLex: A lexical analyzer generator for Java (TM)". *Department of Computer Science, Princeton University. Version* 1, 2005.

[21] Lea, D. "A Java fork/join framework". In *Proceedings of the ACM 2000 conference on Java Grande*, pp. 36-43. ACM, 2000.

[21] Lesk, M. E., "LEX — A Lexical Analyzer Generator". *Computing Science Technical Report 39, Bell Telephone Laboratories*, Murray Hill, NJ, 1975.