The Expressive Power of the Statically Typed Concrete Syntax Trees

Nikolay Handzhiyski[1,2], Elena Somova[1]

[1] *University of Plovdiv "Paisii Hilendarski", 24 Tzar Assen St., Plovdiv, 4000, Bulgaria*
[2] *ExperaSoft UG (haftungsbeschraenkt), 10 Goldasse St., Offenburg, 77652, Germany*

Abstract

The article specifies the definitions of a Concrete Syntax Tree and an Abstract Syntax Tree. The different types of knowledge that are shared between a parser and builder modules in a parsing machine, about the syntax tree building, are discussed. For the building of the syntax tree, various Syntax Structure Construction Commands are presented. They are transmitted from the parser to the builder, depending on the type of tree. Template grammars and a computer program (Parser Generator Profiler) that performs parser tests on their basis are described. The empirical results from the different tests (for different combinations of grammar elements), performed with different types of syntax trees, for different parsers generated by different parser generators, are shown. The measurements are based on different criteria such as the time for the tree building, its traversal time, its destruction time, and the memory used by it.

Keywords

Concrete Syntax Tree, Abstract Syntax Tree, Template Grammar, Parser Generator Profiler

1 Introduction

There are a large number of parsing machines (PMs) that extract information from unstructured data, through a process called parsing (syntax analysis). This information can be represented in different ways, as this article discusses PMs that represent the information as syntax trees. The syntax trees are built based on the rules of formal grammar (called only **grammar**). The derivation of the different parts of a tree is done in steps. The different PMs do that differently, mainly due to the different parsing algorithms that they use to perform the parsing of the data. This article discusses syntax trees that are derived from top to bottom – the root of the tree is derived out first, then the leftmost node in-depth and finally the rightmost node.

A PM performs two main tasks: first, it checks if a string of characters [1] belongs to a given language, and second, builds a syntax tree – a data structure that contains syntactic information about the string. The following questions are of practical interest: how much memory the syntax trees use, how much information they contain and how long it takes to work with them (creation, traversing, and destruction). Clear criteria are needed to assess the different structures in a syntax tree, on which to assess the grammar, according to which this tree was created.

The main purpose of this article is to propose an approach for the building and the classification of syntax trees that are built by a PM. Additionally, the article contains empirical measurements of parsers, generated from grammars with different combinations of grammar elements, according to different criteria (memory and execution time, called only resources). To measure the resources used by a large number of parsers (generated by different parser generators, for different grammars), a special purpose

Education and Research in the Information Society, September 27–28, 2021, Plovdiv, Bulgaria
EMAIL: nikolay.handzhiyski@experasoft.com; eledel@uni-plovdiv.bg
ORCID: 0000-0003-0681-6871; 0000-0003-3393-1058

meta syntax for describing grammars has been developed, described in the article, and applied in the tests.

Section 2 contains the related to the article previous work. Section 3 defines the problem that the article addresses. Section 4 defines the different types of syntax trees, according to clear criteria – the amount of knowledge used to build them. The different **Syntax Structure Construction Commands** (SSCCs) that can be passed from the parser to the next PM module are described. Template grammars and a program (Parser Generator Profiler) that performs tests on their basis are also presented. In Section 5 are shown different tests performed with different types of syntax trees for different parsers (generated by different parser generators), as well as the results obtained. Section 6 highlights the contributions and perspectives of the study.

2 Related work

In parsing theory, there are two main types of syntax trees called **Concrete Syntax Tree** (CST) and **Abstract Syntax Tree** (AST). It is a common understanding that an abstract syntax tree might not contain all grammar rules (for short rules) that are used during the parsing, and it might not contain some of the recognized characters that are implied from the context (for example, the parentheses around mathematical expressions can be omitted, if the expression between the parentheses has its own subtree [2]). This implies that a concrete syntax tree contains all of the used rules and recognized characters during the parsing.

There are no universal definitions of which tree is concrete and which is abstract, according to the information contained therein (based on the input data used by the parser for parsing) and according to its representation (based on grammar).

Some parser generators are building their trees without explicitly separating the modules to a parser (syntax analysis module) and a builder (tree-building module) [3]. When the PM modules have common structures (for example, the symbol tables in [1]), this prevents them from being easily separated (for example, to execute them on different dedicated threads), but when they do not have them, then their combining is trivial. The combining of modules only eliminates the transportation of the messages between them and does not change the way they work. The messages from the parser to the builder are containing SSCCs.

To build a syntax tree by commands is not new [4] (there called "tree-building directives"). However, how to do this and what the commands should be is a completely different topic, that is still open.

A viable part of the parse time is the constant factor multiplier that is often omitted from the parsing algorithms complexity analysis. This factor is important for practical reasons because it might render some algorithms usable or not in different scenarios. The well-known algorithm of Cocke-Younger-Kasami (CYK) has the runtime complexity of $O(n^3)$ [5], but accounting for the constant factor it is $O(n^3.|G|)$. If a given constant factor in the theoretical parse time complexity is not explicitly and exactly shown, that might obscure the expected algorithm performance in practice.

One might transform a given grammar to Chomsky Normal Form (CNF), to be able to use the CYK algorithm, but some grammar transformations might introduce excessive amounts of rules in this form [6]. Some are transforming extended Backus-Naur Form (BNF) grammars to BNF grammars and use them for parsing [7]. It is easy to show that a given Augmented Backus-Naur Form (ABNF) [8] grammar containing countable repetitions [9] of elements can lead to an excessive amount of rules in the CNF, BNF, and extended BNF by Wirth [10] (used by [11]).

Measuring of grammar metrics is done in [12], where the metrics are separated into two main types: a) based on the LR [1] automaton states (used to produce a bottom-up parser for the grammar); and b) based on the produced language (the average or the maximum size of the shortest accepted terminal symbols for each production, the number of different terminal symbol pairs and so on). In a contrast, we focus, in this article, on the measurements of the parsers (based on the grammars) in action, not on the description of their algorithms.

Even when two algorithms have linear to the input length resources requirements (usually for deterministic grammars), it matters whether the parsing will complete in a second or two [13].

3 Problem

At first glance, the popular definitions of the two types of syntax trees are clear: a) a concrete syntax tree contains concrete information – enough information to distinguish the different parts of the tree; and b) an abstract syntax tree contains less information than the concrete one, and lacks predefined information that is not needed or that can be calculated from the information already available. However, these are two definitions that, although both are expressing the difference between the trees, leave much uncertainty as to how much information is "enough" to distinguish the different parts of a tree, and by what criteria a concrete tree can become abstract.

In order to address the problem, the article must first define these two concepts in an appropriate way. Then, to be able to "score" the different trees, it is necessary to compare many generated trees by different parser generators, according to various criteria such as tree build time, traverse time, destruction time, and most of all, how the measurement results change for different combinations of grammar elements. These measurements will give an empirical idea of how many resources a given grammar is expected to require at runtime.

To extract meaningful statistics, a large number of grammars must be used that have a similar structure. In order to avoid the writing of all these grammars by hand (and increase the risk of mistakes), a tool must be created, with which a large number of grammars can be described in a short form. Thus, through this tool, it will be possible to perform a large number of tests (by compiling these grammars) on different parsers generated by different parser generators. As a result, it will be possible to derive empirical data on the "cost" of different grammatical constructions for different types of syntax trees.

A preprocessing of grammars is done in [14], but the syntax, as it is shown, requires the developer to follow a strict syntax for the output of each template. We aim to describe multiple grammars in a compact form that are evaluated multiple times in turn (not just preprocessed). We require that the output of each evaluation is a mixture of already computed previous grammar elements, and to have the ability to generate raw text anywhere in the grammars. A "raw text" here means that there must be no restrictions during its generation. That will allow absolute control over the generation of the final sequence of grammars.

Translation of grammars is done in [15], but the tool, as it is shown, does not have the means to produce a sequence of grammars by outputting raw text, nor to do a generation by logical statements in a series of evaluations. We aim to have a solution that not just translates a single grammar, but derives a sequence of grammars and then collects statistics about the runtime of these grammars when they are compiled with various tools and options.

4 Problem solution

In this section, it is assumed that the grammars are described with the ABNF meta syntax. Definitions of some basic concepts in the field are provided in [16] and used in this article. A PM is a set of modules that produces an output (that depends on the particular PM) from a given input. Two of the PM modules are relevant: a) parser – a module in a PM that outputs SSCC; and b) builder – a module that inputs SSCC from the parser. The possible builder types are: explicit (that builds a syntax structure) and implicit (that uses the SSCC without building anything from them). The explicit builder has an **architect** that builds the syntax structure itself and in the case where the structure is a syntax tree, it can be of two types: an **abstract architect** (builds an abstract syntax tree) and a **concrete architect** (builds a concrete syntax tree).

4.1 Knowledge and facts for the syntax tree building

When the modules of a PM are considered as completely separated, then a detailed description, of the knowledge that each module has about the whole data recognition process, is required. Emphasis is placed hereafter on the types of knowledge that exist between the parser and the builder modules about the building of the syntax tree.

We denote the set of facts with F, the set of facts that a module knows with K, where $K \subseteq F$, the set of the modules in a PM with M, and the knowledge of one module with $K(M_k)$, where $M_k \in M$. The

knowledge of the parser module ($M_p \in M$) and the builder module ($M_b \in M$) are $K(M_p)$ and $K(M_b)$ respectively. The parser always knows all facts: $K(M_p) = F$, but the builder might not know them all: $K(M_b) \subseteq F$. From now on, we assume that everything the builder knows, it also uses. Depending on how much the builder knows about the process, the following types of knowledge differ:

- **Full** – the builder knows everything about the syntax tree: $K(M_b) = F$. This means that the parser might only send information that cannot be calculated by the builder. This is information extracted from the input data;
- **Partial** – parts of the structure of the syntax tree are known by the builder: $K(M_b) \subset F$ and $K(M_b) \neq \emptyset$. The parser should then inform the builder in some way about the unknown to the builder facts;
- **None** – the builder knows nothing about the syntax tree: $K(M_b) = \emptyset$. The parser should then inform the builder in some way about the various structures in the tree.

The builder can have the knowledge about the structure of the syntax tree in the following ways:

- **Statically** – this knowledge is available to the builder at the time of its creation and it remains available during its operation;
- **Dynamically** – the builder acquires its knowledge for the syntax tree at runtime. This can happen: a) for each individual structure in the syntax tree, separately, at its discovery; or b) the knowledge about the tree is sent to the builder by the parser at once before the tree build begins.

In this article, the following facts, extracted from the parser grammar and affecting the syntax tree, are relevant:

- Which alternative is inside a particular rule or a group;
- Which concatenations are inside a particular alternative;
- Which token can be consumed by a particular terminal symbol;
- Which rule is referenced by a particular reference.

Depending on its repetitions, a particular grammar element can be represented differently (each type of presentation is a fact):

- **skippable** – when the element might be in the input from zero to one time;
- **single** – when the element must be in the input exactly one time;
- **list** – when the minimum number of repetitions of the element is less than the maximum number of repetitions and the maximum number is more than one;
- **array** – when the minimum and the maximum number of repetitions of the element are equal, more than one, and are a finite number.

4.2 Definitions

We classify the syntax trees built based on grammars written in the ABNF meta syntax because it is standardized and is used to describe a multitude of Internet protocols and data structures. The most relevant feature of the meta syntax is that it allows the different right-hand side grammar elements to have a minimum and a maximum number of repetitions. The classifications of the syntax trees are done by the information contained inside the tree, the representation of the tree, and the tokens stored inside the tree.

According to the information contained inside the tree, the syntax trees are:

- **Concrete** – this is a syntax tree that is built through the usage of all existing facts (extracted from the parser grammar): $K(M_b) = F$. The organization of the tokens received from the parser into data structures (statically or dynamically; as described later), by itself, is also concrete information that must not be lost during the construction of a contemporary concrete syntax tree. This means that the user of such a tree, must be able to unambiguously tell: a) which rules were parsed; b) which concatenation in a particular alternative was parsed; b) where in the tree the repetitions of a particular grammar element begin and how many are they; c) which elements are together in a group of elements; and d) all of the information about each token that was received from the parser;
- **Abstract** – a tree that was built without the usage of at least one fact: $K(M_b) \subset F$. This means that the maximum number of abstract trees is $2^{|F|-1}$ (to ensure that the tree is not concrete, one is subtracted

from the number of facts). This means that the user of such a tree will be able to unambiguously tell fewer things than when the tree is concrete.

According to their representation, the syntax trees are:

- **Dynamically typed** – a tree in which common data structures are used to represent different grammar elements. For example, all rules are represented by a single data structure and all tokens are represented by another single data structure. Then, a concatenation of elements is simply a list of nodes. To distinguish the different nodes from each other many dynamic checks at runtime are needed;
- **Statically typed** – different data structures (based on the facts from the grammar) are used for each individual grammar element. For example, each grammar rule has its own data structure in a given programming language and each concatenation, of each rule, has its separate data structure. Then each element in the concatenation has its own field in its parent data structure. Each field is an instance of a specific data structure, an array or a list of instances from the data structure. No dynamic checks are needed to distinguish any of the elements. To be able to find a particular instance of a data structure (created for a given grammar element) one can access its field directly in its parent structure (in a constant time). The programming language used to implement such a tree does not have to be with a strict syntax.

According to the tokens stored inside the tree, the syntax trees are:

- **Original** – all of the tokens from the input are stored in the tree without changes and any extra tokens;
- **Altered** – some of the tokens recognized in the input are not stored in the tree (or they are stored with changes) or extra tokens are inserted into the tree that was not found in the input. Such a tree can be produced when an error-correcting algorithm acts on the input.

4.3 Syntax Structure Construction Commands

During the building of the syntax tree, the builder can be instructed by the parser (through SSCC) to remove any of the already built structures by moving backwards on the tree. We assume that the builder keeps enough information to be able to move backwards on the already created elements of the tree. From the fact that a given parsing algorithm might progress backwards, it follows that the SSCCs must be able to indicate not only the creation of elements of the tree but also their destruction. The following definitions of operations and SSCCs are based on the ones used by Tunnel Grammar Studio (TGS) [17] to generate parsers, as the tool generates explicit builders that have their knowledge statically and use the facts listed previously. The implicit builders generated by the tool have no knowledge (type "none") and the information about the tree is dynamically received from the parser.

The following types of operations are distinguished below:

- **enter** – entrance into a container (rule, group, or concatenation) at the beginning of its recognition;
- **success** – exit from the end of a container after its successful recognition;
- **back** – return at the end of a container, with a function exactly opposite to *success*;
- **fail** – exit from the beginning of a container, with a function exactly opposite to *enter*.

Some SSCCs may have parameters. These parameters are optional and may contain facts (extracted from the parser grammar) or other information found during parsing (extracted from the input data). The facts can be sent from the parser to the builder as parameters of the SSCCs, when a fact is not known to the builder (or the parser does not know whether the fact is known to the builder). The different SSCCs are as follows:

1. **prepare** – a command for which the builder will create a new structure in the syntax tree according to the type of a particular grammar element:

- If the grammar element (for which an element of the tree will be built) is skippable or single, then the necessary structure is created in the tree that will be initialized with subsequent commands. The new structure depends on the value of a parameter indicating a token type or container type. The parameter might not be sent, when the builder knows what structure to create, and the parser knows that.

- When the element is represented as an array, each individual element in the array is prepared with this command in the same way as if the grammar element was skippable or single. For example, if the grammar element is a reference to a rule that has exactly five repetitions, then in the tree this element will be represented by an array of five structures. For each *prepare* command, the next unprepared structure in the array will be prepared with a new instance of the relevant structure;
- When a grammar element is represented as a list, each individual element in the list is prepared with this command, in the same way as the preparation of array elements (but represented as a list);

2. **unprepare** – with a function exactly opposite to *prepare*;
3. **rule enter**, **rule success**, **rule back**, **rule fail** – commands working differently according to the type of operation:

- *enter* – the builder will use an already prepared rule to continue building elements in it (the previous command was *prepare*);
- *success* – the builder will go out from the element in which it has built sub-elements so far, because the parsing of the current grammar rule was successful;
- *back* – the builder will go back into the element it built before (with a function exactly opposite to *success*);
- *fail* – the builder will go out from the element in which it has built sub-elements (with a function exactly opposite to *enter*, *unprepare* will follow);

4. **group enter**, **group success**, **group back**, **group fail** – analogically to the commands for rules;
5. **concatenation enter**, **concatenation success**, **concatenation back**, **concatenation fail** – commands working differently according to the type of operation and have a parameter that indicates the concatenation the command applies to:

- *enter* – the builder will create a structure in the syntax tree in which all future structures for the concatenation's elements will be created;
- *success* – exit from the structure in the tree after all grammar elements in the particular concatenation have been recognized;
- *back* – exactly opposite to *success*;
- *fail* – exactly opposite to *enter*;

6. **token forward** – a command with a parameter that contains the token itself that was recognized by the parser when it progresses forwards. The previous command was *prepare*. The data of the token will be stored in the already prepared structure in the tree. The prepared element in the tree can be placed in a list or in an array, depending on the repetitions of the element in the grammar;
7. **token backward** – a command with a parameter that contains the token itself that was recognized by the parser when it progresses backwards. The next command will be *unprepare*;
8. **next** – a command by which the parser informs the builder that it is moving on to the next grammar element. The subsequent commands will be for the next grammar element because the current grammar element was recognized (or at least its minimum number of repetitions). This command may not be sent when the grammar element has an exact number of repetitions and is clear that after the building of the required number of structures, the structures for the next grammar element will follow (which this command explicitly signifies). This must be known to the builder and the parser must know that the builder knows it;
9. **previous** – when the parser progresses backwards, with this command it informs the builder that the following commands will be for the previous grammar element in the parser's grammar. This command may not be sent in a similar way to the next command, given that the progress, in this case, is backwards;
10. **list create** – a command that instructs the builder to prepare to receive commands for tree structures that must be represented in the form of a list. The command has parameters for the minimum and maximum number of structures that might follow;
11. **list delete** – a command that instructs the builder to remove a prepared list;
12. **array create**, **array delete** – commands similar to list, but the future structures must be represented in the form of an array. The commands have a parameter indicating the exact number of structures that might follow;
13. **error** – a command that informs the builder of an error found from the parser in the input data. Backtracking might follow, which in turn can lead to the deletion of parts from the syntax tree.

4.4 Template grammars

To display detailed statistics for different parsers, a large number of grammars and a large number of input data are required. To simplify the manual writing of grammars and inputs that differ in small details (as the effect of the difference is measured according to various criteria is described later) it is necessary to create a specialized scripting language for imperative meta-programming of grammars, called **template grammar language**. The grammars developed in this language are called **template grammars**. Each template grammar is described in a textual format (**script**). It is absolutely mandatory that the language implementation is fully iterative because it is designed to work with large combinations of grammar elements. Iterative here means that no internal parsing or generation algorithms (both used by the language runtime) must contain a recursion that could overflow the thread dedicated stack. If that is not respected, this could complicate (or even prevent) the usage of some template grammars, because the process must run in a thread dedicated stack with excessive size. The iterative execution significantly relaxes (practically eliminating) this problem, because the used runtime memory becomes the whole available memory of the runtime environment. For this reason, the parsing of the template grammars is performed by a parser generated by TGS, because the algorithm (called tunnel parsing algorithm [9]), used by the generated parsers from the tool, is iterative at runtime.

The syntax of context-free grammar is usually expressed with a meta syntax like ABNF. Grammar in this syntax is directly defined (this means that no extra processing is needed to use the grammar). In contrast, template grammar describes one or more grammars (called **object grammars**) in a single compact form. For this reason, extra processing steps (called **evaluations**; explained later) are needed to compute each context-free object grammar from a given template grammar. The object grammars are context-free in this article, but any other grammar types could be used similarly.

Each template grammar has definitions of rules, tokens, templates, inputs, and ranges. The result of the execution of a template grammar script is a list of object grammars made of rules and tokens and a possibly empty list of input sequences. There are two template grammar features that influence the final object grammars:

- Range – a global constant that has a value from an explicitly defined sequence of unsigned integers. A template grammar has zero or more ranges. The evaluation of the template grammar is performed one time for each combination of values for each range. If there are no ranges then the evaluation is only one time.
- Templates – a feature of the grammar that allows the output from an evaluation of a template grammar to be generated by logical statements. The elements of the grammar where the particular template will be expanded are called **template instances**. An **expansion** means that the instance is removed and at its place, the output of the expansion is written directly. Each template has zero or more constant parameters and each template instance must have the same amount of arguments. The statements inside the template might use the grammar ranges and the template parameters by their name. The actual computation for the arithmetic operations between these constants (ranges and parameters) and directly given literals is performed at runtime, as the possible data types are a Unicode® [18] string and an unsigned integer. Note that even though from the perspective of the grammar the expansions might be recursive, they must be performed iteratively, as clarified up. Each expansion of a template is the same for the same combination of ranges and parameters (because they are all constants) and might be memoized for efficiency.

In this article two statements are used: the "if/then/else" statement that functions as in any other programming language, and the "out" statement that outputs strings in the place of the template instance. Because the statement "out" outputs raw strings, it is possible for one template grammar to output another that has more templates than itself. More ranges are also possible to be generated in the same way that will create more object grammars in turn. The possible inputs for each object grammar are also part of the template grammar. They prove the object grammars validity.

4.5 Parser Generator Profiler

Each evaluation of a template grammar's script, due to the expansion of the template instances, generates another script that has to be evaluated in turn. The previously generated grammar elements (rules, tokens, and templates) remain for each subsequent evaluation. The evaluations are performed

until there are no more expansions of template instances. The script without template instances (that will not be evaluated further) contains one object grammar (that consists of the computed till that time rules and tokens) and a sequence of inputs. This full process is shown in Fig. 1.

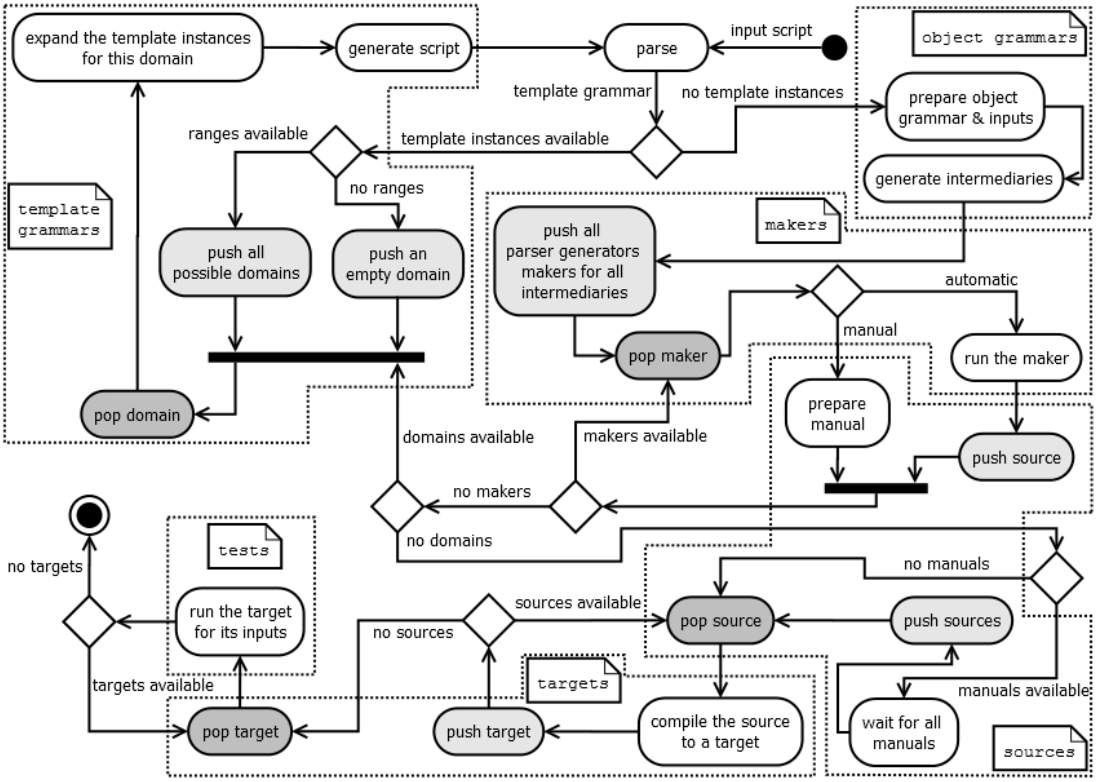

Figure 1: Parser Generator Profiler overall activity diagram

The Parser Generator Profiler (PGP) is a computer program that, by a given template grammar script, first generates a sequence of object grammars and a sequence of inputs. Then each object grammar is automatically translated to the meta syntaxes of different parser generators. These translated grammars are called the **profile intermediaries**. Each profile intermediary is given to a parser generator (manually or automatically) that generates source code files (**profile sources**) for a given programming language and compilation options (for example what kind of a syntax tree will be generated, if any). The profile sources are compiled with the respective language compilers with different compilation options (32/64 bits, level of code optimization, etc.). The compiled files are called **profile targets**. Then these profile targets are executed (several times in turn) for each input that was derived together with the particular object grammar that was used to create the particular profile target. Each execution of a profile target is called a **test**. After successfully completing all tests (that means that the template grammar was valid for all inputs) the visualization of the used **resources** (time and memory) is performed by the user. PGP (Fig. 1) uses four separated stacks: a) domain stack – with all ranges combinations per template grammar; b) maker stack – with the information for the object grammar to profile source conversion; c) source stack – with all of the ready profile sources; and d) target stack – with the information for all ready profile targets. The darker states in the figure are the push operations and the darkest states are the pop operations.

5 Conducted experiments

Four experiments were conducted. Each experiment was designed to measure the used resources (time and memory) by different parsers, generated by different parser generators for different object

grammars derived by different template grammars. Several variations of the compilation options for the profile targets were used: 32/64-bit modes (denoted as 4/8 respectively in the legends of the diagrams below) for C++ and Java™, and optimized/not optimized code (denoted respectively as Y/N in the legends). The different modes of the parsers are as follows:

- PGEN – parsing with a locator (the textual line and column information per token);
- PBLD – parsing without a locator (only measured for TGS);
- ASTG – parsing and generation of a generic abstract syntax tree ("generic" means the tree is the one the parser generator of the parser generates by default);
- ASTO – parsing and generation of an optimized abstract syntax tree ("optimized" here means that the tree nodes are automatically pruned at runtime; only measured for TGS);
- CSTG – parsing and generation of a generic concrete syntax tree;
- ASTV/CSTV – visitor design pattern parsing modes where the respective SSCC are generated but are not used for the building of a syntax tree. These are parsing modes of the TGS's implicit builder module, designed to allow one to build any kind of tree, based on the concrete/abstract syntax information alone, or directly use the SSCC for the specific task, without building anything.

All experiments are performed on Intel® Core™ i3-550 @3.2 GHz, Windows 10 64-bit. The profile target's compilers are Microsoft® C/C++ Optimizing Compiler Version 19.00.24210 and Oracle® Java™ SE Development Kit 1.8.0_301. The profile targets for Java™ are executed by Java™ SE Runtime Environment (build 1.8.0_301-b09). The profile source generators are TGS 1.0.65 [17], ANTLR (ANother Tool for Language Recognition) 4.8 [19], and JavaCC (Java Compiler Compiler) 7.0.5 [20]. The setup of the tools is as shown in their respective websites' primary examples. This means that by changing different options one might be able to influence (for better or worse) the results to some extent. These options were not explored, because the tested variations are already large in numbers. The used experimental methodology is: twelve measurements per input are made, the first two are discarded and the average of all remaining values is used. All input files are preloaded into the memory and these operations are not included in the measured resources. All-time measurements are from the wall clock.

The first experiment based on a template grammar (Fig. 2) is intended to measure the raw throughput abilities of the different parsers. The results are shown in Fig 3.

```
define L in 16, 32, 48, 64, 80, 96, 112, 128;
rule document = *zero;
token zero = "0";
template Q { out #char(34); };
template ZERO(N) { out "0"; if (N > 1 then out #ZERO(N - 1); ) };
input zeroes = #Q #ZERO(L * 1024) #Q;
```

Figure 2: The template grammar for the first experiment

The parser is as simple as it gets: it uses a parser grammar that accepts an infinite number of a single lexical token. We find this to be the best way to test the raw throughput of the parsers because the different parsers use different algorithms that handle the different grammar constructs differently, and the more complex the grammars are the more obscure it becomes, what performance on average, can be expected from other grammars. The purpose of the test is to give an empirical idea of the effectiveness of different parsers, irrelevant of the lexer and parser grammar complexities, not based on their descriptions, but on their practical implementations. The input's length varies linearly from 16 to 128 KiB.

Our interpretation of the results in the diagrams in Fig. 3 are as follows:

- The used memory for ASTG from the parsers generated by ANTLR is notably more for 64-bit C++, than the other parser variations of the same tool (Fig. 3a);

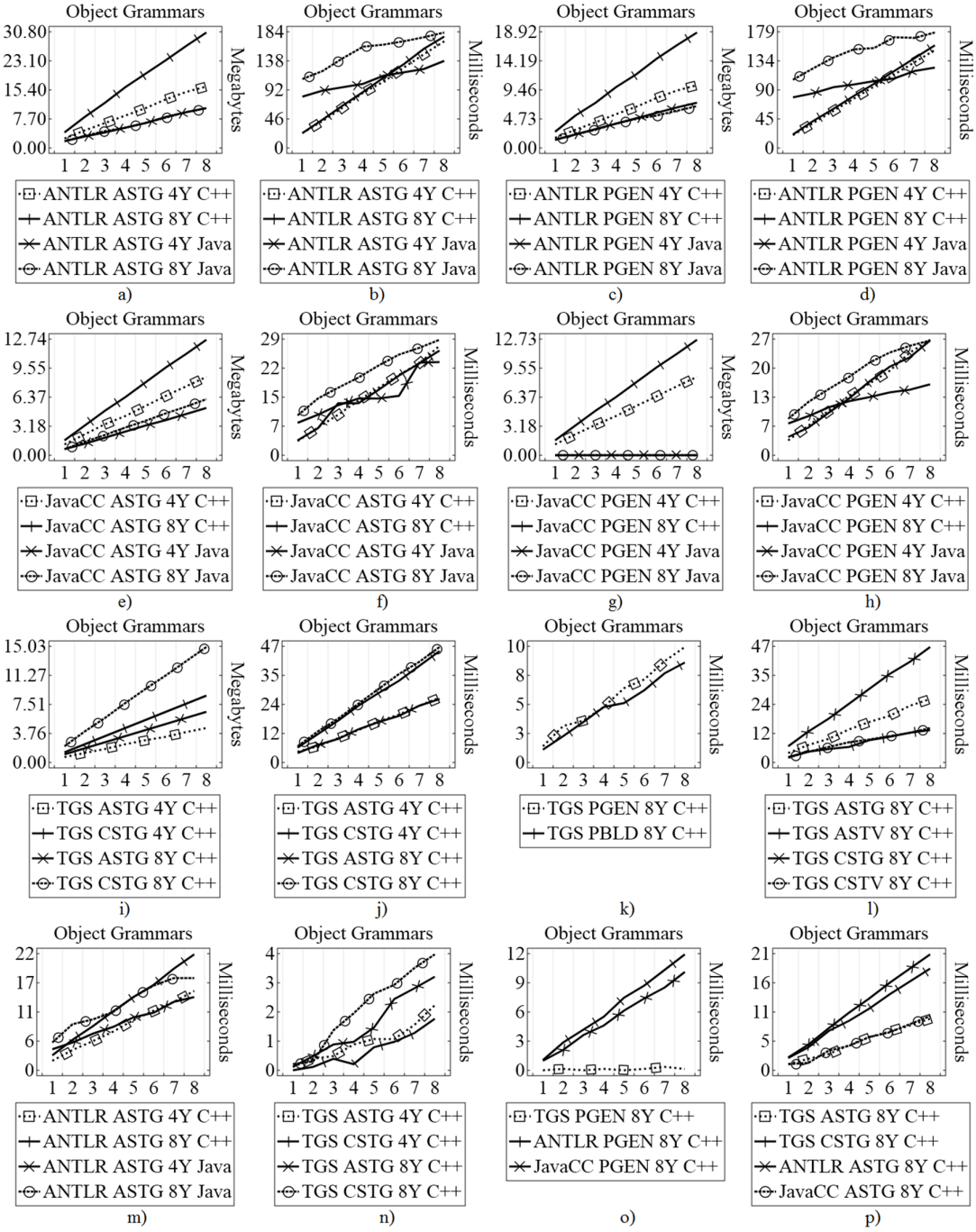

Figure 3: The results from the first experiment

- In the same modes, the 64-bit parser generated on Java™ is notably slower than the other parser variations, and with 32-bit on Java™ the speed eventually surpasses the speed of the C++ variations (Fig. 3b);
- When a syntax tree is not generated the ANTLR generated parsers on Java™ (Fig. 3c) are performing notably better (in terms of memory) than the ones on C++ (better memory results could be achieved with unbuffered tokens stream);
- In the same parser modes, in terms of speed for smaller inputs, the C++ variations are faster, but eventually, for longer inputs, the Java™ versions become faster (Fig. 3d);

- For JavaCC, the variations on Java™ are faster for longer inputs and take less memory (Fig. 3e and Fig. 3f);
- For JavaCC, in parsing mode, the used memory on Java™ (Fig. 3g) is not dependent on the input length (presumably, because it is garbage collected), but on C++ it is dependant (linearly);
- The Java™ variations surpass in speed the C++ ones for longer inputs (Fig. 3h);
- For TGS, the used resources for the CSTG are more than the ones for ASTG (Fig. 3i and Fig. 3j);
- Slightly faster parse time can be archived (Fig. 3k) if the locator is not calculated (mind the ordinate scale);
- The SSCC for an AST and a CST are generated for the same amount of time (Fig. 3l);
- About syntax trees traversing times – the 32-bit variations perform notably better than the 64-bit ones (Fig. 3m and Fig. 3n). The traversing is iterative for C++ (TGS and ANTLR) and recursive for Java (ANTLR);
- About parser destruction times – TGS "streams" the input by default, and for this reason, it has nothing to destroy (Fig. 3o);
- About parsers and syntax trees destruction times – the TGS and JavaCC parsers destroy their abstract trees equally faster. (Fig. 3p).

The second experiment uses a template grammar that will derive ten object grammars. Each object grammar has a sequence of rules that reference each other in the form of a list. The start rule (the first one) accepts an infinite number of the rule it references, and the last rule accepts a terminal symbol "0". In Fig. 4 "#string" is an instance of an internal to the language template that expands an unsigned integer argument to a string of its digits. The purpose of the experiment is to show how much resources are used for "depth" parsing and the building of different types of syntax trees with such a structure.

```
define L in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
template GEN(N)
{
  if (N > 0 then if (N == L
    then out "rule document = *";
    else out "rule sub_" #string(N) " ="; )
    out "sub_" #string(N - 1) ";" #char(13) #GEN(N - 1);
  else out 'rule sub_0 = zero;' #char(13); )
};
out #GEN(L);
template Q { out #char(34); };
template ZEROES(N) { out "0"; if (N > 1 then out #ZEROES(N - 1); ) };
token zero = "0";
input zeroes = #Q #ZEROES(128 * 1024) #Q;
```

Figure 4: The template grammar for the second experiment

The results of the tests performed for the second experiment are in Fig. 5. Our interpretations of the results are:

- The ANTLR generated parsers on C++, perform notably better (speed) in-depth parsing compared to the Java™ variations (Fig. 5a);
- The same parsers, use less memory (compared to each other) in C++ 32-bit, and more in C++ 64-bit modes (Fig. 5b);
- The JavaCC parsers perform nearly identical in all modes, except in the 64-bit Java™ one, where a large spike in the used time is observed (Fig. 5c);
- The same parsers, use a nearly identical amount of memory, with exception of the 64-bit C++ variant (Fig. 5d);
- The optimized abstract syntax trees use a constant amount of memory (not linear as all others) because the nodes in the tree that have only a single sub-node are optimized at runtime to a single node (thus the name, optimized) (Fig. 5e);

- The time to parse and build an ASTO compared to an ASTG are the same for all variations of the TGS parsers (Fig. 5f);
- The parsers generated by JavaCC are more optimization friendly to the C++ compiler, then the ANTLR generated parsers (Fig. 5g);
- The unoptimized parsers of TGS are faster than the unoptimized parsers generated by the other tools (Fig. 5h).

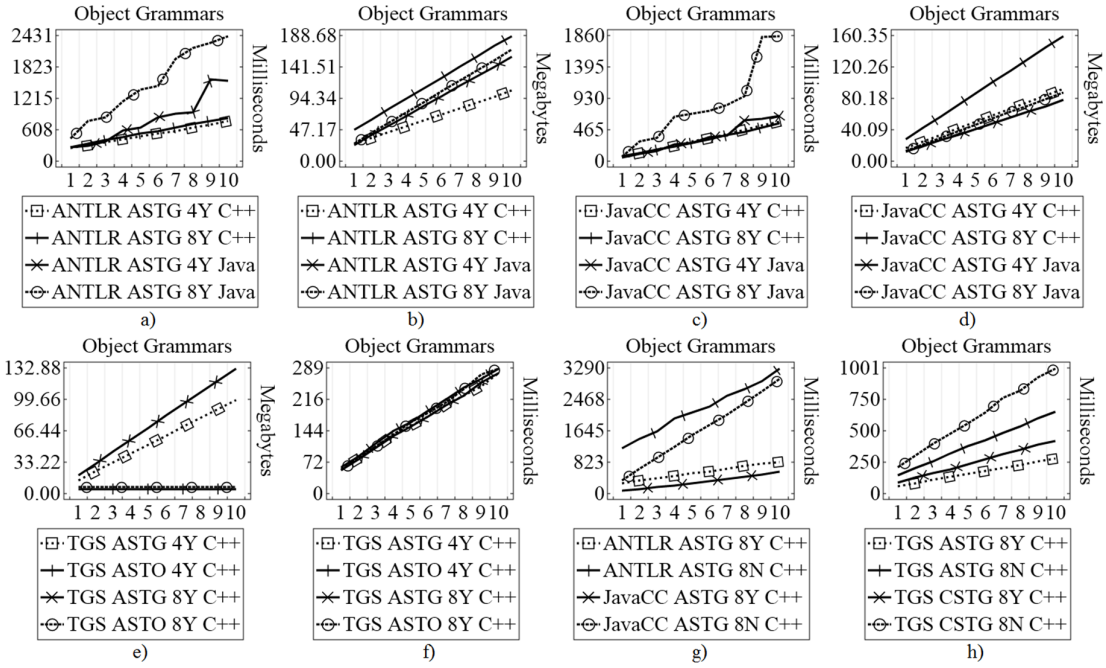

Figure 5: The results from the second experiment

The third experiment uses the template grammar in Fig. 6. The purpose of the experiment is to show the extent to which the number of elements in a concatenation affects the resources used.

```
define L in 5, 10, 15, 20, 25, 30;
template RULE(N) { out 'zero '; if (N > 1 then out #RULE(N - 1); ) };
template Q { out #char(34); };
template INPUT(N) { out "0"; if (N > 1 then out #INPUT(N - 1); ) };
rule document = *(#RULE(L));
token zero = "0";
input zeroes = #Q #INPUT(128 * 1024 / L * L) #Q;
```

Figure 6: The template grammar for the third experiment

Our interpretation of the experiment's results in Fig. 7 are:

- The same input data is stored in less memory (longer concatenations) (Fig. 7a), due to the higher fragmentation of the memory for smaller memory blocks (shorter concatenations). If that is properly utilized, one might store larger trees in less memory;
- This is only observed for the statically typed CST (as defined in this article) of TGS, and is not observed in this experiment in any other parser, because they all have dynamically typed AST (as defined in this article) and the fragmentation plays its role for all of them (Fig. 7b). **That is the expressive power of the statically typed concrete syntax trees generated by TGS. This means**

that one might benefit greatly in terms of memory by using these concrete syntax trees (see the next experiment also).

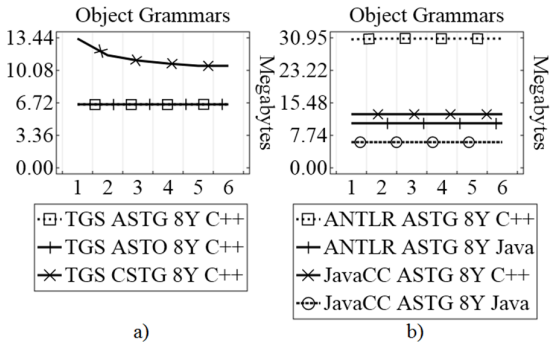

Figure 7: The results from the third experiment

The fourth experiment uses the template grammar in Fig. 8. The experiment's purpose is to show the used resources when the grammar has skippable elements that are never found in the input.

```
define L in 5, 10, 15, 20, 25, 30;
template ONE(N) { out '0*1 one '; if (N > 1 then out #ONE(N - 1); ) };
rule document = *(zero #ONE(L) zero);
token zero = "0";
token one  = "1";
template Q { out #char(34); };
template IZERO(N) { out "0"; if (N > 1 then out #IZERO(N - 1); ) };
input zeroes = #Q #IZERO(64 * 1024 / L * L * 2) #Q;
```

Figure 8: The template grammar for the fourth experiment

Our interpretations of the results (shown in Fig. 9) are:

- The ASTGs for the different parsers use the same amount of memory because they "lose" the information for the skippable (and never found) elements in the concatenation, but the CSTG of TGS represents everything explicitly, including the skippable elements. For this reason, some memory is used for all elements inside a concatenation that are not found in the input (Fig. 9a);
- The CSTG tree is still built quickly, slightly slower than the ASTG of JavaCC/TGS and all of them notably faster than the ASTG of the ANTLR generated parser (Fig. 9b).

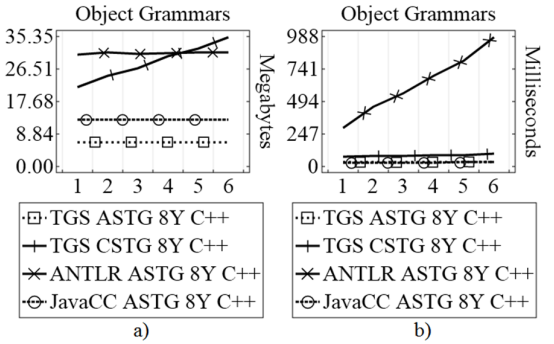

Figure 9: The results from the fourth experiment

6 Conclusion

The article presented a way to determine which tree is concrete and which is abstract. The different types of syntax trees (generated by the different parser generators) that have been studied are classified according to their abstractness, as defined in this article. The way to build a syntax tree is described by SSCCs. The comparison of the used resources (memory and time) between different types of syntax trees (generated by different parsers with different options) is done by using template grammars – a new way of expressing a large number of grammars in a compact form. The comparison results are based on different criteria such as the time for the tree building, its traversal time, its destruction time, and the memory used by the parser. Thanks to the tests, it becomes clear how "heavy" the different sequences of grammar elements are for the different parsers. This new information provides guidance on how a grammar developer can develop so that the resultant grammar can be parsed faster and with less memory.

A tool (Parser Generator Profiler – PGP) has been created that uses template grammars to perform the aforementioned tests. The mandatory iterative implementation of the template grammar language runtime transfers the limits of the runtime from the software side to the hardware side.

As a future development, SSCCs shall be supplemented with new commands so that graph syntax structures can be built, not just trees. The development of PGP shall be in the addition of other parser generators that will be compared with the existing ones. In this development, older versions of the various parser generators shall not be removed so that the tests can show the development of the technologies related to the automatic generation of parsers over time. This makes the PGP a future-oriented program. Another possible extension of PGP is to measure the compilation time of the profile sources and the profile targets, as well as the profile targets startup time and all related files sizes.

References

[1] A. Aho, M. Lam, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed., Addison Wesley, 2006.

[2] Abstract Syntax Tree Metamodel, 2011. https://www.omg.org/spec/ASTM/.

[3] A. J. Dos Reis, Compiler Construction Using Java, JavaCC, and YACC, Wiley, 2012.

[4] F. DeRemer , Practical translators for LR(k) languages, MIT, 1969.

[5] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction To Automata Theory, Languages and Computation, 3rd. ed., Pearson, 2013.

[6] M. Lange, H. Leiß, To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm, Informatica Didactica 8 (2009) 1-21.

[7] A. Johnstone, E. Scott, G. Economopoulos, Evaluating GLR parsing algorithms, Science of Computer Programming 61 (2006) 228–244. doi: 10.1016/j.scico.2006.04.004.

[8] D. Crocker, P. Overell, ABNF RFC 5234, 2008. URL: https://www.rfc-editor.org/info/rfc5234.

[9] N. Handzhiyski, E. Somova, Tunnel Parsing with countable repetitions, Computer Science 21.4 (2020) 441-462. doi: 10.7494/csci.2020.21.4.3753.

[10] N. Wirth, What can we do about the unnecessary diversity of notation for syntactic definitions?, Communications of the ACM 20.11 (1977) 822–823. doi: 10.1145/359863.359883.

[11] H. Moessenboeck, Coco/R A generator for fast compiler front-ends, ETH, Eidgenössische Technische Hochschule Zürich, Institut für Computer Systeme, vol. 127, 1990.

[12] J. Cervelle, M. Črepinšek, R. Forax, T. Kosar, M. Mernik, G. Roussel, On defining quality based grammar metrics, in: Proceedings of the 2009 International Multiconference on Computer Science and Information Technology, 2009, pp. 651-658. doi: 10.1109/IMCSIT.2009.5352768.

[13] B. Dean, Backlinko, 2019. URL: https://backlinko.com/page-speed-stats.

[14] X. Wu, Component-based language implementation with object-oriented syntax and aspect-oriented semantics, PhD thesis, The University of Alabama, BIRMINGHAM, USA, 2007.

[15] M. Forsberg, A. Ranta, BNF converter, in: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, Haskell '04, 2004, pp. 94–95. doi: 10.1145/1017472.1017475.

[16] N. Handzhiyski, E. Somova, A parsing machine architecture encapsulating different parsing approaches, Information Technologies and Security (IJITS) 13.3 (2021) 27-38.

[17] Tunnel Grammar Studio. https://www.experasoft.com/products/tgs/. Accessed: 2021-09-01.

[18] Unicode. URL: https://home.unicode.org/. Accessed: 2021-09-01.

[19] ANTLR. URL: https://www.antlr.org/. Accessed: 2021-09-01.

[20] JavaCC. URL: https://javacc.github.io/javacc/. Accessed: 2021-09-01.