SOFTWARE

“gnparser”: A powerful parser for scientific names based on parsing expression grammars

Dmitry Y. Mozzherin[1*†], Alexander A. Myltsev[2†] and David J. Patterson[3]

[*]Correspondence:
mozzheri@illinois.edu
[1]University of Illinois, Illinois Natural History Survey, Species File Group, 1816 South Oak St., Champaign, IL, 61820, US
Full list of author information is available at the end of the article
[†]Equal contributors

Abstract

Background: We are able to investigate biology on grander scales by integrating biological data from multiple sources. The use of scientific names of organisms allows aggregation of information on the same taxa located in many different places. There are impediments to such aggregation because there is often more than one name for a taxon, or one name may apply to more then one taxon. Names are often spelled with variations, sometimes misspelled, abbreviated, or annotated. Information about the author of a name often varies dramatically. To effectively match different scientific names for the same taxon to each other we want to parse them — to divide them into their elements and establish the roles of each element. We significantly improve the matching of different spellings for the same species by emphasizing the most widely used elements of names and then fine-tune matched results using other elements of names.

Results: We introduce Global Names Parser (*gnparser*), a Java tool written in Scala language (a language for Java Virtual Machine) to parse scientific names. It is based on a Parsing Expression Grammar. The parser can be applied to scientific names of any complexity. It assigns a semantic meaning (such as genus name, species epithet, rank, year of publication, names of authors, annotations, etc.) to all elements of a name. It is able to work with nested structures like for example in hybrid formulas. *gnparser* performs with $\approx 99\%$ accuracy and processes 30 million name-strings/hour per CPU thread. The *gnparser* library is compatible with Scala, Java, R, Jython, and JRuby. The parser can be used as a command line application, as a socket server, a web-app or as a RESTful http-service. It is released under an Open source MIT license.

Conclusions: Global Names Parser (*gnparser*) is a fast, high precision tool for bioinformaticians and biologists working with large numbers of scientific names. It can replace expensive and error-prone manual parsing and standardization of scientific names in many situations, and can quickly enhance the interoperability of distributed biological information.

Keywords: biodiversity; biodiversity informatics; scientific name; parser; semantic parser; names-based cyberinfrastructure

Notes

Addendum (Alex)

Conventions

Throughout the paper we distinguish “name”, “scientific name”, and “name-string”. “Name” refers to one or several words that act(s) as a label for a taxon. A “scientific name” is a name formed in compliance with a nomenclatural code (Code) or, if beyond the scope of the Codes, is consistent with the expectations of a Code. The term “name-string” is the sequence of characters (letters, numbers, punctuation, spaces, symbols) that forms the name. A name can be expressed by many name-strings (for example see Figure 1). There are millions of legitimately formed scientific names and probably billions of possible name-strings for them. We use the term “elements” for components of a name or name-string. Traditionally, scientific names for genera, and taxa below genus are presented in *italics*. In this paper, where we wish to emphasize examples of name-strings, we use **bold font**.

Background

Biology is entering a “Big Data” age, where global and fast access to all knowledge is envisaged. Progress towards this vision is still limited in scope. One impediment, especially for the long tail of smaller sources (of which some are not yet digital), is the absence of devices to inter-connect distributed data. The names of organisms are invaluable in “Big Data” biology because they can be treated as metadata that can be used to discover, index, organize, and interconnect distributed information about species and other taxa [1]. The use of names for informatics purposes is not straightforward because, for example, there may be many legitimate spellings for a name (Figure 1). A names-based infrastructure must recognize which name-strings represent the same scientific name.

Figure 1 illustrates that there is no single correct way to spell scientific names. Because of such variations, less than 15% of the names in comparisons of large biological databases could be matched based on exact spellings of name-strings [2]. In order to improve this simple metric for interoperability, we need to be able to identify such spelling variants as the same name, a process referred to as “lexical reconciliation”. Lexical reconciliation involves linking alternative spelling variants for the same taxon into a "lexical group". Most biologists do this intuitively — they recognize that the name-strings in Figure 1 refer to the same taxon. They do so by “parsing” the name-strings into elements (genus name, species name, authors, ranks etc.) and mentally discarding less significant elements such as annotations and authorship. It then becomes clear all of name-strings have a common group of latin elements **Carex scirpoidea convoluta**. We refer to the form of the name without authority or annotations as the “canonical form”. Further analysis of the name-strings reveals two different lexical groups (separated in Figure 1 by a line break) for, probably, one concept:

- **Carex scirpoidea var. convoluta** description by **Kükenthal**
- **Carex scirpoidea subsp. convoluta** rank determination by **Dunlop**.

In the past, the need to parse scientific names to form normalized names has mostly been achieved manually. A person familiar with rules of botanical nomenclature would be able to analyse the example of 24 name-strings with relative ease, but not thousands or millions of name-strings to which more than one nomenclatural code may be applied. The manual splitting of names into even only two parts — the latinized elements of taxon names that make up the canonical form and the authorship — is expensive, slow, and inflexible. To scale this exercise up requires an algorithmic solution, a scientific name parser!

The strategy of the algorithmic approach is to identify which combinations of the most atomic parts of a name-string (i.e. the UTF-8 encoded characters) represent words (such as genus name, species name, authors, annotations) or dates. An early algorithmic approach to parsing scientific names was with regular language implemented as regular expression [3]. A regular expression is a sequence of characters that describes a search pattern [4]. For example, a regular expression "[A-Z][a-z]{2}" recognizes a word that starts from a capital letter followed by two small letters (e.g. "Zoo"). Scientific names almost universally follow patterns such as the use of spaces to separate words, capitalization (of generic names and authors) or the inclusion of four digit dates between the middle of the 18th century and the present. This makes most names amenable to parsing by regular expressions. Examples of parsers based on regular expressions are GBIF's *name-parser* [5], and *YASMEEN* [6].

While regular expression is a powerful approach to string parsing, it has limitations. It cannot elegantly deal with name-strings where an authorship element is present in the middle of the name (for example **Carex scirpoidea Michx. subsp. convoluta (Kük.) D.A.Dunlop**). Indeed regular expressions are not well suited to any targets with recursive (nested) elements [7], such as hybrid formulae (e.g. **Brassica oleracea L. subsp. capitata (L.) DC. convar. fruticosa (Metzg.) Alef. × B. oleracea L. subsp. capitata (L.) var. costata DC.**). Name parsing built on regular expressions is impractical for complex name-strings.

Another limitation with most regular expression software tools is that they are "black boxes" that allow limited interaction with the parsing process, and do not reveal much information about the parsing context. Developers cannot call a procedure during a parsing event. As a result complex regular expression-based parsers are difficult to implement and maintain, and functions such as error recovery, detailed warnings, descriptions of errors are missing.

We wanted an approach able to deal with scientific names across a very broad range of complexity to give more flexibility than can be achieved with a regular expression approach. We believe that a general use parser should satisfy the following requirements.

1 **High Quality.** A parser should be able to break names into their semantic elements to the same standards that can be achieved by a trained nomenclaturalist or better. This will give users confidence in the automated process and allow them to set aside tedious and expensive manual parsing.

2 **Global Scope.** A parser should be able to parse all types of scientific names, inclusive of the most complex name-strings such as hybrid formulae, multi-infraspecific names, names with multilevel authorships and so on. No name-

strings should be left unparsed, otherwise information attached to them may remain undiscoverable.

3 **Parsing Completeness.** All information included in a name-string is important, not only the canonical form of the scientific name. Authorship, year, rank information allow us to distinguish homonyms, similar names, synonyms, spelling mistakes, or chresonyms. Access to such information improves the performance of subsequent reconciliation (the mapping of all alternative name-strings for the same taxon against each other).

4 **Speed.** Users, especially large-scale aggregators of biodiversity data, are more satisfied with quick responses, as it allows them to move forward to more purposeful value-adding tasks. Speed reduces the costs of building or running the hardware used for production parsing.

5 **Accessibility.** To be available to the widest possible audience, a parser should be released as a stand-alone program, have good documentation, be able to work as a library, to function as a command line tool, as a tool within a graphical interface, and to run as a socket or as RESTful services.

These requirements became our design goals. Based on our experience with prototype systems, we selected Parsing Expression Grammars and Scala language.

Adoption of Parsing Expression Grammars

Parsing Expression Grammars (PEG) [8] have been recently introduced for parsing strings. PEG allows developers to define the rules ("grammar") which describe general structure of target strings. In our case, such rules can be used to deconstruct scientific names. The rules are built from the ground up, starting from the simplest — such as a combination of "characters" separated by "spaces". That process will identify most "words". Characters can be distinguished as digits and other characters to make dates identifiable. Further rules can be applied, such as a "genus" rule would describe a part of a name-string in which the first word begins with combination of a "capital_character" followed by several "lower_case_characters" that fall within a relatively small spectrum of allowed characters; "authorship" would consists of one or more capitalized words and optionally a "year" within which, in some cases, authors may be grouped to form author-teams. PEG rules are designed to be recursive. They can be expanded to deal with increasingly complex name-strings, or address errors such as absent spaces, extra spaces, or OCR errors. Each rule can have programmatic logic attached, making the PEG approach very flexible. We believe that PEG suits our goals better than regular expressions for the following reasons:

- PEG is better suited than regular expressions for strings with a recursive structure;
- the syntax of scientific names is formal enough to be closer to an algebraic structure rather than to a natural language. Inconsistencies and ambiguities in scientific names are relatively rare due to compliance with the conventions of nomenclatural codes;
- scientific name-strings are short enough to avoid problems with computational complexity and memory consumption;
- programming a parser is easier because PEGs can describe parsing rules in a domain-specific language;

- domain specific language offers great flexibility for logic within the rules, for example to report errors in name-strings.

In 2008, as a part of the Global Names project, we created a specialized parsing library *biodiversity* [9] written in Ruby and based on PEG. We used an excellent *TreeTop* Ruby library [10] as the underlying PEG implementation.

The PEG approach allowed us to deal with complex scientific names gracefully. It gave us flexibility to incorporate edge cases and to detect common mistakes during the parsing process. The library *biodiversity* enjoyed considerable popularity. At the time of writing, it had been downloaded more than 150,000 times [11], it is used by many taxon name resolution projects (e.g. Encyclopedia of Life [12], Canadian Register of Marine Species (CARMS) [13], the iPlant TNRS [14], and World Registry of Marine Species (WoRMS) [15]. According to statistics compiled by BioRuby, *biodiversity*, at the time of writing, has been the most popular bio-library in the Ruby language [16].

We were pleased with PEG approach for scientific names parsing, but now regard the *biodiversity* parser library as a working prototype. It has allowed us to make further improvements and deliver a better, faster parser.

Adoption of Scala

The *biodiversity* package has performance and scalability issues because of the choice of Ruby as its programming language. Ruby is one of the best languages for rapid prototyping, but it is an interpreted dynamic language with, originally, a single-threaded runtime during execution. This makes it slow and inappropriate for rapid service or "Big Data" tasks. We determined that we needed a language environment with the following properties:

- a mature technology;
- multi-threaded, with high performance and scalability;
- an active support community with an Open source friendly culture;
- a wide range of libraries: utilities, web frameworks, etc.;
- a powerful development environment with IDEs, testing frameworks, debuggers, profilers and the like;
- mature libraries for search and cluster computations;
- interoperable with languages popular in scientific community (R, Python, Matlab);
- natural support of domain specific languages embedded in the hosted language.

While many of the properties are true for Ruby, other properties, such as high performance, scalability and interoperability, are lacking. To meet all requirements, and exploiting what we had learned from *biodiversity*, we refactored the code using the Java virtual machine, Scala programming language [17], and the Open source *parboiled2* library [18, 19] with our improvements [20]. The *parboiled2* library implements PEG in Scala[1].

The functional programming features of Scala allowed us to build a domain specific language that describes the rules of the grammars to parse scientific names. This produces a Parsing Expression Grammar with considerably more flexibility than

[1]An alternative to *parboiled2* is the Scala parser combinators library [21] but it has known problems with speed and memory consumption making *parboiled2* our preferred choice.

external lexers such as Bison or Yacc. As this domain specific language is within *parboiled2* it can take advantage of the Macro capacity of Scala [22] to optimize the compilation of the code and the subsequent running of the program. As a result the software performs with high efficiency. With this combination, the *gnparser* library achieved a significant boost in speed, scalability, and portability.

We limited this version to work with scientific names that comply with the botanical, zoological, and prokaryotic codes of nomenclature, but not with names of viruses because they are formed in different ways [23, 2] and need a different PEG that we intend to integrate later.

Implementation

The *gnparser* project is entirely written in Scala. It supports two major Scala versions: 2.10.6+ and 2.11.x. The code is organized into four modules:

1 “*parser*” is the core module used by all other modules. It parses scientific names from the most atomic components of a name-string to semantically-defined terms. It includes the parsing grammar, abstract syntax tree (AST) composed of the elements of scientific names, and warning and error facilities. When the parsing is complete and semantic elements of name-strings have been assigned to AST nodes, the elements can be recombined and formatted to meet further needs. For example:
   - *normalizer* can convert input name-strings into a consistent style;
   - *canonizer* creates canonical forms of the latinized elements of names; and with
   - *JSON renderer*, the parsing result is converted to JSON [24] to allow developers to work with the output from other languages. The output (Figure 2, also see Discussion) has the following information: **’details’** contains the JSON-representation of a parsed scientific name; **’quality_warnings’** describes potential problems if names are not well-formed; **’quality’** depicts a quality level of the parsed name; and **’positions’** maps the positions of every element in a parsed name to the semantic meaning of the element. Full and formal explanation of all parser fields is given as a JSON schema and can be found online [25] [also see additional file gnparser.json].

2 “*spark-python*” module contains facilities to use “*gnparser*” with Apache Spark scripts written in Python. Apache Spark is a highly distributive and scalable development environment for processing massive sets of data (“Big Data”). Spark is written in Scala, but can also be used with Python, R and Java languages. Spark programs written in Java and Scala are able to run “*parser*” in distributed fashion natively. The “*spark-python*” project makes such use possible for Python programs as well.

3 “*examples*” module contains examples to assist developers in adding “*parser*” functionality into other popular programming languages such as Java, Scala, Jython, JRuby, and R.

4 “*runner*” module contains the code that allows users to run “*parser*” from a command line as a standalone tool or to run it as a TCP/IP socket or HTTP web server. It depends on the “*parser*” module. The core part is the launch script “*gnparse*” (for Linux/Mac and Windows) that creates a JVM instance and runs “*parser*” on multiple threads against the input provided via a socket or

file. The “*runner*” also contains a web application and a RESTful interface as simpler methods to access “*parser*”. “*web*” achieves interactions with “*parser*” via HTTP protocol. It works both with simple web (HTML) and REST API interfaces. Figure 2 illustrates a parsing example using the web-interface. Socket and REST services use Akka framework which makes them highly concurrent and scalable.

“*parser*“ and “*examples*“ can run in JVM 1.6+. “*runner*” requires JVM 1.8+. Documentation is available in a README file [see additional file README.rst.html].

Parsing rules

gnparser v0.3.3 contains 76 PEG rules. These rules in turn make use of more elementary rules provided by *parboiled2* library. The rules distill knowledge received through many hours of conversations with leading taxonomists, studying nomenclatural codes, implementing feedback of the users. Lets follow a few of such rules as an example:

How many PEG rules did you create? Please provide a set of sample rules and discuss

A *yearNumber* rule detects a year when a name was published. *Rule[Year]* is a type of the returning value of the rule. Using domain-specific language and elementary rules of *parboiled2* we capture the start and the end positions of a year substring (lines #1 and #2). This capture matches a substring that is a common representation of a year in scientific name-strings. A publication year is usually a number between 1753 [26] and present. A year substring might have one or two digits substituted with question marks if the exact year of a publication is unknown. The capture is then passed as a parameter to a parser action (line #3). Parser action, which is a Scala function, might produce side effects (such as warnings) and returns a class instance of defined type (*Rule[Year]* in this case).

```
def yearNumber: Rule[Year] = rule { capturePos( // #1
    CharPredicate("12") ~ CharPredicate("0789") ~ \
      Digit ~ (Digit|'?') ~ '?'.? // #2
  ) ~> { (yPos: CapturePosition) => // #3
    FactoryAST.year(yPos) // #4
  }
}
```

Later we assemble more complex inter-dependent rules (lines #5 to #10). And finally the we combine all of them into the rule *year* on line #11 that consists of prioritized alternatives of all previously defined rules.

```
def yearWithChar = rule { yearNumber ~ capturePos(Alpha) } // #5
def yearWithParens = rule { '(' ~ (yearWithChar |\
  yearNumber) ~ ')' } // #6
def yearWithPage = rule { (yearWithChar | yearNumber) ~\
  ':' ~ oneOrMore(Digit) } // #7
def yearApprox = rule { '[' ~ yearNumber ~ ']' } // #8
def yearWithDot = rule { yearNumber ~ '.' } // #9
def yearRange = rule { yearNumber ~ '-' ~\
  capturePos(Digit.+) ~ (Alpha ++ "?").* } // #10
def year = rule { yearRange | yearApprox |\
  yearWithParens | yearWithPage | yearWithDot |\
  yearWithChar | yearNumber // #11
}
```

Now we can incorporate the *year* rule into all cases where it might be needed. For example on line #12 we indicate that *year* must be present in the matcher for the *authorsYear* rule.

```
def authorsYear: RuleNodeMeta[AuthorsGroup] = rule {
  authorsGroup ~ softSpace ~ (',' ~ softSpace).? ~ year ~> { // #12
    (aM: NodeMeta[AuthorsGroup], yM: NodeMeta[Year]) =>
      val a1 = for { a <- aM; y <- yM } yield a.copy(year = y.some)
      a1.changeWarningsRef((aM.node, a1.node))
  }
}
```

Installation

"*gnparser*" is available for launch in three bundles.

- A *parser* artifact is provided via the Maven central repository of Java code [27]. Physically it is a relatively small jar file without embedded external dependencies. The artifact can be accessed by a build system such as Maven, Gradle, or SBT in custom projects. The build system then identifies and provides access to all dependent jars.
- A Zip-archived "fat jar" is located at the project's GitHub repository. The jar contains the compiled files of *gnparser* along with all necessary dependencies to launch it within JVM. The archive is also bundled with a launch script (for Windows, OS X and Linux) that can run a command line interface to *gnparser*.
- The project's Docker container image is located at Docker Hub [28]. Docker provides an additional layer of abstraction and automation of operating-system-level virtualization on Linux. It can be thought as a lightweight virtualization technology within a Linux OS host. When it is setup properly, everything — starting from JVM and ending with Scala and SBT — can be run with simple commands that will, for example, pull the *gnparser*'s Docker image from the DockerHub, and run the socket or web server on an appropriate port.

Testing Methods

Editor: it's not very clear (to me or the reviewers, apparently) what the evaluation data actually was

Data for our tests were sets of 1,000 and 100,000 name-strings randomly chosen from 24 million unique name-strings of the Global Names Index (GNI) [29]. The name-strings in GNI are collected from a large variety of biodiversity data sources. Therefore the resulting datasets should represent data existing "in the wild" quite well. The datasets consist out of a randomly chosen mixture of well-formed names, names with formatting and spelling mistakes, and name-strings that were misrepresented as names. Name-strings in the sets are independent of each other. Evaluation datasets are included in the supplementary material.

We compared performance of *gnparser* with two other projects: *biodiversity* parser [30, 9] (also developed by Global Names team), and GBIF *name-parser* [5]. For comparison, we calculated $Precision$, $Recall$ and $Accuracy$ (as described below) using a dataset consisted of 1000 name-strings. Another project we considered was the YASMEEN parser from iMarine [6]. We found that with our dataset YASMEEN generated dramatically more mistakes than other parsers ($Precision$ 0.534, $Recall$ 1.0, $F1$ 0.6962), and was unable to finish a full dataset without crashing. We excluded it from further tests.

To estimate the quality of the parsers being compared, we used a combination of canonical forms and terminal authorships. A canonical form represents the scientific (latinised) elements of scientific names, while the terminal authorship refers to the author of the lowest subtaxon found in a scientific name. For example, with **Oriastrum lycopodioides Wedd. var. glabriusculum Reiche**, the canonical form is **Oriastrum lycopodioides glabriusculum** and the terminal authorship is **Reiche**, not **Wedd.**.

When both the canonical form and the terminal authorship were determined correctly we marked the result as true positive (N_{tp}). If one or both of them were determined incorrectly, the result was marked a false positive (N_{fp}). Name-strings correctly discarded from parsing were marked as true negatives (N_{tn}). False negatives (N_{fn}) were "suitable" name-strings which should be parsed, but were not. The following parameters where used for analysis:

$Accuracy$ — the proportion of correct results to all results. It is calculated as:

$$Accuracy = \frac{N_{tp} + N_{tn}}{N_{tp} + N_{tn} + N_{fp} + N_{fn}}$$

$Precision$ — the proportion of name-strings parsed correctly compared to all detected name-strings. It is is calculated as:

$$Precision = \frac{N_{tp}}{N_{tp} + N_{fp}}$$

$Recall$ — the proportion of correctly detected name-strings relative to all parseable name-strings and is calculated as:

$$Recall = \frac{N_{tp}}{N_{tp} + N_{fn}}$$

$F1 - measure$ is a balanced harmonic mean (where $Precision$ and $Recall$ have the same weight). When $Precision$ and $Recall$ differ, $F1 - measure$ allows results to be compared. It is calculated as

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Some names in the dataset were not well-formed. If a human could extract the canonical form and the terminal authorship from them, we included them. Examples of such name-strings are **"Hieracium nobile subsp. perclusum (Arv. -Touv.) O. Bolòs & Vigo"** (the issue here is an introduced space within an author's name),

“Campylium gollanii C. M?ller ex Vohra 1970 [1972]” (with a miscoded UTF-8 symbol and an additional year in square brackets), **"Myosorex muricauda (Miller, 1900)."** (with a period after the authorship).

Parsers analyze the structure of name-strings, but they cannot determine if a string is a “real” name. For example, in the case of a name-string that has the same form as a subspecies such as **"Example name Word var. something Capitalized Words, 1900"**. In such a case, the identification of a canonical form as **"Example name something"** and terminal authorship as **"Capitalized Words, 1900"** would be considered a true positive. Clearly, it will be important for name-management services to distinguish between name-strings of scientific names, names of viruses, surrogate names, and non-names. To find out how well parsers distinguished strings which are not scientific names, we calculated $Precision$ for discarded/non-parsed strings. If the parser has worked well, not-parsed strings would include only names of viruses and terms that do not comply with the Codes of zoological, prokaryotic, and botanical nomenclature.

We processed 100,000 name-strings with each parser. Each parser discarded close to 1000 name-strings as not-parseable. $Precision$ in this case showed percentage of correctly discarded names. We do not know $Recall$, as it was not reasonable to manually determine this for 100,000 names. To get a sense of names which should be discarded but were parsed instead, we analysed intersections and differences of the results between the three parsers. The following versions had been used for quality comparisons: *gnparser* v. 0.2.0, GBIF *name-parser* v. 0.1.0, *biodiversity* v. 3.4.1.

To establish the throughput of parsing we used a computer with an Intel i7-4930K CPU (6 cores, 12 threads, at 3.4 GHz), 64GB of memory, and 250GB Samsung 840 EVO SSD, running Ubuntu version 14.04. Throughput was determined by processing of 1,000,000 random name-strings from Global Names database.

To study the effects of parallel execution on throughput we used the *ParallelParser* class from *biodiversity* parser. We used ‘*gnparse file –simple*’ (a command line-based script set to return simplified output) for *gnparser*. For GBIF *name-parser*, we created a thin wrapper with multi-threaded capabilities [31]. The following versions had been used for throughput benchmarks: *gnparser* v. 0.3.1, GBIF *name-parser* v. 0.1.0, *biodiversity* v. 3.4.1.

Results and Discussion

We discuss and compare *gnparser*, GBIF *name-parser* and *biodiversity* parser in terms of our requirements for quality, global scope, parsing completeness, speed, and accessibility.

High Quality Parsing

Quality is the most important of the 5 requirements. GBIF *name-parser* uses regular expressions approach, while *gnparser* and *biodiversity* parsers use the PEG approach. Results for quality measurements are shown in Table 1.

If test data contain a large proportion of true negatives (N_{tn}) $Accuracy$ will not be a good measure as it favors algorithms which distinguish negative results, rather than finding positive ones. We manually checked our test datasets and established that $\approx 1\%$ were not scientific names. Given that true negatives are rare, they will have very limited influence on $Accuracy$. $Recall$ for all parsers was high, which means that false negatives are not important.

We hold that $Accuracy$ is probably the best measure for our tests. All 3 parsers performed very well, with $Accuracy$ values higher than 95%. Both *gnparser* and *biodiversity* parser approached 99% mark which we regard as production quality. Moreover, most of the false positives came from name-strings with mistakes. For example, out of 11 false positives (below) that *gnparser* found in the 1000 name-string test data set, only 2 (the first 2) were well-formed names.

Eucalyptus subser. Regulares Brooker
Jacquemontia spiciflora (Choisy) Hall. fil.

Acanthocephala declivis variety guianensis Osborn, 1904
Atysa (?) frontalis
Bumetopia (bumetopia) quadripunctata Breuning, 1950
Cyclotella kã¼tzingiana Thwaites
Elaphidion (romaleum) tæniatum Leconte, 1873
Hieracium nobile subsp. perclusum (Arv. -Touv.) O. Bolòs & Vigo
Leptomitus vitreus (Roth) Agardh{?}
Myosorex muricauda (Miller, 1900).
Papillaria amblyacis (M<81>ll.Hal.) A.Jaeger}

We do expect a parser to deal with names that are not well-formed. That means overcoming problems such as aberrant characters which might arise from Unicode character miscodings, inappropriate annotations, or other mistakes. To alert users, *gnparser* generates a warning when it identifies a problem in a name-string. The other parsers do not have this feature.

When parsers reach $\approx 80\%$ $Accuracy$, they hit a "long tail" of problems where each particular type of a problem is rare. Every new manual test against a new test set of 1,000–10,000 name-strings reveals new issues. Examples of these challenges are given elsewhere [2]. For all three parsers, developers had to perform the meticulous task of adding rules to address one rare case after another. That is, parsers need to be subject to continuous improvement. The problems found during preparation of this paper are being addressed in the next version of *gnparser*. As the parsing rules improve, we believe that *gnparser* can reach $> 99.5\%$ $Accuracy$ without diminishing $Recall$.

As we incorporate new rules to increase $Recall$, we have to consider the risks of reducing $Precision$ by introducing new false positives. For example, the GBIF *name-parser* allows the genus element of a name-string to start with a lowercase character. As a result the name-strings below were parsed as if they were scientific names, while other parsers ignored them:

acid mine drainage metagenome
agricultural soil bacterium CRS5639T18-1
agricultural soil bacterium SC-I-8
algal symbiont of Cladonia variegata MN075
alpha proteobacterium AP-24
anaerobic bacterium ANA No.5
anoxygenic photosynthetic bacterium G16
archaeon enrichment culture clone AOM-SR-A23
bacterium endosymbiont of Plateumaris fulvipes
bacterium enrichment culture DGGE band 61_3_FG_L
barley rhizosphere bacterium JJ-220
bovine rumen bacterium niuO17

Strategies like these might increase $Recall$ with certain low-quality datasets, but they decrease $Precision$. Many "dirty" datasets contain recurring problems. As an example, DRYAD contains many name-strings in which elements of scientific names are concatenated with an interpolated character such as '_' (e.g. "Homo_sapiens" and "Pinoyscincus_jagori_grandis") [2]. For them, our solution was to include a "preparser" script which "normalizes" known problems that are inherent within that dataset and then apply a high quality parser to the result.

Our testing also revealed differences between regular expressions and PEG approaches. Both can achieve high quality results with canonical forms of scientific names, but the regular expressions are less suitable for more complex name-strings. The reason is that the recursive or nested nature of some scientific names lead to greater problems which at some point become unsurmountable for regular expressions.

Global Scope

If we want to connect biological data using scientific names, no name-strings should be missed or rejected, no matter how complex they are. During our testing we found that $Accuracy$ of GBIF's *name-parser* was negatively affected because of how it managed (or did not manage) hybrid formulae and infrasubspecific names (names with more then one infraspecific epithet). This is where the regular expression approach reveals its limitations. For example the following names were not parsed by the GBIF *name-parser*:

Crataegus chlorosarca subtaxon pubescens E.L.Wolf
Erigeron peregrinus ssp.callianthemus var. eucallianthemus
Salvelinus fontinalis x Salmo gairdneri
Echinocereus fasciculatus var. bonkerae × E. fasciculatus var. fasciculatus

The PEG approach supports nested parsing rules to create progressively more complex rules that do manage such examples. Its capacity to address recursion allows *gnparser* to handle the full spectrum of scientific names that we have presented to it.

Table 1 Precision/Recall for parsers applied to 1000 name-strings

	gnparser	gbif-parser	biodiversity
True Positive	976	955	971
True Negative	13	12	13
False Positive	11	32	16
False Negative	0	1	0
Precision	0.9888551	0.967578	0.9837893
Recall	1.0	0.998954	1.0
F1	0.9943963	0.983016	0.9918284
Accuracy	0.989	0.967	0.984

Table 2 Precision for discarded by parsers names, out of 100 000 name-strings

	gnparser	gbif-parser	biodiversity
Total discarded	1131	1082	1161
True Positive	1129	940	1152
False Positive	2	142	9
Precision	0.998231	0.868761	0.9922481

Parsing Completeness

The extraction of canonical forms from name-strings representing scientific names is the most beneficial and widely used parsing goal. Sometimes, however, this may not be sufficient because the canonical form does not determine a name completely.

In the example in Figuire 1 **Carex scirpoidea convoluta** is a canonical form for **Carex scirpoidea var. convoluta Kükenthal** and **Carex scirpoidea ssp. convoluta (Kük.) Dunlop.** The first unparsed name-string refers to the variety **convoluta** of **Carex scirpoidea** that had been described by **Kükenthal**. The second captures Dunlop's reclassification of **convoluta** as a subspecies. We are not able to distinguish between these two different names without knowing the rank and/or the corresponding authorship. Furthermore, it is useful to see in the second example that **(Kük.)** was the original author and **Dunlop** was the author of the new combination.

After matching by canonical form, the ranks, authors, and "types" of authorship help users to distinguish name-strings with similar or identical canonical names from each other. The name-string **Carex scirpoidea Michx. var. convoluta Kükenth.** adds new information not evident in the examples in the paragraph above, that the species **Carex scirpoidea** was described by **Michx**.

Another area in which parsers with limited abilities can give misleading results is with negated names [2]. In these cases, the name-string includes some annotation or marks to indicate that the information associated with the name does NOT refer to the taxon with the scientific name that is included. Examples include **Gambierodiscus aff toxicus** or **Russula xerampelina-like sp**.

All components of a name may be important and need to be parsed and categorized. With *gnparser*, we describe the meaning of every word in the parsed name-string and present the results in JSON format. Parsing of **Carex scirpoidea Michx. subsp. convoluta (Kük.) D.A. Dunlop** gives the following JSON output

```
{
  "name_string_id" : "203213f3-99d1-5f5e-810a-4453c4d220cb",
  "parsed" : true, "quality" : 1, "parser_version" : "0.3.1",
  "verbatim" : "Carex scirpoidea Michx. subsp. convoluta (Kük.) D.A. Dunlop",
  "normalized" : "Carex scirpoidea Michx. ssp. convoluta (Kük.) D. A. Dunlop",
  "canonical_name" : {
    "value" : "Carex scirpoidea convoluta", "extended" : "Carex scirpoidea ssp. convoluta"
  },
  "hybrid" : false, "surrogate" : false, "virus" : false,
  "details" : [ {
    "genus" : { "value" : "Carex" },
    "specific_epithet" : {
      "value" : "scirpoidea",
      "authorship" : {
        "value" : "Michx.",
        "basionym_authorship" : { "authors" : [ "Michx." ] }
      }
    },
    "infraspecific_epithets" : [ {
      "value" : "convoluta", "rank" : "ssp.",
      "authorship" : {
        "value" : "(Kük.) D. A. Dunlop",
        "basionym_authorship" : { "authors" : [ "Kük." ] },
        "combination_authorship" : { "authors" : [ "D. A. Dunlop" ] }
      }
    } ]
  } ],
  "positions" : [ [ "genus", 0, 5 ], [ "specific_epithet", 6, 16 ], [ "author_word", 17, 23 ],
  [ "rank", 24, 30 ], [ "infraspecific_epithet", 31, 40 ], [ "author_word", 42, 46 ],
  [ "author_word", 48, 50 ], [ "author_word", 50, 52 ], [ "author_word", 53, 59 ] ]
}
```

The output includes the semantic meaning of all parsed elements in a name-string, indicates if the name-string was parsed successfully, if it is a virus name, a hybrid, or a surrogate. Surrogates are name-strings that are alternatives to names (such as acronyms) and they may or may not include part of a scientific or colloquial name (e.g. **Coleoptera sp. BOLD:AAV0432**). The output also includes a statement of the position of each element in the name-string. Last, but not least, the JSON output contains UUID version 5 calculated from the verbatim name-string. This UUID is guaranteed to be the same for the same name-string, promoting its use to globally connect information and annotations.

The output usually covers every semantic element in the name-string. The fields in the output illustrated above have the following meanings.

name_string_id: UUID v5 identifier;

parsed: whether a name-string was successfully parsed (true/false);

quality: how well-formed a name-string is (range from 1 to 3, 1 is the best);

parser_version: version of a parser used;

verbatim: name-string as was submitted to *gnparser*;

normalized: name-string modified by the parser to give a normalized style;

canonical_name: a special form of normalization that includes only the scientific elements of the name, this form is contained within most name-strings;

hybrid: whether the name-string refers to a hybrid (true/false);

surrogate: whether a name-string is a surrogate name (true/false);

details: describes the semantic elements within the name-string inclusive of the following;

genus: reports the genus part of the name (in this case Carex);

specific epithet: reports the species epithet (scirpoidea);

authorship: reports the authorship of the combination (Michx.);

basionym authorship: reports the authorship of the basionym (Michx.)

infraspecific epithets: reports the infraspecies name if present (convoluta) with rank (ssp.)

authorship: reports the authors of the infraspecies name ((Kük.) D. A. Dunlop)

basionym authorship: reports the author of the basionym of infraspecies name element (["Kük."]);

combination authorship: reports the author of the infraspecies name combination (D. A. Dunlop); and

positions: identifies each name element and where it starts and ends.

The complete list of fields for the *gnparser*'s output exists as a JSON Schema file [25].

Parsing Speed

In the areas of performance discussed above, there is little difference between *biodiversity* parser and *gnparser*. There is, however, a dramatic difference in their parsing speed and ability to scale. Parsing tasks that took 20 hours with earlier *biodiversity* parsers can now be completed in a few minutes on a multithreaded computer. Parsing is a key to other services such as name-reconciliation and subsequent resolution. Improving the parser will increase user satisfaction elsewhere.

Results on the speed performance are given in Figure 3. Depending on a number of CPU threads, *gnparser* had been $\approx 10-20\%$ faster than *gbif-parser*. On 1 thread *gnparser* was 7 times faster than *biodiversity*, 10 times faster on 4 threads, and 14 times faster on 12 threads.

gnparser displays functionality not presented in the GBIF *name-parser* as described in previous sections. In spite of this additional functionality *gnparser* outperformed other tested parsers.

Accessibility

By ‘accessibility’ we refer to the ability of the software code to be used by a wide audience. For Open source projects, accessibility is very important, because, when more people use a software, the more cost-effective is its development.

Parsing of scientific names is essential for organizing biodiversity data, such that many biodiversity database environments and projects include a parsing algorithm. Examples are uBio [32], the Botanical Society of Britain and Ireland [33], FAT [34], NetiNeti [35], and Taxonome [36]. A modular approach offers an option of re-use and avoids replication of effort. *biodiversity* was the first biodiversity parser to be released as a stand-alone package that could be used as a module — as it was with the iPlant project [30]. The same approach has now been adopted with the GBIF *name-parser* [5], *YASMEEN* [6], and *gnparser*.

We designed *gnparser* with accessibility in mind from the outset. Scala language allows the use of *gnparser* as a library in Scala, Java, Jython, JRuby and a variety of other languages based on Java Virtual Machine it can also be used natively in R and Python via JVM-binding libraries. If programmers want to use *gnparser* in some JVM-incompatible language they can connect to the parser via a socket server interface. There is also a command line tool, a web interface, and a RESTful API.

We pay close attention to documentation, trying to keep it detailed, clear, and up to date. We have an extensive test suite [see additional file test_data.txt] that describes the parser’s behavior and contains examples of *gnparser* functionality and output format.

This commitment to accessibility creates a larger potential audience for the parser, and will help many researchers and programmers to deal with the problems that arise from variant forms of scientific names.

Conclusions

The summary of results and discussion is depicted in Table 3.We can see that there is a significant similarity between two PEG-based parsers — *biodiversity* and *gnparser*. Both of them are based on the same algorithmic approach and follow similar design goals. We definitely could continue to modify rules for *biodiversity* and achieve the same or higher level of $Accuracy$ as we have now with *gnparser*. Instead we decided to create a completely new tool from scratch because we found that *biodiversity* is limited in speed, scalability and accessibility. For example at Global Names we often need to reparse 24 million name-strings, and the process was too slow for a production system. Also *gnparser* can be used natively by larger variety of programming languages than *biodiversity*, because JVM-based languages and tools are so widely spread. When we started to work on *gnparser* our first goal was the complete coverage of the *biodiversity*’s test suite. After we achieved the same level of quality the following tuning continued on *gnparser* alone, and *biodiversity* entered maintanance mode. That explains a slight difference in $Accuracy$ by these two parsers.

What is the difference between biodiversity and gnparser?

gbif-parser is a high quality product as well, however its regular-expressions-based algorithm limits its usability. Recursive nature of scientific names creates significant obstacles for intrinsically non-recursive algorithms such as regular-expressions. Coverage of multi-infraspecific names as well as hybrids, dealing with recursive patterns in authorship is prohibitively expensive for such approach.

Table 3 Summary comparison of Scientific Name Parsers

	gnparser	gbif-parser	biodiversity
Accuracy	98.9%	96.7%	98.4%
Hybrid formulas support	Yes	No	Yes
Infrasubspecies support	Yes	No	Yes
Throughput (names/s/thread)	8178	6389	1111
Parsing details	Complete	Partial	Complete
Library for the same language	Yes	Yes	Yes
Library for other languages	Yes	Yes	No
Command line tool	Yes	No	Yes
Socket server	Yes	No	Yes
Web Interface	Yes	Yes	Yes
RESTful service	Yes	Yes	Yes

In conclusion in this paper we describe *gnparser*, a powerful tool designed to break names of taxa into their semantic elements. This then allows standardization of names by transforming them into a canonical form, a step that in turn dramatically improves name matching and data integration. Parsing aids the discovery of names in sources, and sharing them in standardised forms (for example to create a common index). Parsing further allows users to extract, compare and analyse metadata within the name-strings, and allowing comparisons of the efforts of individuals or to map trends over time. The *gnparser* tool is released under MIT Open source license, contains command line executable, socket, web, and REST services, and is optimized for use as a library in languages like Scala, Java, R, Jython, JRuby.

Additional Files

Additional file 1 — gnparser.json

Full and formal explanation of all parser fields is given as a JSON schema.

Additional file 2 — test_data.txt

Extensive test suite that describes the parser's behavior. It is also a source of examples of parser functionality and output format. Test suite consists of a pipe delimited input (scientific name) and parsed output in JSON format.

Additional file 3 — README.rst.html

README.rst file that is converted to HTML format. It is also available at project home page [37].

Abbreviations

AAM	– Alexander A. Myltsev
API	– Application Program Interface
AST	– Abstract Syntax Tree
BHL	– Biodiversity Heritage Library
DJP	– David J. Patterson

DYM – Dmitry Y. Mozzherin
GBIF – Global Biological Informatics Facility
GNA – Global Names Architecture
JSON – JavaScript Object Notation
JVM – Java Virtual Machine
PEG – Parsing Expression Grammar
REST – Representational State Transfer

Acknowledgements

This work is supported by National Science Foundation under grant number NSF DBI-1356347. The authors thank administrators of the Species File Group for generous and much needed financial and logistic support during the transfer of the GNA grant from the Marine Biological Laboratory to University of Illinois.

Declarations

All authors have gone through the manuscript and contents of this article have not been published elsewhere.

Funding

We thank National Science Foundation for supporting this work via grant number NSF DBI-1356347. We also thank the Species File Group of the University of Illinois for additional funding.

Availability and Requirements

Project Name: gnparser
Project home page: https://github.com/GlobalNamesArchitecture/gnparser
Operating System: Any platform able to run JVM 1.8
Programming Language: Scala
License: The MIT License
Any restrictions to use by non-academic: no restriction

Availability of Data and Materials

The datasets supporting the conclusions of this article are available in the https://github.com/GlobalNamesArchitectu
paper/data repository.

Author's Contributions

DYM and AAM designed *gnparser*. DYM created requirements and test suite. AAM optimized *gnparser* for speed, refactored it into three internal subprojects. DYM set Docker containers and Kubernetes scripts. DYM and AAM wrote online documentation and JSON schema to formalize output. DJP corrected parser's results, calibrated quality output and errors output. DYM and AAM drafted manuscript and DJP edited its final version. All authors read and approved the final manuscript.

Competing Interests

The authors declare that they have no competing interests.

Concent for Publication

Not applicable

Ethics approval and consent to participate

Not applicable

Author details

[1]University of Illinois, Illinois Natural History Survey, Species File Group, 1816 South Oak St., Champaign, IL, 61820, US. [2]IP Myltsev, Kaslinskaya St., Chelyabinsk, , 454084, Russia. [3]University of Sydney, Sydney, Australia.

References

1. Patterson, D.J., Cooper, J., Kirk, P.M., Pyle, R.L., Remsen, D.P.: Names are key to the big new biology. Trends in Ecology and Evolution **25**(12), 686–691 (2010). doi:10.1016/j.tree.2010.09.004
2. Patterson, D., Mozzherin, D., Shorthouse, D., Thessen, A.: Challenges with using names to link digital biodiversity information. Biodiversity Data Journal **4**, 8080 (2016). doi:10.3897/BDJ.4.e8080
3. Leary, P.R., Remsen, D.P., Norton, C.N., Patterson, D.J., Sarkar, I.N.: uBioRSS: Tracking taxonomic literature using RSS. Bioinformatics **23**(11), 1434–1436 (2007). doi:10.1093/bioinformatics/btm109
4. Aho, A.V., Ullman, J.D.: Foundations of Computer Science vol. 2. Computer Science Press New York, ??? (1992)
5. GBIF name-parser. `https://github.com/gbif/name-parser/releases/tag/name-parser-2.10` Accessed 15 September 2016
6. Vanden Berghe, E., Coro, G., Bailly, N., Fiorellato, F., Aldemita, C., Ellenbroek, A., Pagano, P.: Retrieving taxa names from large biodiversity data collections using a flexible matching workflow. Ecological Informatics **28**, 29–41 (2015). doi:10.1016/j.ecoinf.2015.05.004
7. Yu, S.: Handbook of formal languages, regular languages (1997)
8. Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–122 (2004)
9. GlobalNamesArchitecture/biodiversity: Scientific Name Parser. `https://github.com/GlobalNamesArchitecture/biodiversity` Accessed 15 September 2016
10. Treetop. `http://treetop.rubyforge.org/` Accessed 15 September 2016
11. search | RubyGems.org | your community gem host. `https://rubygems.org/search?utf8=%E2%9C%93&query=biodiversity` Accessed 15 September 2016
12. Encyclopedia of Life. `http://eol.org/` Accessed 15 September 2016
13. Canadian Register of Marine Species. `http://www.marinespecies.org/carms/` Accessed 15 September 2016
14. iPlant Taxonomic Name Resolution Service. `http://tnrs.iplantcollaborative.org/` Accessed 15 September 2016
15. WoRMS - World Register of Marine Species. `http://www.marinespecies.org/` Accessed 15 September 2016
16. Ruby Libraries for Biology. `http://biogems.info/` Accessed 15 September 2016
17. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the scala programming language. Technical report (2004)
18. A macro-based PEG parser generator for Scala 2.10+. `https://github.com/GlobalNamesArchitecture/parboiled2` Accessed 15 September 2016
19. Myltsev, A., Doenitz, M.: parboiled2: Improved parsing expression grammars parsers generator in scala
20. GlobalNamesArchitecture/parboiled2: A macro-based PEG parser generator for Scala 2.10+. doi:10.5281/zenodo.50340. `https://github.com/GlobalNamesArchitecture/parboiled2` Accessed 15 September 2016
21. Moors, A., Piessens, F., Odersky, M.: Parser combinators in scala (2008)
22. Burmako, E.: Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In: Proceedings of the 4th Workshop on Scala. SCALA ’13, pp. 3–1310. ACM, New York, NY, USA (2013). doi:10.1145/2489837.2489840. http://doi.acm.org/10.1145/2489837.2489840
23. King, A.M.Q., Adams, M.J., Carstens, E.B., Lefkowitz, E.J. , E.: Virus Taxonomy: Classification and Nomenclature of Viruses: Ninth Report of the International Committee on Taxonomy of Viruses., pp. 1–1338 (2011)
24. Bray, T.: The javascript object notation (JSON) data interchange format (2014)
25. JSON schema for gnparser output. `http://globalnames.org/schemas/gnparser.json` Accessed 15 September 2016
26. Linne, C.V.: Plantarum: Exhibentes Plantas Rite Cognitas Ad Genera Relatas Cum Differentiis Specificis, Nominibus Trivialibus, Synonymis Selectis, Locis Natalibus Secundum, p. 583 (1753). http://books.google.com/books?hl=en&lr=&id=jQ0AAAAAQAAJ&oi=fnd&pg=PA1&dq=Species+plantarum:+exhibentes+plantas+rite+cognitas,-
27. Maven Central: Global Names Artifacts. `https://search.maven.org/#search|ga|1|globalnames` Accessed 15 September 2016
28. Global Names Parser Docker Image. `https://hub.docker.com/r/gnames/gnparser/` Accessed 15 September 2016
29. Global Names Index. `http://gni.globalnames.org` Accessed 15 September 2016
30. Boyle, B., Hopkins, N., Lu, Z., Raygoza Garay, J.A., Mozzherin, D., Rees, T., Matasci, N., Narro, M.L., Piel, W.H., McKay, S.J., Lowry, S., Freeland, C., Peet, R.K., Enquist, B.J.: The taxonomic name resolution service: an online tool for automated standardization of plant names. BMC bioinformatics **14**(1), 16 (2013). doi:10.1186/1471-2105-14-16
31. gbifparser: v0.1.0 (2015). doi:10.5281/zenodo.34848. `http://dx.doi.org/10.5281/zenodo.34848` Accessed 15 September 2016
32. uBio Name Parser. `http://www.ubio.org/tools/explode.php` Accessed 15 September 2016
33. Botanical Society of Britain and Ireland Taxon Name Parser. `http://bsbidb.org.uk/taxonnameparser.php` Accessed 15 September 2016
34. Sautter, G., Böhm, K., Agosti, D.: A combining approach to Find All taxon names (FAT) in legacy biosystematics literature. Biodiversity Informatics **3**, 46–58 (2006). doi:10.2307/1216144
35. Akella, L.M., Norton, C.N., Miller, H.: NetiNeti: discovery of scientific names from text using machine learning methods. BMC bioinformatics **13**(1), 211 (2012). doi:10.1186/1471-2105-13-211
36. Kluyver, T.A., Osborne, C.P.: Taxonome: a software package for linking biological species data. Ecology and Evolution **3**(5), 1262–1265 (2013). doi:10.1002/ece3.529
37. GlobalNamesArchitecture/gnparser: Split scientific names to meaningful elements with meta information. `https://github.com/GlobalNamesArchitecture/gnparser` Accessed 15 September 2016

38. Flora of North America Editorial Committee, E.: Flora of North America. Vol. 23, Magnoliophyta: Commelinidae (in Part): Cyperaceae, p. 551 (2002)
39. Global Names Parser Web App. `http://parser.globalnames.org` Accessed 15 September 2016

Figure 1 Some legitimate versions of the scientific name for the 'Northern Bulrush' or 'Singlespike Sedge'. The genus (*Carex*), species (*scirpoidea*), and subspecies (*convoluta*) may be annotated (var., subsp., and ssp.) or include or omit the name of the original authority for the infraspecies (Kükenthal), the species (Michaux), the current infraspecific combination (Dunlop), sometimes be abbreviated, differently spelled, and with or without initials and dates. This list is not complete. Image courtesy of [38].

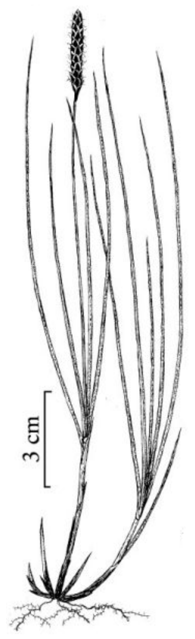
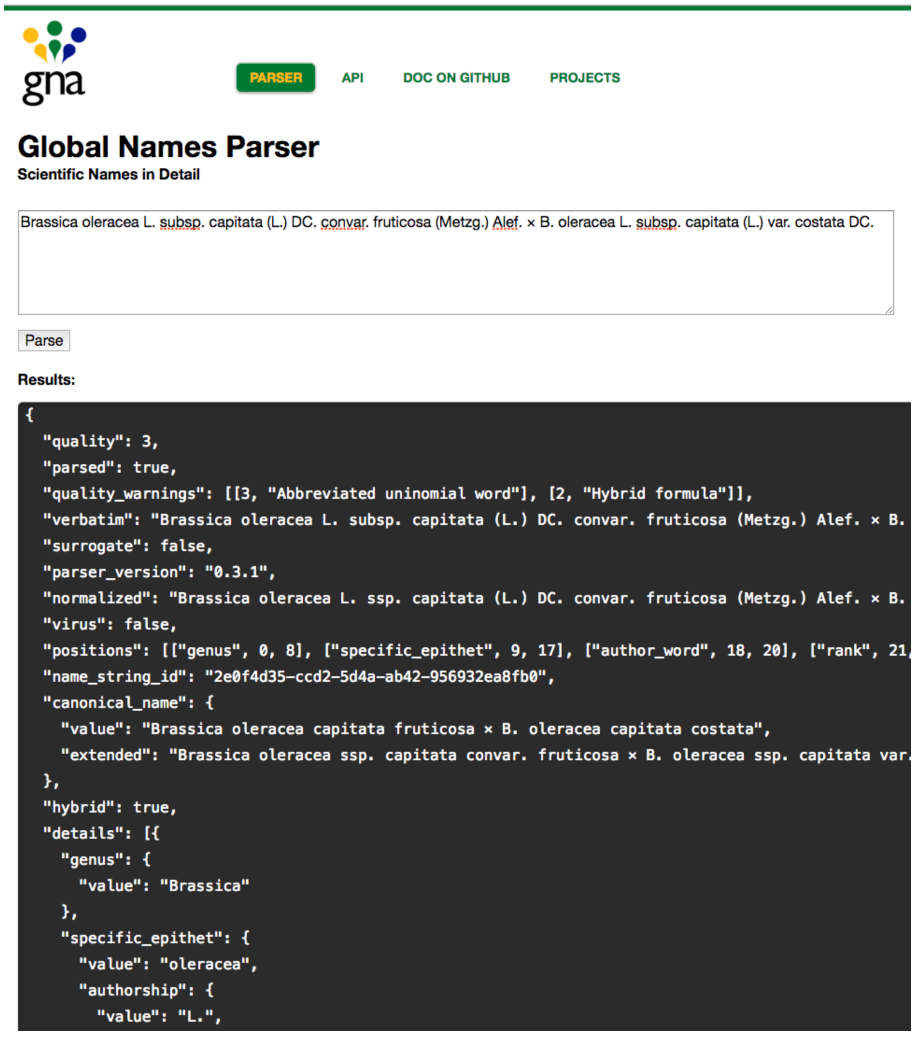
Carex scirpoidea convoluta **Carex scirpoidea var. convoluta** **Carex scirpoidea convoluta Kükenth.** **Carex scirpoidea var. convoluta Kuk.** **Carex scirpoidea var. convoluta Kük.** **Carex scirpoidea var. convoluta Kükenth.** **Carex scirpoidea var. convoluta Kükenthal** **Carex scirpoidea Michx. var. convoluta Kük.** **Carex scirpoidea Michx. var. convoluta Kükenth.** **Carex scirpoidea Michaux var. convoluta Kükenthal** **Carex scirpoidea subsp. convoluta** **Carex scirpoidea ssp. convoluta (Kük.) Dunlop** **Carex scirpoidea subsp. convoluta (Kük.) Dunlop** **Carex scirpoidea ssp. convoluta (Kukenth.) Dunlop** **Carex scirpoidea subsp. convoluta (Kük.) D.A.Dunlop** **Carex scirpoidea subsp. convoluta (Kük.) D.A. Dunlop** **Carex scirpoidea Michx. ssp. convoluta (Kük.) Dunlop** **Carex scirpoidea subsp. convoluta (Kuk.) D. A. Dunlop** **Carex scirpoidea Michx. subsp. convoluta (Kük.) Dunlop** **Carex scirpoidea Michx. ssp. convoluta (Kükenth.) Dunlop** **Carex scirpoidea subsp. convoluta (Kükenthal) D.A. Dunlop** **Carex scirpoidea Michx. subsp. convoluta (Kük.) D.A.Dunlop** **Carex scirpoidea Michx. subsp. convoluta (Kük.) D.A. Dunlop** **Carex scirpoidea subsp. convoluta (Kükenthal 1909) D.A. Dunlop 1998**	

Figure 2 Web Graphical User Interface [39]. In this example a user entered a name-string of a hybrid name consisted of 21 elements. The "Results" section contains detailed parsed output using compact JSON format.



Global Names Parser

Scientific Names in Detail

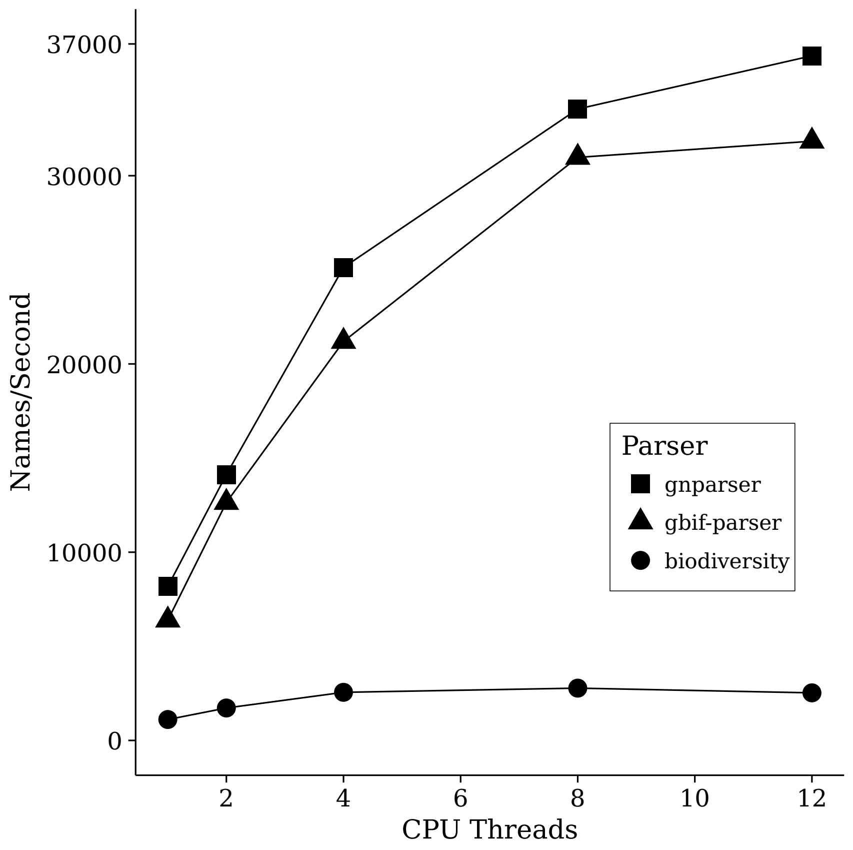
Brassica oleracea L. subsp. capitata (L.) DC. convar. fruticosa (Metzg.) Alef. × B. oleracea L. subsp. capitata (L.) var. costata DC.

Results:

```
{
  "quality": 3,
  "parsed": true,
  "quality_warnings": [[3, "Abbreviated uninomial word"], [2, "Hybrid formula"]],
  "verbatim": "Brassica oleracea L. subsp. capitata (L.) DC. convar. fruticosa (Metzg.) Alef. × B.
  "surrogate": false,
  "parser_version": "0.3.1",
  "normalized": "Brassica oleracea L. ssp. capitata (L.) DC. convar. fruticosa (Metzg.) Alef. × B.
  "virus": false,
  "positions": [["genus", 0, 8], ["specific_epithet", 9, 17], ["author_word", 18, 20], ["rank", 21,
  "name_string_id": "2e0f4d35-ccd2-5d4a-ab42-956932ea8fb0",
  "canonical_name": {
    "value": "Brassica oleracea capitata fruticosa × B. oleracea capitata costata",
    "extended": "Brassica oleracea ssp. capitata convar. fruticosa × B. oleracea ssp. capitata var.
  },
  "hybrid": true,
  "details": [{
    "genus": {
      "value": "Brassica"
    },
    "specific_epithet": {
      "value": "oleracea",
      "authorship": {
        "value": "L.",
```

Figure 3 Names parsed per second by GN, GBIF and Biodiversity parsers (running on 1–12 parallel threads).

Threads	gnparser	gbif-paser	biodiversity	Ratio		
				gn	gbif	bio
1	8178	6389	1111	1	0.78	0.14
2	14125	12638	1722	1	0.89	0.12
4	25125	21994	2556	1	0.88	0.10
8	33541	30972	2777	1	0.92	0.08
12	36369	31833	2527	1	0.88	0.07


Names/Second
CPU Threads
Parser
gnparser
gbif-parser
biodiversity