# A General Framework for Representing Controlled Natural Language Sentences and Translation to KR Formalisms

**Simone Caruso[1], Carmine Dodaro[2], Marco Maratea[2], Alice Tarzariol[3]**
[1]DIBRIS, University of Genoa
[2]DeMaCS, University of Calabria
[3]AICS, University of Klagenfurt
simone.caruso@edu.unige.it, carmine.dodaro@unical.it, marco.maratea@unical.it, alice.tarzariol@aau.at

## Abstract

Languages for Knowledge Representation and Reasoning, such as ASP, CP, and SMT, excel at solving complex problems, but encoding them into a higher-level language may be more profitable, leaving these formalisms as targets for solving. Recent studies aim to convert controlled natural languages (CNLs) into formal representations, yet these solutions are often tailored to specific languages and require significant effort. This paper introduces a general framework that generates grammars for target representation languages, enabling the translation of problems stated in CNL into formal representations. The related system, CNLWizard, offers a flexible, high-level approach to defining desired grammars, significantly reducing the time and effort needed to create custom grammars. Finally, we demonstrate the system's effectiveness through an experimental analysis.

## 1 Introduction

Answer Set Programming (ASP) [Lifschitz, 2019], Constraint Programming (CP) [Rossi *et al.*, 2006], and Satisfiability Modulo Theories (SMT) [Barrett *et al.*, 2021] are three powerful computational paradigms widely used for addressing complex combinatorial problems, each offering distinct approaches to problem modelling and solving. ASP is a form of declarative programming rooted in logic programming and non-monotonic reasoning. ASP solvers generate solutions, known as answer sets, that satisfy a given logic program. ASP is capable of dealing with problems requiring reasoning with incomplete or evolving information, such as knowledge representation, reasoning, and AI applications [Erdem *et al.*, 2016]. CP focuses on representing a problem through variables, domains, and constraints. The goal is to find assignments to variables that satisfy all constraints, making CP particularly effective for problems with intricate combinatorial structures, such as scheduling, planning, and resource allocation [Wallace, 1996; Hooker and van Hoeve, 2018]. SMT builds upon the foundation of Boolean Satisfiability (SAT) by extending it to more expressive theories like arithmetic, bit-vectors, and arrays [de Moura and Bjørner, 2011]. SMT solvers determine the satisfiability of logical formulas within these theories, enabling efficient handling of problems in verification, model checking, and software synthesis, where mathematical precision is crucial. They can be seen as complementary formalisms within the same broader framework of declarative problem-solving, where each formalism excels in different problem domains, yet they can be integrated or used in tandem to leverage their respective strengths.

These formalisms can benefit from the use of a controlled natural language (CNL), a subset of natural language with a restricted grammar and vocabulary designed to reduce ambiguity and complexity. Indeed, CNLs provide a bridge between formal methods and human-readable specifications, allowing domain experts to describe complex problems in a more intuitive and accessible manner without requiring deep expertise in the underlying formalism. As argued by Clark *et al.* [2005], and by Caruso *et al.* [2024], the benefits of using CNLs include *(i)* enhanced accessibility, as CNLs make it easier for non-experts to participate in problem formulation, as they can express constraints, rules, and requirements in a language close to the natural one; *(ii)* reduced errors by minimizing ambiguity, since CNLs help to prevent misinterpretations and errors in problem specifications, leading to more accurate solutions; *(iii)* improved collaboration, as CNLs facilitate communication between diverse teams, including domain experts and developers; and *(iv)* improved performance of natural language processing tools, as recently shown by Borroto Santana *et al.* [2024]. Integrating CNLs with ASP, CP, and SMT allows for a more natural and efficient approach to complex problem-solving, combining the precision of formal methods with the accessibility of natural language, broadening the applicability and impact of these powerful computational paradigms. Nevertheless, implementing custom CNLs is a time-consuming and challenging process. Indeed, from a technical perspective, creating a custom CNL involves writing the grammar, processing the Abstract Syntax Tree (AST), and generating code that translates the natural language input into the specific formalism. The first step requires defining a set of syntactic rules capable of capturing the nuances of the language while maintaining precision and clarity. The grammar must be comprehensive enough to cover the wide variety of constructs that users may wish to express, yet restricted enough to avoid ambiguity. Once the CNL input is parsed according to the grammar, it must be converted into an AST. This stage is complex because it

involves resolving ambiguities, managing scope, and ensuring compliance with all semantic rules of the CNL. The final step involves translating the AST into code that conforms to the specific formalism, whether it be ASP, CP, or SMT. This translation process is intricate, as it requires mapping high-level CNL constructs to the lower-level constructs of the formal language. In this paper, we present a novel framework that $(i)$ enables the user to specify a grammar for multiple target languages abstractly, $(ii)$ automatically suggests a possible default implementation for some elements involved in the pipeline, and $(iii)$ provides auxiliary data structures that make the construction of the CNL language more flexible and guided. We implemented the framework, obtaining a system called CNLWizard, designed to reduce the effort involved in developing a custom CNL. CNLWizard processes inputs described in a simple YAML-based language and automatically generates the grammar for the CNL, along with a set of pre-implemented imperative functions that minimise boilerplate code. This allows developers to focus primarily on writing the specific code required to convert sentences into the target formalism, streamlining the process and making custom CNL development more accessible and efficient. Indeed, as a practical evaluation, we compared the lines of code required by CNLWizard with those required by CNL2ASP [Caruso *et al.*, 2024] to define and convert CNL sentences into ASP rules, demonstrating that CNLWizard consistently requires significantly fewer lines of code than CNL2ASP.

## 2 Framework

We recall that a *grammar* [Chomsky, 1959] is a tuple $(N, T, S, P)$, where $N$ and $T$ are disjoint sets of non-terminal and terminal symbols, respectively, $S \in N$ is the starting symbol, and $P$ is a finite binary relation defined on $(N \cup T)^* \circ N \circ (N \cup T)^* \times (N \cup T)^*$, called *production rules*, where $*$ is the Kleene star operator [Hopcroft and Ullman, 1979]. According to the Chomsky Hierarchy, four types of grammar can be classified depending on the expressiveness allowed by $P$. For the purpose addressed in this work, we use *type 2* or *context-free grammars*, where $P$ is restricted to relations defined on $N \times (N \cup T)^*$. Let $F$ be a set containing KR formalisms. We represent with $fn : P, F \mapsto \mathcal{I}$ the function that maps a production rule $p \in P$ and a KR formalism $k \in F$ into an imperative function invoked when applying $p$ in the Abstract Syntax Tree (AST), where an element in $\mathcal{I}$ is a triple $(name, \vec{args}, code)$ containing, respectively, the function's name, arguments and implementation. In other words, the imperative function defined through *fn* specifies how CNL's text matching a production rule is "processed" to obtain statement(s) in a target KR formalism or to define the structure of the entities described.

Let $F_{Tar} \subset F$ represent a set of target KR formalisms for which a programmer aims to define a CNL grammar (and the corresponding *fn*). Our framework provides a high-level language to compactly define the CNL grammar $G_t = (N_t, T_t, S_t, P_t)$ for each formalism $t \in F_{Tar}$, together with an initialization of $fn(p, t)$ for each production rule $p \in P_t$. For the sake of simplicity, we will define how each set of production rule $P_t$ is built and assume that $N_t$ and $T_t$ are implicitly derived by projecting all the non-terminal and terminal symbols occurring in $P_t$. Without loss of generality, the starting symbol of each grammar, $S_t$, can be defined as a default non-terminal symbol, $S$.

Our framework consists of a function $\phi : \mathcal{C} \mapsto 2^{(P,\mathcal{I},F)}$ that maps a command $c \in \mathcal{C}$ into a set of triples defining a production rule and corresponding imperative function for a formalism. Given a set of commands $\hat{\mathcal{C}} \subseteq \mathcal{C}$, we define every $P_t$ for $t \in F_{Tar}$ as $P_t = \bigcup_{c \in \hat{\mathcal{C}}} \{p \mid (p, I, t) \in \phi(c)\}$. Then, for each $c \in \hat{\mathcal{C}}$ and each $(p, I, t) \in \phi(c)$, we initialize $fn(p, t) = I$. A command is a pair where the first argument is a keyword (such as **`syntax`**, **`concat`**, etc.) or a non-terminal symbol $N$ and the second argument can be any element of a grammar, formalisms or a set of commands. In the following, we are going to define the set of accepted command $\mathcal{C}$, together with the definition of $\phi$ for each of them.

A possible command is $(n, Par)$, where $n \in N$ and $Par$ is a set of commands with exactly one occurrence of:

- $(\textbf{`syntax`}, ST)$, where $ST \subseteq (N \cup T)^*$; and
- $(\textbf{`target`}, TG)$, where $TG \subseteq F_{Tar}$;

and at most one occurrence of the command:

- $(\textbf{`concat`}, CN)$, where $CN \in (N \cup T)$.

Then, the function $\phi((n, Par))$ returns:

$$\{(n \times ST, (n, ST \cap N, \emptyset), t) \mid t \in TG\} \cup \tag{1}$$

$$\{(n \times n \circ CN \circ n, \\ (n\_\textbf{`concat`}, (ST \cap N)^*, \texttt{code\_con}), t) \mid \\ t \in TG, (\textbf{`concat`}, CN) \in Par\}. \tag{2}$$

This command is used to define a production rule for each target formalism and automatically bind it with the corresponding imperative function, as shown in (1). Notice that the code part is left empty, since the implementation depends on the specific application. The optional command $(\textbf{`concat`}, CN)$ specifies an additional production rule for $n$, where its right-hand side consists of multiple applications of $ST$, concatenated by the symbol $CN$. The corresponding imperative function is named $n$_**`concat`** and has a single argument consisting of the list of non-terminal symbols that appeared in the rule, while its implementation is already provided by the framework and called `code_con`, as shown in (2).

A mandatory command is $(\textbf{`start`}, Par)$, where $Par$ is a set of commands defined as in the previous item. This command is used to specify the production rule of the starting symbol for each target formalism. Indeed, $\phi((\textbf{`start`}, Par))$ returns the sets in (1) and (2), where $n$ is replaced by the default starting symbol $S$.

Another possible command is $(\textbf{`operation`}, Par)$, where $Par$ is a set of commands with exactly one occurrence of:

- $(\textbf{`name`}, Op)$, where $Op \in N$;
- $(\textbf{`syntax`}, ST)$, where $ST \subseteq (N \cup T)^*$ and $Op$_**`oper`** $\in ST$;
- $(\textbf{`target`}, TG)$, where $TG \subseteq F_{Tar}$; and
- $(\textbf{`operators`}, (s, f_s))$, where $s \in T$ represents an operator for the operation $Op$, and $f_s$ defines how to translate $s$ when encountering its node in the AST;

and at most one occurrence of the command:

- $(\texttt{concat}, CN)$, where $CN \in (N \cup T)$.

Then, the function $\phi(\texttt{operation}, Par)$ returns:

$$\{(Op \times ST, (Op, ST \cap N, \texttt{code\_op}), t) \mid t \in TG\} \quad (3)$$

$$\cup \{(Op\_\texttt{oper} \times s, (Op\_\texttt{oper}, s, f_s), t) \mid t \in TG\} \quad (4)$$

$$\cup \{(Op \times Op \circ CN \circ Op, (Op\_\texttt{concat}, (ST \cap N)^*, \texttt{code\_con}, t) \mid t \in TG, (\texttt{concat}, CN) \in Par\}. \quad (5)$$

The command $(\texttt{operation}, Par)$ simplifies the definition of operations (e.g., mathematical, comparison, etc.), where the symbol $Op$ occurring with the keyword **name** represents a class of operations that includes all the operator's symbols defined after the keyword **operators**. The right-hand side of the production rule for $Op$, namely $ST$, must contain a special keyword, called $Op\_\texttt{oper}$, that indicates the position in the sentence of the string describing an operator. The framework then defines $\phi$ to return: a tuple for each $t \in TG$ containing the production rule of $Op$, where the implementation of the corresponding function is already provided by the framework and called `code_op`, as shown in (3); a tuple containing a production rule that maps $Op\_\texttt{oper}$ to each operation $s$, where the implementation of the corresponding function is defined by $f_s$, as shown in (4); and, lastly, the tuples obtained from the concatenation, as shown in (5).

Similarly to library usage for programming languages, our framework imports common patterns occurring for standard KR formalisms. Let $F_{Aux} \subset F$ represent a set of KR formalisms. For each $s \in F_{Aux}$ we assume the presence of an auxiliary context-free production rules $P^s_{Aux}$, where for each auxiliary rule $p \in P^s_{Aux}$ the function $fn(p, s)$ is pre-defined.

Another command is $(\texttt{import}, Par)$, where $Par$ is a set of commands with exactly one occurrence of:

- $(\texttt{source}, s)$, where $s \in F_{Aux}$; and
- $(\texttt{target}, t)$, where $t \in F_{Tar}$;

and one or more occurrences of the command:

- $(\texttt{rules}, name(p))$, where $name(p)$ is a label identifying a production rule $p$.

This command can be used to import auxiliary production rules and pre-defined imperative functions. Namely, $\phi((\texttt{import}, Par)) = \{(p, fn(p, s), t) \mid (\texttt{source}, s) \in Par, (\texttt{target}, t) \in Par, (\texttt{rules}, name(p)) \in Par, p \in P^s_{Aux}\}$.

Lastly, to extend the flexibility of the grammars, our framework provides a set of data structures characterizing elements that frequently occur in KR formalisms. By activating them, the user can access auxiliary production rules. In particular, we consider: *(i) Signatures* that provide a template to define concepts, which intuitively can be traced back to types in imperative programming languages; for example, they can be used to define the structure of an atom in ASP, a variable in CP, and a literal/variable in SAT and SMT. A signature $\sigma$ is composed of an entity, which includes a name and attributes; *(ii) Variables* that provide a template to define a variable that can range over different values. For example, one can state a variable $X$ is between 0 and 10, and then our framework is able to recognise that $X$ can range over the values and substitute it with its possible values in all of its occurrences.

## 2.1 Instantiation of the Framework

This section shows how the framework is instantiated for defining grammars that can parse CNLs tailored to the most common KR formalisms. $F_{Aux}$ is the set {ASP, CP, SMT} and $P_{Aux}$ is a set containing the following elements (for brevity, we show only the most important production rules), where non-terminal symbols are highlighted in bold and $\vee$ represents an or between elements, elements followed by ? are optional, **string** is a placeholder for a production rule matching any non-empty string containing letters, numbers and symbol _, and **number** is a placeholder for a production rule matching any number:

- (**signature**, (A $\vee$ An) **string** has (a $\vee$ an) **string** ((, $\vee$ and) (a $\vee$ an) **string**)*);
- (**variable**, where **string** is between **number** and **number**);
- (**there_is_clause**, (there $\vee$ There) is **entity**);
- (**positive_constraint**, It is required that **positive_constraint_body**);
- (**negative_constraint**, It is prohibited that **negative_constraint_body**);
- (**math**, (The $\vee$ the) **math_operator** between **math_first** and **math_second**);
- (**comparison**, **comparison_first** is? **comparison_operator comparison_second**);
- (**formula**, **formula_first formula_operator formula_second**);
- (**entity**, (a $\vee$ an)? **string attribute**?);
- (**verb**, (a $\vee$ an)? **string attribute**? **string**);
- (**attribute**, with **string** equal to (**string** $\vee$ **number**));
- (**simple_prop**, **entity** (have $\vee$ has $\vee$ are $\vee$ is) **verb entity**);
- (**neg_simple_prop**, **entity** (do not have $\vee$ does not have $\vee$ are not $\vee$ is not) **verb entity**);
- (**consequence**, If (**simple_prop** $\vee$ **neg_simple_prop**) then (**simple_prop** $\vee$ **neg_simple_prop**)).

These elements are provided by the framework and can be imported by users who want to extend the framework with new elements. In the example below, we demonstrate a simple extension of the framework.

**Example 1.** *In the following, we assume the user wants to create a CNL that contains only sentences of the form:*

- *An **entity** has an **attribute** and a **attribute**.*
- *There is a **entity** with **attribute** equal to **value**, and with **attribute** equal to **value**.*
- *When there is a **entity** with **attribute** equal to **value** and **attribute** equal to **value**, then there is a **entity** with **attribute** equal to **value** and **attribute** equal to **value**.*

*where **entity** is an element in the signature, **attribute** is part of the entity, and **value** is either a string or a number.*

*To this end, the user has to define the following set of commands, with ASP as a target:*

```
(import, {
    (source, asp), (target, asp),
    (rules, signature);
    (rules, there_is_clause)});
(start, {
    (syntax, signature), (target, {asp}),
    (concat, .)});
(start, {
    (syntax, there_is_clause),
    (target, {asp}), (concat, .)});
(start, {
    (syntax, rule),
    (target, {asp}), (concat, .)});
(rule, {
    (syntax, When there_is_clause, then
        there_is_clause), (target, {asp})}).
```

*The following CNL sentences are thus parsed by the resulting grammar:*

```
A user has an id and a name.
An admin has an id and a name.
There is a user with id equal to 1, and with
    name equal to john.
There is a user with id equal to 2, and with
    name equal to susan.
When there is a user with id equal to 2, and
    with name equal to susan, then there is
    an admin with id equal to 2, and with
    name equal to susan.
```

## 3 CNLWizard

The framework described in the previous section has been implemented as part of a practical tool called CNLWizard, which operates through two main computational steps. The former corresponds to the framework, while the latter is responsible for the actual translation of a CNL file into a target language representation. For the first step, our system expects a YAML file (see https://yaml.org/ for the syntax of YAML) specifying the set of commands $\hat{\mathcal{C}}$, where terminal symbols are identified as double-quoted strings and non-quoted strings are either non-terminal symbols or keywords; then, CNLWizard generates a grammar and a Python file containing the corresponding imperative functions, for each target language $t$ specified in $\hat{\mathcal{C}}$ through the keyword **`target`**. In the second step, CNLWizard applies the generated grammar to parse a CNL file, automatically producing an AST, and then it obtains the translated CNL by evaluating each node of the tree with the corresponding function generated in the previous step.

### 3.1 Step 1) Generate Target Grammar and Functions

Figure 1 depicts the first step of CNLWizard. The dotted arrow states that the user can optionally provide in input a file containing some of the auxiliary functions. If this is the case, instead of producing a new file containing all the function templates, the system just appends the missing functions.

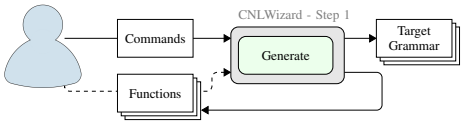


Figure 1: Generate Target Grammar and Functions.

In the following, we describe the YAML file expressing the set of commands $\hat{\mathcal{C}}$, where keywords are highlighted in bold, optional elements are enclosed in square brackets, and round brackets followed by the symbol plus indicate that multiple elements can be listed. The structure of the commands follows the template defined in the framework but relies on syntactic sugar that group commands with the same keywords, in order to further reduce the size of the file. The YAML file accepts commands of the form:

```
import:
    rules:  (rule_name)+
    source: (source_lang)+
    target: (target_lang)+
```

where each `rule_name` is a reserved word identifying an auxiliary production rule provided by CNLWizard, for each formalism listed in **`source`** (at the moment, ASP, CP, and SMT). The imported rule is added to the grammar of the languages specified by **`target`**. The lists in **`source`** and **`target`** must have the same length to have a pair-wise match that reduces the length of the specification for homonymous rules. For each non-terminal symbol defined by the user, `non_term_sym`, the following entry is expected:

```
non_term_sym:
    syntax:  (regex)+
   [target:  (target_lang)+]
   [concat:   conc_symbol]
```

where `regex` is a regular expression of terminal and non-terminal symbols, `target_lang` is the name of a KR formalism for which CNLWizard will generate the grammar and imperative functions, and `conc_symbol` is either a terminal or non-terminal symbol. If **`target`** is absent, then this rule is produced for every language specified with the starting symbol. Let us note that after **`syntax`** it is allowed to have a list of regular expressions. This is CNLWizard's syntactic sugar that compactly expresses groups of commands for the same non-terminal symbol. Additionally, CNLWizard allows unifying the definition of multiple rules, while being able to recall each of them in other regular expressions through the identifier `rule_name` specified after the keyword `name`:

```
non_term_sym:
  ( name:     rule_name
    syntax:  (regex)+
   [target:  (target_lang)+]
   [concat:   conc_symbol]
  )+
```

Then, we can have commands defining the starting symbol:

```
start:
    syntax:  regex
    target: (target_lang)+
   [concat:  conc_symbol]
```

where its structure matches the one used in the framework. The specification of the language operators is obtained with:

```
operation:
  ( name:       op_name
    operators: (sym : op_repr)+
    syntax:    (regex)+
   [target:    (target_lang)+]
   [concat:     conc_symbol]
  )+
```

This command is similar to the one for non-terminal symbols but additionally requires the presence of the non-terminal symbol `op_name_operator` in `regex`. After the keyword **`operators`**, it is expected a list of mappings from the operation name used in the CNL file, `sym`, to its representation in the target language, `op_repr`. In `op_repr`, it is possible to have either a terminal symbol or a function defined as:

```
fun:
    name:  function_name
    args: (argument)+
```

allowing for distinguishing the function name and list of arguments and making them easily accessible in the function implementation.

Lastly, together with the formalism for ASP, CP and SMT, CNLWizard provides further auxiliary grammar elements and relative implementation. Indeed, it provides the definition of the most common regular expressions, such as **`string`** and **`number`**, that can be used in the entries of **`syntax`**. Moreover, through the following statements:

```
non_term_aux:
  ( syntax:  regex
   [target: (lang)+]
   [concat:  token]
  )+
```

the system allows overwriting the syntax of the auxiliary non-terminal symbols appearing in the imported rules.

### 3.2 Example of YAML and Python Functions

In this section, we describe a use case to demonstrate the usefulness of CNLWizard. Specifically, let us assume that the user wants to solve the following (simplified) scheduling problem: Given three natural numbers (namely, $p_1$, $p_2$, and $b$), a set of teams, and a set of employees, where each employee has a salary, the goal is to assign employees to teams such that the following conditions are met: *(i)* each employee is assigned to exactly one team; *(ii)* each team has at least $p_1$ employees and at most $p_2$ employees; *(iii)* the total salaries of the employees on each team must not exceed the given budget $b$. Moreover, let us assume (s)he wants to use the following CNL to describe an instance of the problem with 3 employees, 2 teams, $p_1$, $p_2$, and $b$ set to 1, 2, and 5000, respectively:

```
An employee has a name and a salary.
A team has an id.
Assigned has an employee_name and a team_id.
There is an employee with name equal to 1,
    with salary equal to 2000.
There is an employee with name equal to 2,
    with salary equal to 3000.
There is an employee with name equal to 3,
    with salary equal to 2000.
There is a team with id equal to X, where X
    is between 1 and 2.
Employee with name equal to X is assigned to
    a team with id equal to 1 or with id
    equal to 2, where X is between 1 and 3.
The number of employees assigned to a team
    with id equal to X is greater than or
    equal to 1, where X is between 1 and 2.
The number of employees assigned to a team
    with id equal to X is less than 3, where
    X is between 1 and 2.
The sum between the salary of the employees
    that are assigned to a team with id equal
     to X is less than 5000, where X is
    between 1 and 2.
```

Listing 1: Example of simplified scheduling problem.

where sentences in lines 1–3 defines the concepts of employee, team, and assigned, respectively. Then, sentences in lines 4–6 define the employees and their salaries, and sentence at line 7 defines the two teams. Finally, sentences in lines 8–11 ensure that all conditions are met.

In this case, the YAML file reported in Listing 2 defines a set of commands for generating the grammar and the code needed by the aforementioned CNL, where we consider all the three different formalisms (ASP, CP, and SMT) as target.

```
import:
   rules: [attribute, there_is_clause, math,
       comparison,  simple_proposition,
       negated_simple_proposition, entity,
       verb, variable, signature]
   source: [asp, cp, smt]
   target: [asp, cp, smt]
start:
   syntax: (proposition ".")*
   target: [asp, cp, smt]
proposition:
 - name: disjunction
     syntax: simple_clause "or" attribute
     concat: ", and"
 - name: there_is_clause
 - name: comparison
simple_clause:
   syntax: [ simple_proposition,
       negated_simple_proposition]
comparison_first:
   syntax: [math, aggregate]
aggregate:
   syntax: '"The number of" entity verb
       entity'
comparison_second:
   syntax: number
math_first:
   syntax: '"the" string "of the" entity
       "that are" verb entity'
```

Listing 2: YAML Specification.

The specification begins with the keyword **`import`**, which adds to the grammar the auxiliary production

rules identified by `attribute`, `there_is_clause`, `math`, `comparison`, `simple_proposition`, `negated_simple_proposition`, `entity`, `verb`, `variable`, and `signature`. Then, the **`start`** keyword represents the root node of the AST, and its syntax is made of the concatenation of `proposition`. Here, the list of target formalisms, namely `asp`, `cp`, and `smt`, is also defined. Following, we define the syntax of `proposition`, which can be a `there_is_clause`, a `comparison`, or a `disjunction` (which also presents the `syntax` and the `concat` keywords). In the following, other rules are defined using the same format, nevertheless, notice that some non-terminal symbols, e.g. `comparison_first`, `comparison_second` and `math_first`, must be defined because of the imported rules, `comparison` and `math`.

Once the specification is defined, the tool generates the two files for each target language: the grammar, and the functions. An example of an unimplemented function generated by the tool is shown in Listing 3, where the function `start` raises a `NotImplementedError` meaning that it must be implemented by the user.

```
def start(propositions):
    raise NotImplementedError
```

Listing 3: Example of a generated unimplemented Python function.

Other rules, instead, as for example the **`operation`** rule, have a default implementation as shown in Listing 4. In this, case the `math` function returns the joined string of the operator with the operands.

```
def math(*args):
    operator_index = 0
    operator = args[operator_index]
    args = list(args)
    args.pop(operator_index)
    return operator.join(map(str, args))
```

Listing 4: Example of a generated implemented Python function.

After the generation step, the user should implement the functions with empty code, allowing CNLWizard to invoke them during the translation step. By providing the structure for each function, we facilitate the user's task of implementing how each parsed element is mapped into the corresponding expression in the target representation language. As the functions are defined in an imperative language, it is possible to rely on external libraries for the implementation and evaluation of each expression.

Moreover, a user can define a proposition that is valid only for one target formalism, e.g. ASP. This can be done by adding the following line under `proposition`:

```
- name: weak_constraint
      syntax: '"It is preferred as much as
          possible, that" comparison'
      target: [asp]
```

Listing 5: Addition to Listing 2 to target only one formalism.

which adds an optimization statement only for the target `asp`.

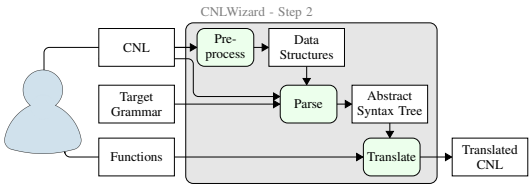


Figure 2: CNL translation step.

### 3.3 Step 2) Translate CNL into Target Representation

Once the target grammar(s) and corresponding functions have been generated and implemented by the user, we can proceed with the translation step, illustrated in Figure 2. First, CNLWizard pre-processes the input CNL. In this phase, it initializes the auxiliary data structures, i.e. signatures, if imported, and substitutes the variables. The signatures support the writing of the grammar and translation into the target formalism. As mentioned above, they define the concepts of the problem and their structure, and then one can easily recall them with their defined structure in the propositions. On the other hand, variable substitution allows the straightforward application of the same statement with different values. After the pre-processing, CNLWizard applies the generated grammar to parse the processed CNL and obtains an AST, which is evaluated according to the corresponding function, obtaining the corresponding CNL translated in the target representation. As an example, consider the sentence `There is an employee with name equal to 1, with salary equal to 2000.` reported in Section 3.2 and consider the ASP target. This sentence is automatically parsed by the generated grammar and the following Python function is automatically created:

```
def there_is_clause(entity):
    return Fact(entity)
```

Listing 6: Generated function for `there_is_clause` sentence.

where `Fact` is the name of a Python class implemented in CNLWizard which translates the `entity` in the ASP fact `employee(1, 2000).` Note that `entity` is also a data structure that is automatically created by CNLWizard and can be modified by accessing its internal fields, as follows:

```
def there_is_clause(entity):
    if 'salary' in entity.fields:
        entity.fields['salary'] += 'USD'
    return Fact(entity)
```

Listing 7: Modified function for `there_is_clause` sentence.

In this case, CNLWizard creates the ASP fact `employee (1, "2000USD").` All the examples and data for the target formalisms are available at https://github.com/dodaro/CNLWizard/tree/main/examples.

## 4 Implementation and Experiments

CNLWizard has been implemented in Python and released under an open-source licence [Caruso *et al.*, 2025]. The

| | CTS | GC | MAO | NSP | All |
|---|---|---|---|---|---|
| CNLWizard | 266 | 130 | 287 | 415 | 514 |
| CNL2ASP | 2620 | 2162 | 2422 | 2566 | 2936 |

Table 1: Comparison of the lines of code needed to define CNLs using CNLWizard and CNL2ASP.

target grammar is obtained and expressed using the library lark [Lark, ], while the file containing the templates for the functions is expressed in Python. For each target language, CNLWizard uses state-of-the-art tools, such as clingo [Gebser *et al.*, 2016] for ASP, OR-tools [Perron *et al.*, 2023] for CP, and Z3 [de Moura and Bjørner, 2008] for SMT. Moreover, it includes several error messages designed to identify misspelled text, and common user errors. As for the evaluation of the performance, as CNLWizard is a tool to define novel CNLs, we measured how it can help developers reduce their development time. This is often measured in Software Engineering as the number of lines of code needed to solve a problem (see, e.g. [Nguyen *et al.*, 2007] for a discussion about pros and cons). In our case, the problem is the generation of a grammar for a CNL and its translation to a KR formalism. Specifically, we compared the required lines of code (both grammar and python code) to implement the translations from a CNL into ASP using CNLWizard with the ones required by CNL2ASP, which is an open-source tool recently proposed by Caruso *et al.* [2024]. Concerning the problem specifications, we used the domains available from the CNL2ASP repository (available at https://github.com/dodaro/cnl2asp/tree/main/examples), namely Chemotherapy Treatment Scheduling (CTS), Graph Coloring (GC), Manipulation of Articulated Objects (MAO), and Nurse Scheduling (NSP). We also mention that CNL2ASP is a versatile tool with many constructs and functions and a total of about 6 thousand lines of code. As an example, it supports temporal operators and other constructs that are out of the scope of this paper. Therefore, for a fair evaluation of the lines of codes, for CNL2ASP we only considered the lines needed for parsing and translating the specific problem. The results are presented in Table 1, where the column labeled "All" indicates the total lines of code required to support the CNL sentences across all the considered domains. The main advantages of CNLWizard consist of a compact way of describing the grammar of the CNLs, a quick and easy way to import parts of grammar, and import/generate Python functions, and internal management of features such as signature, concatenation, and variable templates. As a result, CNLWizard consistently generated a corresponding CNL with significantly fewer lines of code, up to 10 times less than CNL2ASP.

## 5 Related Work

Many CNLs have been proposed for different tasks. Konrad and Cheng [2005] introduced a CNL which can be translated into various formalisms: linear time logic (LTL), computational tree logic, graphical interval logic, metric temporal logic, timed computational tree logic, and real-time graphical interval logic. This CNL was used in different practical applications in many domains. For example, Post *et al.* [2011] tested the applicability of this CNL in the automotive domain, while Filipovikj *et al.* [2017] used it to check industrial system requirements with SMT translations. Vuotto *et al.* [2019] and Narizzano *et al.* [2018] converted CNL constraints into LTL formulae and Mahmud *et al.* [2016] employed a controlled language to structure automotive embedded systems specifications in a natural language, which is then translated into Boolean expressions to check their consistency using Z3.

In logic programming, the first approaches were proposed by Fuchs and Schwitter [1995] and Schwitter *et al.* [1995], resulting in Attempto CNL [Fuchs, 2005], whose idea was to convert sentences expressed in a CNL as Prolog clauses. Clark *et al.* [2005] presented a computer-processable language designed to be more accessible for computers than for human users. Erdem and Yeniterzi [2009] proposed BIOQUERYCNL, a CNL for biomedical queries, and developed an algorithm designed to automatically encode a biomedical query expressed in this language as an ASP program. BIOQUERYCNL is a subset of Attempto CNL that can represent queries, and it was also used as a basis to generate explanations of complex queries [Öztok and Erdem, 2011]. Baral and Dzifcak [2012] proposed a CNL specific for solving logic puzzles, whose sentences are then converted into ASP. Lifschitz [2022] showed the process of translating one English sentence into ASP code. Caruso *et al.* [2024] proposed a CNL (similar to the one proposed by [Schwitter, 2018]) which is then converted into ASP by the tool CNL2ASP.

Another line of related work concerns LLMs, even if they differ fundamentally from CNLs. Indeed, while LLMs show generally good performance, they are currently unreliable in producing correct ASP, CP, or SMT code due to their probabilistic nature. CNLs, instead, ensure precision through grammar-based determinism. Nevertheless, CNLs can be used for improving the performance of LLMs, as shown by Borroto Santana *et al.* [2024], where CNLs were used as an intermediate layer to map NL to ASP. Our tool could be used to define new CNLs, which in turn can serve as targets for converting NL sentences using their method. Other recent works on LLMs and KR formalisms address different goals: Alviano and Grillo [2024] focuses on using LLMs to generate ASP facts; Coppolillo *et al.* [2024] convert simple patterns in ASP code, but they do not generalize on complex encodings; while Wang *et al.* [2024] and Yang *et al.* [2024] use KR formalisms to improve LLMs performance.

## 6 Conclusions

In this paper, we introduced a novel framework and its implementation, CNLWizard, which simplifies the creation of CNLs for KR formalisms such as ASP, CP, and SMT. The framework also facilitates the development of CNLs for other languages by automating much of the CNL creation process. Using a YAML-based input language, CNLWizard streamlines grammar generation and the creation of imperative functions, reducing the need for extensive boilerplate code.

For future work, the framework could be extended to natively support additional languages and enable users to share their custom CNLs with the broader community. Moreover, CNLWizard can be extended to prevent misspelled CNL text.

## Acknowledgments

Carmine Dodaro and Marco Maratea were supported by the European Union - NextGenerationEU and by Italian Ministry of Research (MUR) under PNRR project FAIR "Future AI Research", CUP H23C22000860006 and by the European Union - NextGenerationEU and by the Ministry of University and Research (MUR), National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.5, project "RAISE - Robotics and AI for Socio-economic Empowerment" (ECS00000035) under the project "Gestione e Ottimizzazione di Risorse Ospedaliere attraverso Analisi Dati, Logic Programming e Digital Twin (GOLD)", CUP H53C24000400006. Carmine Dodaro was supported by the European Union - NextGenerationEU and by Italian Ministry of Research (MUR) under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006. This research was funded in part by the Austrian Science Fund (FWF) 10.55776/COE12.

## References

[Alviano and Grillo, 2024] Mario Alviano and Lorenzo Grillo. Answer set programming and large language models interaction with YAML: preliminary report. In *Proc. of CILC*, volume 3733 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2024.

[Baral and Dzifcak, 2012] Chitta Baral and Juraj Dzifcak. Solving puzzles described in english by automated translation to answer set programming and learning how to do that translation. In *Proc. of KR*. AAAI Press, 2012.

[Barrett *et al.*, 2021] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability - Second Edition*, volume 336 of *FAIA*, pages 1267–1329. IOS Press, 2021.

[Borroto Santana *et al.*, 2024] Manuel Borroto Santana, Irfan Kareem, and Francesco Ricca. Towards automatic composition of asp programs from natural language specifications. In *Proc. of IJCAI*, 2024.

[Caruso *et al.*, 2024] Simone Caruso, Carmine Dodaro, Marco Maratea, Marco Mochi, and Francesco Riccio. CNL2ASP: converting controlled natural language sentences into ASP. *Theory Pract. Log. Program.*, 24(2):196–226, 2024.

[Caruso *et al.*, 2025] Simone Caruso, Carmine Dodaro, Marco Maratea, and Alice Tarzariol. Github repository of CNLWizard. MIT Licence, available at https://github.com/dodaro/CNLWizard, 2025.

[Chomsky, 1959] Noam Chomsky. On certain formal properties of grammars. *Inf. Control.*, 2(2):137–167, 1959.

[Clark *et al.*, 2005] Peter Clark, Philip Harrison, Thomas Jenkins, John A. Thompson, and Richard H. Wojcik. Acquiring and using world knowledge using a restricted subset of english. In *Proc. of FLAIRS*, pages 506–511. AAAI Press, 2005.

[Coppolillo *et al.*, 2024] Erica Coppolillo, Francesco Calimeri, Giuseppe Manco, Simona Perri, and Francesco Ricca. LLASP: fine-tuning large language models for answer set programming. In *Proc. of KR*, 2024.

[de Moura and Bjørner, 2008] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[de Moura and Bjørner, 2011] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[Erdem and Yeniterzi, 2009] Esra Erdem and Reyyan Yeniterzi. Transforming controlled natural language biomedical queries into answer set programs. In *Proc. of the BioNLP Workshop*, pages 117–124. Association for Computational Linguistics, 2009.

[Erdem *et al.*, 2016] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Mag.*, 37(3):53–68, 2016.

[Filipovikj *et al.*, 2017] Predrag Filipovikj, Guillermo Rodríguez-Navas, Mattias Nyberg, and Cristina Seceleanu. Smt-based consistency analysis of industrial systems requirements. In *Proc. of SAC*, pages 1272–1279. ACM, 2017.

[Fuchs and Schwitter, 1995] Norbert E. Fuchs and Rolf Schwitter. Specifying logic programs in controlled natural language. *CoRR*, abs/cmp-lg/9507009, 1995.

[Fuchs, 2005] Norbert E. Fuchs. Knowledge representation and reasoning in (controlled) natural language. In *Proc. of ICCS*, volume 3596 of *LNCS*, pages 51–51. Springer, 2005.

[Gebser *et al.*, 2016] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *Proc. of ICLP TCs*, volume 52 of *OASIcs*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[Hooker and van Hoeve, 2018] John N. Hooker and Willem Jan van Hoeve. Constraint programming and operations research. *Constraints An Int. J.*, 23(2):172–195, 2018.

[Hopcroft and Ullman, 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Konrad and Cheng, 2005] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proc. of (ICSE)*, pages 372–381. ACM, 2005.

[Lark, ] Lark repository. URL: https://github.com/lark-parser/lark.

[Lifschitz, 2019] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[Lifschitz, 2022] Vladimir Lifschitz. Translating definitions into the language of logic programming: A case study.

In *Proc. of the ICLP Workshops*, volume 3193 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[Mahmud *et al.*, 2016] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. Resa tool: Structured requirements specification and sat-based consistency-checking. In *Proc. of FedCSIS*, volume 8 of *Annals of Computer Science and Information Systems*, pages 1737–1746. IEEE, 2016.

[Narizzano *et al.*, 2018] Massimo Narizzano, Luca Pulina, Armando Tacchella, and Simone Vuotto. Consistency of property specification patterns with boolean and constrained numerical signals. In *Proc. of NFM*, volume 10811 of *LNCS*, pages 383–398. Springer, 2018.

[Nguyen *et al.*, 2007] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

[Öztok and Erdem, 2011] Umut Öztok and Esra Erdem. Generating explanations for complex biomedical queries. In *Proc. of AAAI*. AAAI Press, 2011.

[Perron *et al.*, 2023] Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP solver (invited talk). In *Proc. of CP*, volume 280 of *LIPIcs*, pages 3:1–3:2. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[Post *et al.*, 2011] Amalinda Post, Igor Menzel, and Andreas Podelski. Applying restricted english grammar on automotive requirements - does it work? A case study. In *Proc. of REFSQ*, volume 6606 of *LNCS*, pages 166–180. Springer, 2011.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[Schwitter *et al.*, 1995] Rolf Schwitter, Bernhard Hamburger, and Norbert E. Fuchs. Attempto: Specifications in controlled natural language. In *Proc. of the Workshop on Logische Programmierung*, pages 151–160, 1995.

[Schwitter, 2018] Rolf Schwitter. Specifying and verbalising answer set programs in controlled natural language. *Theory Pract. Log. Program.*, 18(3-4):691–705, 2018.

[Vuotto *et al.*, 2019] Simone Vuotto, Massimo Narizzano, Luca Pulina, and Armando Tacchella. Poster: Automatic consistency checking of requirements with reqv. In *Proc. of IEEE ICST*, pages 363–366. IEEE, 2019.

[Wallace, 1996] Mark Wallace. Practical applications of constraint programming. *Constraints An Int. J.*, 1(1/2):139–168, 1996.

[Wang *et al.*, 2024] Zhongsheng Wang, Jiamou Liu, Qiming Bao, Hongfei Rong, and Jingfeng Zhang. Chatlogic: Integrating logic programming with large language models for multi-step reasoning. In *Proc. of IJCNN*, pages 1–8. IEEE, 2024.

[Yang *et al.*, 2024] Xiaocheng Yang, Bingsen Chen, and Yik-Cheung Tam. Arithmetic reasoning with LLM: prolog generation & permutation. *CoRR*, abs/2405.17893, 2024.