

### 3 Knowledge Representation

Knowledge representation is at the intersection of maths, logic, language and computer sciences. Knowledge description systems rely on syntax to interoperate systems and users to one another. The base of such languages comes from the formalization of automated grammars by Chomsky (1956). It mostly consists of a set of production rules aiming to describe all accepted input strings. Usually, the rules are hierarchical and deconstruct the input using simpler rules until it matches a terminal symbol. This deconstruction is called parsing and is a common operation in computer science. More tools for the characterization of computer language emerged soon after thanks to Backus (1959) while working on a programming language at IBM. This is how the Backus-Naur Form (BNF) metalanguage was created on top of Chomsky's formalization.

A similar process happened in the 1970s, when logic based knowledge representation gained popularity among computer scientists (Baader 2003). Systems at the time explored notions such as rules and networks to try and organize knowledge into a rigorous structure. At the same time other systems were built based on First Order Logic (FOL). Then, around the 1990s, the research began to merge in search of common semantics in what led to the development of Description Logics (DL). This domain is expressing knowledge as a hierarchy of classes containing individuals.

From there and with the advent of the world wide web, actors of the internet were on the lookout for standardization and interoperability of computer systems. One such standardization took the name of "semantic web" and aimed to create a widespread network of connected services sharing knowledge between one another in a common language. At the beginning of the 21<sup>st</sup> century, several languages were created, all based on the World Wide Web Consortium (W3C) specifications called Resource Description Framework (RDF) (Klyne and Carroll 2004). This language is based on the notion of statements as triples. Each can express a unit of knowledge. All the underlying theoretical work of DL continued with it and created more expressive derivatives. One such derivative is the family of languages called Web Ontology Language (OWL) (Horrocks *et al.* 2003). The ontologies and knowledge graphs are more recent names for the representation and definition of categories (DL classes), properties and relation between concepts, data and entities.

Nowadays, when designing a knowledge representation, one usually starts with existing framework. The most popular in practice is certainly the classical relational database, followed closely by more novel methods for either big data or more expressive solutions like ontologies.

In our case we need a tool that is more expressive than ontologies while remaining efficient. Of course, this will lead to compromises, but can also have some interesting properties.

et tu vas parler de quoi dans ce chapitre (background, état de l'art , contribution ?) + lien avec le sujet de thèse + présenter le plan du chapitre (tu vas présenter grammaire dynamique car tu les utilises, tu en as besoin pour x raisons, tu vas présenter les ontologies car ... puis une proposition SELF)!



Figure 3.1: Noam Chomsky 2017

## 3.1 Grammar and Parsing

Grammar is an old tool that used to be dedicated to linguists. With the funding works by Chomsky and his Context-Free Grammars (CFG), these tools became available to mathematicians and shortly after to computer scientists.

A CFG is a formal grammar that aims to generate a formal language given a set of hierarchical rules. Each rule is given a symbol as a name. From any finite input of text in a given alphabet, the grammar should be able to determine if the input is part of the language it generates.

### 3.1.1 BNF

In computer science, popular metalanguage called BNF was created shortly after Chomsky's work on CFG. The syntax is of the following form :

```
1 <rule> ::= <other_rule> | <terminal_symbol> | "literals"
```

A terminal symbol is a rule that does not depend on any other rule. It is possible to use recursion, meaning that a rule will use itself in its definition. This actually allows for infinite languages. Despite its expressive power, BNF is often used in one of its extended forms.

In this context, we present a widely used form of BNF syntax that is meant to be human readable despite not being very formal. We add the repetition operators `*` and `+` that respectively repeat 0 and 1 times or more the preceding expression. We also add the negation operator `~` that matches only if the following expression does not match. We also add parentheses for grouping expression and brackets to group literals.

**Example 9.** We can make a grammar for all sequence of `A` using the rule `<scream> ::= "A"+`. If we want to make a rule that prevent the use of the letter `z` we can write `<no-sleep> ::= ~"z"`.

in this context : Dans le contexte de ta thèse ? de ce paragraphe ? We present: ici , dans ce chapitre ou plus tard dans ta thèse ?

### 3.1.2 Dynamic Grammar



A regular grammar is static, it is set once and for all and will always produce the same language. In order to be more flexible we need to talk about dynamic grammars and their associated tools and explain our choice of `gramatical` framework.

`gramMatical`

One of the main tools for both static and dynamic grammar is a parser. It is the program that will interpret the input into whatever usage it is meant for. Most of the time, a parser will transform the input into another similarly structured language. It can be a storage inside objects or memory, or compiled into another format, or even just for syntax coloration. Since a lot of usage requires the same kind of function, a new kind of tool emerged to make the creation of a parser simpler. We call those tools parser or compiler generators (Paulson 1982). They take a grammar description as input and gives the program of a parser of the generated language as an output.



dans ce paragraphe tu parles de grammaire dynamique ou des parsers pour grammaire dynamique ?

For dynamic grammar, these tools can get more complicated. There are a few ways a grammar can become dynamic. The most straightforward way to make a parser dynamic is to introduce code in the rule handling that will tweak variables affecting the parser itself (Souto *et al.* 1998). This allows for handling context in CFG without needing to rewrite the grammar.

Another kind of dynamic grammar is grammar that can modify themselves. In order to do this a grammar is valuated with reified objects representing parts of itself (Hutton and Meijer 1996). These parts can be modified dynamically by rules as the input gets parsed (Renggli *et al.* 2010; Alessandro and Piumarta 2007). This approach uses Parsing Expression Grammars (PEG)(Ford 2004) with Packrat parsing that Packrat parsing backtracks by ensuring that each production rule in the grammar is not tested more than once against each position in the input stream (Ford 2002). While PEG is easier to implement and more efficient in practice than their classical counterparts (Loff *et al.* 2018; Henglein and Rasmussen 2017), it offset<sup>s</sup> the computation load in memory making it actually less efficient in general (Becket and Somogyi 2008).

pas clair reformuler

Some tools actually just infer entire grammars from inputs and software (Höschele and Zeller 2017; Grünwald 1996). However, these kinds of approaches require a lot of input data to perform well. They also simply provide the grammar after expensive computations.

My system uses a grammar, composed of classical rules and is extended using meta-rules that activates once the classical grammar fails.

quel système ? il faut que tu dises dans l'intro du chapitre pourquoi et quelle partie de ta contribution utilise la grammaire dynamique, qu'on sache pourquoi tu nous présente tout ça et pourquoi tu dis ça ici.

### 3.2 Description Logics

On of the most standard and flexible way of representing knowledge is by using ontologies. They are based mostly on the formalism of Description Logics (DL). It is based on the notion of classes (or types) as a way to make the knowledge hierarchically structured. A class is a set of individuals that are called instances of the classes. Classes got the same basic properties as sets but can also be constrained with logic formula. Constraints can be on anything about the class or its individuals. Knowledge is also encoded in relations that are predicates over attributes of individuals.

It is common when using DLs to store statements into three boxes (Baader 2003):

ca serait possible d'avoir un petit exemple ?

- The TBox for terminology (statements about types)
- The RBox for rules (statements about properties) (Bürckert 1994)
- The ABox for assertions (statements about individual entities)

These are used mostly to separate knowledge about general facts (intentional knowledge) from specific knowledge of individual instances (extensional knowledge). The extra RBox is for "knowhow" or knowledge about entity behavior. It restricts usages of roles (properties) in the ABox. The terminology is often hierarchically ordered using a subsumption relation noted  $\sqsubseteq$ . If we represent classes or type as a set of individuals then this relation is akin to the subset relation of set theory.

There are several versions and extensions of DL. They all vary in expressivity. Improving the expressivity of DL system often comes at the cost of less efficient inference engines that can even become undecidable for some extensions of DL.

### 3.3 Ontologies and their Languages

Most AI problem needs a way to represent knowledge. The classical way to do so has been more and more specialized for each AI community. **Each their Domain Specific Language (DSL) that neatly fit the specific use it is intended to do.** There was a time when the branch of AI wanted to unify knowledge description under the banner of the "semantic web". **From numerous works,** a repeated limitation of the "semantic web" seems to come from the languages used. In order to guarantee performance of generalist inference engines, these languages have been restricted so much that they became quite complicated to use and quickly cause huge amounts of recurrent data to be stored because of some forbidden representation that will push any generalist inference engine into undecidability.

??? je comprends pas la phrase

en citer 1 ou 2

The most basic of these languages is perhaps RDF Turtle (Beckett and Berners-Lee 2011). It is based on triples with an XML syntax and has a graph as its knowledge structure (Klyne and Carroll 2004). A RDF graph is a set of RDF triples  $\langle sub, pro, obj \rangle$  which fields are respectively called subject, property and object. It can also be seen as a partially labeled directed graph  $(V, E)$  with  $V$  being the set of RDF nodes and  $E$  being the set of edges. This graph also comes with an incomplete label relation that associates a unique string called a Uniform Resource Identifier (URI) to most nodes. Nodes without an URI are called blank nodes. It is important that, while not named, blank nodes have a distinct internal identifier from one another that allows to differentiate them.

pas de ref our OWL et ses 3 versions ?

Built on top of RDF, the W3C recommended another standard called **OWL**. It adds the ability to have hierarchical classes and properties along with more advanced description of their arity and constraints. OWL is, in a way, more expressive than RDF (Van Harmelen *et al.* 2008, 1,p825). **It adds most formalism used in knowledge representation** and is widely used and interconnected. OWL comes in three versions: OWL Lite, OWL DL and OWL Full. The lite version is less advanced but its inference is decidable, OWL DL contains all notions of DL and the full version contains all features of OWL but is strongly undecidable.

je ne comprends pas cette phrase

W3C (2004a)

```
1 ex:ontology rdf:type owl:Ontology .
2 ex:name rdf:type owl:DatatypeProperty .
3 ex:author rdf:type owl:ObjectProperty .
4 ex:Book rdf:type owl:Class .
5 ex:Person rdf:type owl:Class .
6
7 _:x rdf:type ex:Book .
8 _:x ex:author _:x1 .
9 _:x1 rdf:type ex:Person .
10 _:x1 ex:name "Fred"^^xsd:string .
```

mettre cela en figure avec légende et citer dans le texte

The expressivity can also come from a lack of restriction. If we allow some freedom of expression in RDF statements, its inference can quickly become undecidable (Motik

2007). This kind of extremely permissive language is better suited for specific usage for other branches of AI. Even with this expressivity, several works still deem existing ontology system as not expressive enough, mostly due to the lack of classical constructs like lists, parameters and quantifiers that don't fit the triple representation of RDF.

One of the ways which have been explored to overcome these limitations is by adding a 4<sup>th</sup> field in RDF. This field is meant for context and annotations. This field is used for information about any statement represented as a triple, such as access rights, probabilities, or most of the time the source of the data (Tolksdorf *et al.* 2004). One of the other uses of the fourth field of RDF is to reify statements (Hernández *et al.* 2015). Indeed by identifying each statement, it becomes possible to efficiently form statements about statements.

A completely different approach is done by Hart and Goertzel (2008) in his framework for Artificial General Intelligence (AGI) called OpenCog. The structure of the knowledge is based on a rhizome, a collection of trees, linked to each other. This structure is called Atomspace. Each vertex in the tree is an atom, leaf vertexes are nodes, the others are links. Atoms are immutable, indexed objects. They can be given values that can be dynamic and, since they are not part of the rhizome, are an order of magnitude faster to access. Atoms and values alike are typed.

The goal of such a structure is to be able to merge concepts from widely different domains of AI. The major drawback being that the whole system is very slow compared to pretty much any domain specific software.

In my system, a similar structure is used but along with ontology oriented notions.

mettre le nom complet et acronyme entre parenthèse et en majuscule SELF

3.4 Self euh OK, on l'a vu pour les grammaires, pour RDF et Ontologies, c'est un peu noyé dans le discours, en tout cas, ce ne ressort pas clairement. Il faut mieux justifier dans tes paragraphes précédents et mettre en avant cette limitation (parmi d'autres) pour que quand on lise cette phrase, cela soit évident pour le lecteur.

As we have seen, the most used knowledge description systems (e.g. RDF, Ontologies and relational databases) have a common drawback: they are static. This means that they are created to be optimized for a specific use case, or gets general at the cost of efficiency. The main issue is that such system are unable to adapt to the use case by themselves. To fix this issue, a new knowledge representation model must be presented. The goal is to make a minimal language framework that can adapt to its use to become as specific as needed. If it becomes specific is must start from a generic base. IT ??? Since that base language must be able to evolve to fit the most cases possible, it must be neutral and simple.

To summarize, that framework must maximize the following criteria:

1. **Neutral:** Must be independent from preferences and be localization. ???
2. **Permissive:** Must allow as many data representation as possible.
3. **Minimalist:** Must have the minimum number of base axioms and as little native notions as possible.
4. **Adaptive:** Must be able to react to user input and be as flexible as possible.

je suppose que tu n'es pas le seul à mettre en avant cette limitation et qu'il y a d'autres approches proposés dans la littérature qui essaient d'y répondre. Pourquoi tu ne les reprends pas ? Ou est-ce que tu t'en inspires mais de quelle manière ? Une manière de positionner SELF serait de faire un tableau avec les approches existantes et les critères et cocher pour chaque approche à 43 quelles critères elles répondent (en expliquant).

Puis y mettre SELF pour montrer que SELF réponds à tous les critères et pas les autres approches par exemple, et justifier comment SELF répond à chaque critère.

In order to respect these requirements, we developed a framework for knowledge description. This Structurally Expressive Language Framework (SELF) is our answer to these criteria. SELF is inspired by RDF Turtle and Description Logic.



### 3.4.1 Knowledge Structure

SELF extends the RDF graphs by adding another label to the edges of the graph to uniquely identify each statement. This basically turns the system into a quadruple storage even if this forth field is transparent to the user.

**Axiom (Structure).** A SELF graph is a set of statements that transparently include their own identity. The closest representation of the underlying structure of SELF is as follows:

$$g_{\mathbb{U}} = (\mathbb{U}, S) : S = \{s = \langle sub, pro, obj \rangle : s \in \mathcal{D} \vdash s \wedge \mathcal{D}\}$$

with:

- $sub, obj \in \mathbb{U}$  being entities representing the *subject* and *object* of the statement  $s$ ,
- $pro \in P$  being the *property* of the statement  $s$ ,
- $\mathcal{D} \subset S$  is the *domain* of the world  $g_{\mathbb{U}}$ ,
- $S, P \subset \mathbb{U}$  with  $S$  the set of statements and  $P$  the set of properties,

This means that the world  $g_{\mathbb{U}}$  is a graph with the set of entities  $\mathbb{U}$  as vertices and the set of statements  $S$  as edges. This model also suppose that every statement  $s$  must be true if they belong to the domain  $\mathcal{D}$ . This graph is a directed 3-uniform hypergraph.

[rappeler section ou tu les présentes](#)  
Since sheaves are a representation of hypergraphs, we can encode the structure of SELF into a sheaf-like form. Each seed is a statement, the germ being the statement vertex. It is always accompanied of an incoming connector (its subject), an outgoing connector (its object) and a non-directed connector (its property). The sections are domains and must be coherent. Each statement, along with its property, makes a stalk as illustrated in figure 3.2.

The difference with a sheaf is that the projection function is able to map the pair statement-property into a labeled edge in its projection space. We map this pair into a classical labeled edge that connects the subject to the object of the statement in a directed fashion. This results in the projected structure being a correct RDF graph.

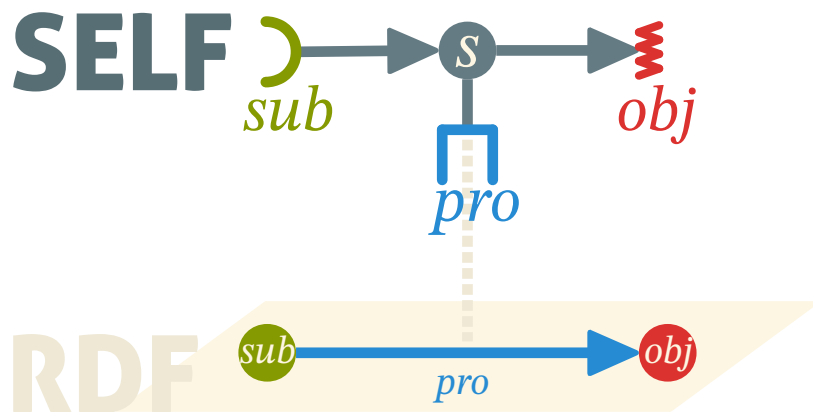
#### 3.4.1.1 Consequences

[convenient ? \(à vérifier je ne suis pas sur\)](#)

The base knowledge structure is more than simply convenience. The fact that statements have their own identity, changes the degrees of freedom of the representation. RDF has a way to represent reified statements that are basically blank nodes with properties that are related to information about the subject, property and object of a designated statement. The problem is that such statements are very differently represented and need 3 regular statements just to define. Using the fourth field, it becomes possible to make statements about any statements. It also becomes possible to

[je comprends pas: it requires to define 3 regular statements ?](#)





**Figure 3.2:** Projection of a statement from the SELF to RDF space.

express modal logic about statements or to express, various traits like the probability or the access rights of a statement.

The knowledge structure holds several restrictions on the way to express knowledge. As a direct consequence, we can add several theorems to the logic system underlying SELF. The axiom of **Structure** is the only axiom of the system.

**Theorem 1** (Identity). *Any entity is uniquely distinct from any other entity.*

This theorem comes from the axiom of **Extensionality** of ZFC. Indeed it is stated that a set **is a** unordered collection of distinct objects. Distinction is possible if and only if intrinsic identity is assumed. This notion of identity entails that a given entity cannot change in a way that would alter its identifier.

**Theorem 2** (Consistency). *Any statement in a given domain is consistent with any other statements of this domain.*

Consistency comes from the need for a coherent knowledge system and is often a requirement of such constructs. This theorem **also is a** consequence of the axiom of **Structure**:  $s \in \mathcal{D} \vdash s \wedge \mathcal{D}$ .

**Theorem 3** (Uniformity). *Any object in SELF is an entity. Any relations in SELF **are** restricted to  $\mathbb{U}$ .*

This also means that all native relations are closed under  $\mathbb{U}$ . This allows for a uniform knowledge database.

### 3.4.1.2 Native Properties

In the following, we suppose all notions from previous chapter. The difference is that we define and use only a subset of the functions defined in the SELF formalism. In

relation to the theory of SELF, we use the functional theory previously defined as the underlying formalism.

#### FIXME: Case variation for crossref

**Theorem** **theorem 1** lead to the need for two native properties in the system : *equality* and *name*.

The **equality relation**  $= : \mathbb{U} \rightarrow \mathbb{U}$ , behaves like the classical operator. Since the knowledge database will be expressed through text, we also need to add an explicit way to identify entities. This identification is done through the **name relation**  $\nu : \mathbb{U} \rightarrow L_{String}$  that affects a string literal to some entities. This lead us to introduce literals into SELF that is also entities that have a native value.

The axiom of **Structure** puts a type restriction on property. Since it compartments  $\mathbb{U}$  using various named subsets, we must adequately introduce an explicit type system into SELF. That type system requires a **type relation (named using the colon)**  $:: : \mathbb{U} \rightarrow T$ . That relation is complete as all entities have a type. **theorem 3** causes the set of entities to be universal. Type theory, along with Description Logic (DL), introduces a **subsumption relation**  $\subseteq : T \rightarrow T$  as a partial ordering relation to the types. Since types can be seen as sets of instances, we simply use the subset relation from set theory. In our case, the entity type is the greatest element of the lattice formed by the set of types with the subsumption relation  $(T, \subseteq)$ .

The **theorem 3** also allows for some very interesting meta-constructs. That is why we also introduce a signed **Meta relation**  $\mu : \mathbb{U} \rightarrow D$  with  $\mu^* = \bullet\mu$ . This allows to create domain from certain entities and to encapsulate domains into entities.  $\mu^*$  is for reification and  $\mu$  is for abstraction. This Meta relation also allows to express value of entities, like lists or various containers.

To fulfill the principle of adaptability and in order to make the type system more useful, we introduce the **parameter relation**  $\rho : \mathbb{U} \rightarrow \mathbb{U}$ . This relation affects a list of parameters, using the Meta relation, to some parameterized entities. This also allows for variables in statements.

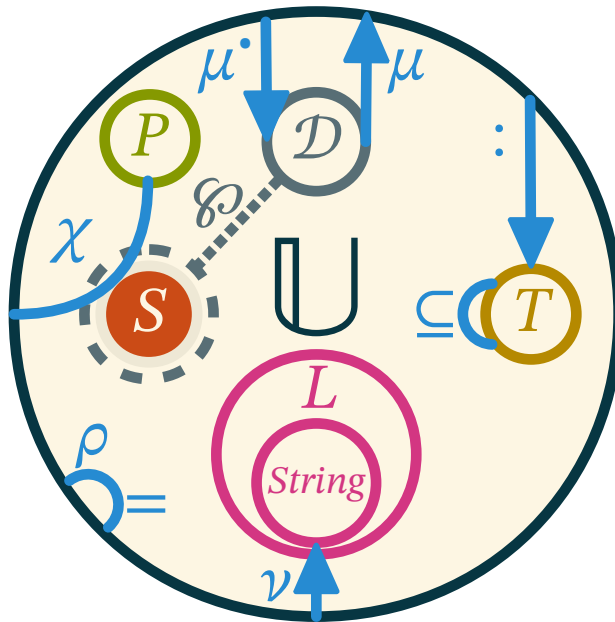
Since axiom of **Structure** gives the structure of SELF a hypergraph shape, we must port some notions of graph theory into our framework. Introducing the **statement relation**  $\chi : S \rightarrow \mathbb{U}$  reusing the same symbol as for the adjacency and incidence relation of graphs. This isn't a coincidence as this relation has the same properties.

**Example 10.** Since statements are triplets and edges,  $s_0$  gives the subject of a statement  $s$ . Respectively,  $s_1$  and  $s_2$  give the property and object of any statement. For adjacencies,  $\chi$  can give the set of statements any entity is the object or subject of. For any property  $pro$ , the notation  $\chi(pro)$  gives the set of statements using this property.

This allows us to port all the other notions of graphs using this relation as a base.

In figure 3.3, we present all the native relations along with their domains and most subsets of  $\mathbb{U}$ .





**Figure 3.3:** Venn diagram of subsets of  $\mathbb{U}$  along with their relations. Dotted lines mean that the sets are defined a subset of the wider set.

### 3.4.2 Syntax

Since we need to respect the requirements of the problem, the RDF syntax cannot be used to express the knowledge. Indeed, RDF states native properties as English nodes with a specific URI that isn't neutral. It also isn't minimalist since it uses an XML syntax so verbose that it is not used for most examples in the documents that defines RDF because it is too confusing and complex (W3C 2004b; W3C 2004c). The XML syntax is also quite restrictive and cannot evolve dynamically to adapt to the usage.

We need to define a new language that has contradictive qualities. It must be general, yet specific and minimalistic while expressive.

So the solution to the problem is to actually **define two languages that fit the criteria**: one minimalist and one adaptive. The issue is that we don't want the user to learn two languages and the second kind of language must be very specific and that violates the principle of neutrality we try to respect.



The only solution is to make a mechanism to adapt the language as it is used. We start off with a simple framework that uses a grammar.

The description of  $\mathcal{G}_0$  is pretty straightforward: it mostly is just a triple representation separated by whitespaces. The goal is to add a minimal syntax consistent with the axiom of **Structure**. In ?? 3.1, we give a simplified version of  $\mathcal{G}_0$ . It is written in a pseudo-BNF fashion, which is extended with the classical repetition operators  $*$  and  $+$  along with the negation operator  $\sim$ . All tokens have names in uppercase. We also add the following rule modifiers:

- `<~name>` are ignored for the parsing. However, the tokens are consumed and therefore acts like separators for the other rules.
- `<?name>` are inferred rules and tokens. They play a key role for the process of derivation explained in section 3.4.3.

```

1 <~COMMENT: <INLINE: "//" (~["\n", "\r"])*>
2 | <BLOCK: "/*" (~["*/"])*> > //Ignored
3 <~WHITE_SPACE: " " | "\t" | "\n" | "\r" | "\f">
4 <LITERAL: <INT> | <FLOAT> | <CHAR> | <STRING>> //Java definition
5 <ID: <TYPE: <UPPERCASE>(<LETTERS>|<DIGITS>)* >
6 | <ENTITY: <LOWERCASE>(<LETTERS>|<DIGITS>)*>
7 | <SYMBOL: (~[<LITERALS>, <LETTERS>, <DIGITS>])*>>
8
9 <worselfld> ::= <first> <statement>* <EOF>
10 <first> ::= <subject> <?EQUAL> <?SOLVE> <?EOS>
11 <statement> ::= <subject> <property> <object> <EOS>
12 <subject> ::= <entity>
13 <property> ::= <ID> | <?meta_property>
14 <object> ::= <entity>
15 <entity> ::= <ID> | <LITERAL> | <?meta_entity>

```

**Listing 3.1:** Simplified pseudo-BNF description for basic SELF.

In order to respect the principle of neutrality, the language must not suppose of any regional predisposition of the user. There are few exceptions for the sake of convenience and performance. The first exception is that the language is meant to be read from left to right and have an occidental biased `subject verb object` triple description. Another exception is for literals that use the same grammar as in classical Java. This means that the decimal separator is the dot (.). This concession is made for reasons of simplicity and efficiency, but it is possible to define literals dynamically in theory (see section 7.2.1).

Even if sticking to the ASCII subset of characters is a good idea for efficiency, SELF can work with UTF-8 and exploits the Unicode Character Database (UCD) for its token definitions (Unicode Consortium 2018b). This means that SELF comes keywords free and that the definition of each symbol is left to the user. Each notion and symbol is inferred (with the exception of the first statement which is closer to an imposed configuration file).

In  $\mathcal{G}_0$ , the first two token definitions are ignored. This means that comments and white-spaces will act as separation and won't be interpreted. Comments are there only for convenience since they do not serve any real purpose in the language. It was arbitrarily decided to use Java-style comments. White-spaces are defined against UCD's definition of the separator category `Z&` (see Unicode Consortium 2018a, chap. 4).

#### **FIXME: Lines ref are borken**

????? line 3.4.2 uses the basic Java definition for literals. In order to keep the independence from any natural language, boolean `literals` are not natively defined (since they are English words).

Another aspect of that language independence is found starting at line 3.4.2 where the definitions of `<UPPERCASE>`, `<LOWERCASE>`, `<LETTERS>` and `<DIGITS>` are defined from the UCD (respectively categories `Lu`, `Ll`, `L&`, `Nd`). This means that any language's upper case

can be used in that context. For performance and simplicity reasons we will only use ASCII in our examples and application.

The rule at section 3.4.5.1 is used for the definition of three tokens that are important for the rest of the input. `<EQUAL>` is the symbol for equality and `<SOLVE>` is the symbol for the *solution quantifier* (and also the language pendant of  $\mu^*$ ). The most useful token `<EOS>` is used as a statement delimiter. This rule also permits the inclusion of other files if a string literal is used as a subject. The underlying logic of this first statement will be presented in section 3.4.5.1. In the following examples we will consider that `<EQUAL> ::= "=", <SOLVE> ::= "?"` and `<EOS> ::= ";"`.

At line 3.4.2, we can see one of the most defining features of  $\mathcal{G}_0$ : statements. The input is nothing but a set of statements. Each component of the statements are entities. We defined two specific rules for the subject and object to allow for eventual runtime modifications. The property rule is more restricted in order to guarantee the non-ambiguity of the grammar.

### 3.4.3 Dynamic Grammar

The syntax we described is only valid for  $\mathcal{G}_0$ . As long as the input is conforming to these rules, the framework keeps the minimal behavior. In order to access more features, one needs to break a rule. We add a second outcome to handling with violations : **derivation**. There are several kinds of possible violations that will interrupt the normal parsing of the input :

- Violations of the `<first>` statement rule : This will cause a fatal error.
- Violations of the `<statement>` rule : This will cause a derivation if an unexpected additional token is found instead of `<EOS>`. If not enough tokens are present, a fatal error is triggered.
- Violations of the secondary rules (`<subject>`, `<entity>`, ...) : This will cause a fatal error except if there is also an excess of token in the current statement which will cause derivation to happen.

Derivation will cause the current input to be analyzed by a set of meta-rules. The main restriction of these rules is given in  $\mathcal{G}_0$ : each statement must be expressible using a triple notation. This means that the goal of the meta-rules is to find an interpretation of the input that is reducible to a triple and to augment  $\mathcal{G}_0$  by adding an expression to any `<meta_*>` rules. If the input has fewer than 3 entities for a statement then the parsing fails. When there is extra input in a statement, there is a few ways the infringing input can be reduced back to a triple.

#### 3.4.3.1 Containers

The first meta-rule is to infer a container. A container is delimited by, at least, a left and right delimiter (they can be the same symbol) and an optional middle delimiter. We infer the delimiters using the algorithm 1.

je t'ai dit d'expliquer algo 1 et 2 et je vois que pour algo 2 ce n'est pas fait et pour le un, il n'y a qu'une ligne qui est expliqué

---

#### Algorithm 1 Container meta-rule

---

```

1: function CONTAINER(Token current)
2:   LOOKAHEAD(current, EOS) ▷ Populate all tokens of the statement
3:   for all token in horizon do
4:     if token is a new symbol then delimiters.APPEND(token)
5:   if LENGTH(delimiters) < 2 then
6:     if COHERENTDELIMITERS(horizon, delimiters[0]) then
7:       INFERMIDDLE(delimiters[0]) ▷ New middle delimiter in existing containers
8:       return Success
9:     return Failure
10:  while LENGTH(delimiters) > 0 do
11:    for all (left, middle, right) in SORTEDDELIMITERS(delimiters) do
12:      if COHERENTDELIMITERS(horizon, left, middle, right) then
13:        INFERDELIMITER(left, right)
14:        INFERMIDDLE(middle) ▷ Ignored if null
15:        delimiters.REMOVE(left, middle, right)
16:        break
17:      if LENGTH(delimiters) stayed the same then return Success
18:  return Success

```

---

The function sortedDelimiters at line 1 is used to generate every ordered possibility and sort them using a few criteria. The default order is possibilities grouped from left to right. All coupled delimiters that are mirrors of each other following the UCD are preferred to other possibilities.

Checking the result of the choice is very important. At line 1 a function checks if the delimiters allow for triple reduction and enforce restrictions.

**Example 11.** For example, a property cannot be wrapped in a container (except if part of parameters). This is done in order to avoid a type mismatch later in the interpretation.

Once the inference is done, the resulting calls to inferDelimiter will add the rules listed in ?? 3.2 to  $\mathcal{G}_0$ . This function will create a `<container>` rule and add it to the definition of `<meta_entity>`. Then it will create a rule for the container named after the UCD name of the left delimiter (searching in the `NamesList.txt` file for an entry starting with "left" and the rest of the name or defaulting to the first entry). Those rules are added as a conjunction list to the rule `<container>`. It is worthy to note that the call to inferMiddle will add rules to the token `<MIDDLE>` independently from any container and therefore, all containers share the same pool of middle delimiters.

```

1 <meta_entity> ::= <container>
2 <container> ::= <parenthesis> | ...
3 <parenthesis> ::= "(" [<naked_entity>] (<?MIDDLE> <naked_entity>)* ")"
4 <naked_entity> ::= <statement> | <entity>

```

**Listing 3.2:** Rules added to the current grammar for handling the new container for parenthesis

The rule at section 3.4.3.1 is added once and enables the use of meta-statements inside containers. It is the language pendant of the  $\mu$  relation, allowing to wrap abstraction in a safe way.

**Example 12.** If we parse the expression  $a = (b, c);$ , we start by tokenizing it as `<ENTITY> <EQUAL> <SYMBOL><ENTITY><SYMBOL><ENTITY><SYMBOL> <EOS>` (ignoring whitespaces and comments). This means that the statement is 4 tokens too long to form a triple. This triggers a parsing error and then an evaluation using meta-rules. All the `<ID>` tokens are new symbols, but they don't have the same subtype. This means that candidate delimiters are `(, ,` and `)`. To Infer the inference there's only one combination and the left delimiter and right delimiters are found via their Unicode description. The comma is left to be inferred as the middle delimiter. The grammar is rewritten and the statement becomes `<ENTITY> <EQUAL> <container> <EOS>` which is a valid triple statement.

### 3.4.3.2 Parameters

If the previous rule didn't fix the parsing of the statement, we continue with the following meta-rule. Parameters are extra containers that are used after an entity. Every container can be used as parameters. We detail the analysis in algorithm 2.

---

#### Algorithm 2 Parameter meta-rule

---

```

1: function PARAMETER(Entity[] statement)
2:   reduced = statement
3:   while LENGTH(reduced) > 3 do
4:     for i from 0 to LENGTH(reduced) - 1 do
5:       if NAME(reduced[i]) not null and
6:       TYPE(reduced[i+1]) = Container and
7:       COHERENTPARAMETERS(reduced, i) then
8:         param = INFERPARAMETER(reduced[i], reduced[i+1])
9:         reduced.REMOVE(reduced[i], reduced[i+1])
10:        reduced.INSERT(param, i)                                ▷ Replace parameterized entity
11:        break
12:     if LENGTH(statement) stayed the same then return Success
13:   return Failure

```

---

The goal is to match extra containers with the preceding named entity. The container is then combined with the preceding entity into a parameterized entity.

The call to `inferParameter` will add the rule in ?? 3.3, replacing `<?container>` with the name of the container used.

**Example 13.** In this case we have to parse  $f(x) = x;$ . If we already have the parenthesis delimiter defined, it becomes `<ENTITY> <container> <EQUAL> <ENTITY> <EOS>`. Since the statement isn't a triple we execute the meta-rules. The container rule finds no new symbols and fails. Then the parameter meta-rule will reduce the statement by bounding the first entity to the following container and mark it as a parameterized entity. This gives `<meta_entity> <EQUAL> <ENTITY> <EOS>`.

```

1 <meta_entity> ::= <ID> <?container>
2 <meta_property> ::= <ID> <?container>

```

**Listing 3.3:** Rules added to the current grammar for handling parameters

### 3.4.3.3 Operators

A shorthand for parameters is the operator notation. It allows to affect a single parameter to an entity without using a container. This is essentially syntactic sugar: a feature that makes the language easier to write. It is most used for special entities like quantifiers or modifications. This is why, once used, the parent entity takes a polymorphic type, meaning that type inference will not issue errors for any usage of them. Details of the way the operators are reduced is exposed in algorithm 3.



#### Algorithm 3 Operator meta-rule

```
1: function OPERATOR(Entity[] statement)
2:   reduced = statement
3:   while LENGTH(reduced) > 3 do
4:     for i from 0 to LENGTH(reduced) - 1 do
5:       if  $\nu$ (reduced[i]) not null and
6:        $\nu$ (reduced[i+1]) not null and
7:       ( $\nu$ (reduced[i]) is a new symbol or
8:       reduced[i] has been parameterized before) and
9:       COHERENTOPERATOR(reduced, i) then
10:        op = INFEROPERATOR(reduced[i], reduced[i+1])
11:        reduced.REMOVE(reduced[i], reduced[i+1])
12:        reduced.INSERT(op, i) ▷ Replace parameterized entity
13:        break
14:     if LENGTH(statement) stayed the same then return Success
15:   return Failure
```



#### TODO: More explanation

From the call of inferOperator, comes new rules explicated in ?? 3.4. The call also adds the operator entity to an inferred token <OP>.

```
1 <meta_entity> ::= <?OP> <ID>
2 <meta_property> ::= <?OP> <ID>
```

Listing 3.4: Rules added to the current grammar for handling operators

**Example 14.** With the input ! $x = 0$ ; we parse <SYMBOL> <ENTITY> <EQUAL> <LITERAL> <EOS>. This cannot be a container since the new symbol is at the beginning without any mirroring possible. It cannot be a parameter since no container is present. But this will conclude with the operator meta-rule as the new symbol precedes an entity. This becomes <meta\_entity> <EQUAL> <LITERAL> <EOS> and becomes a valid statement.

If all meta-rules fail, then the parsing fails and returns an error to the user like in classical occurrences.

### 3.4.4 Contextual Interpretation

While parsing another important part of the processing is done after the success of a grammar rule. The grammar in SELF is valuated, meaning that each rule has to return an entity. A set of functions are used to then populate the knowledge description system



with the right entities or retrieve an existing one that corresponds to what is being parsed.

When parsing, the rules `<entity>` and `<property>` will trigger the creation or retrieval of an entity. This mechanism will use the name of the entity to retrieve an entity with the same name in a given scope. If no such entity exists it is created and added to the current scope.

#### 3.4.4.1 Naming and Scope

When parsing an entity by name, the system will first request for an existing entity with the same name. If such an entity is retrieved, it is returned instead of creating a new one. The validity of a name is limited by the notion of scope.

A scope is the reach of an entity's direct influence. It affects the naming relation by removing variable names. Scopes are delimited by containers and statements. This local context is useful when wanting to restrict the scope of the declaration of an entity. The main goal of such restriction is to allow for a similar mechanism as the RDF namespaces. This also makes the use of variables possible, akin to RDF blank nodes.

The scope of an entity has three special values :

- Variable: This scope restricts the scope of the entity to only the other entities in its scope.
- Local: This scope is temporarily **bound** to a given entity during the parsing. This scope is limited to the statement being interpreted.
- Global: This scope means that the name has no scope limitation.

The scope of an entity also contains all its parent entities, meaning all containers or statement the entity is part of. This is used when choosing between the special values of the scope. The process is detailed in algorithm 4.

The process happens for each entity created or requested by the parser. If a given entity is part of any other entity, the enclosing entity is added to its scope. When an entity is enclosed in any entity while already being a parameter of another entity, it becomes a variable since it is referenced twice in the same statement.


#### 3.4.4.2 Instanciation identification

When a parameterized entity is parsed, another process starts to identify if a compatible instance already exists. From theorem 1, it is impossible for two entities to share the same identifier. This makes mandatory to avoid creating an entity that is equal to an existing one. Given the order of which parsing is done, it is not always possible to determine the parameter of an entity before its creation. In that case a later examination will merge the new entity onto the older one and discard the new identifier.

---

**Algorithm 4** Determination of the scope of an entity


---



```
1: function INFERSCOPE(Entity  $e$ )
2:   Entity[] reach = []
3:   if : ( $e$ ) =  $S$  then
4:     for all  $i \in \chi(e)$  do reach.APPEND(INFERVARIABLE( $i$ ))    ▷ Adding scopes nested in statement  $e$ 
5:     for all  $i \in \mu^*(e)$  do reach.APPEND(INFERVARIABLE( $i$ ))    ▷ Adding scopes nested in container  $e$ 
6:     if  $\exists \rho(e)$  then
7:       Entity[] param = INFERSCOPE( $\rho(e)$ )
8:       for all  $i \in \text{param}$  do param.REMOVE(INFERSCOPE( $i$ ))    ▷ Remove duplicate scopes from
parameters
9:       for all  $i \in \text{param}$  do reach.APPEND(INFERVARIABLE( $i$ ))    ▷ Adding scopes from paramters of  $e$ 
10:    SCOPE( $e$ )  $\leftarrow$  reach
11:    if GLOBAL  $\notin$  SCOPE( $e$ ) then SCOPE( $e$ )  $\leftarrow$  SCOPE( $e$ )  $\cup$  {LOCAL}
return reach
12: function INFERVARIABLE(Entity  $e$ )
13:   Entity[] reach = []
14:   if LOCAL  $\in$  SCOPE( $e$ ) then
15:     for all  $i \in \text{SCOPE}(e)$  do
16:       if  $\exists e_p \in \mathbb{U} : \rho(p) = i$  then    ▷  $e$  is already a parameter of another entity  $e_p$ 
17:         SCOPE( $e$ )  $\leftarrow$  SCOPE( $e$ )  $\setminus$  {LOCAL}
18:         SCOPE( $e_p$ )  $\leftarrow$  SCOPE( $e_p$ )  $\cup$  SCOPE( $e$ )
19:         SCOPE( $e$ )  $\leftarrow$  SCOPE( $e$ )  $\cup$  {VARIABLE,  $p$ }
20:     reach.APPEND( $e$ )
21:     reach.APPEND(SCOPE( $e$ ))
return reach
```

---

### 3.4.5 Structure as a Definition




The derivation feature on its own does not allow to define most of the native properties. For that, one needs a light inference mechanism. This mechanism is part of the **default inference engine**. This engine only works on the principle of structure as a definition. Since all names must be neutral from any language, **that engine cannot rely on classical mechanisms** like configuration files with keys and values or predefined keywords.

To use SELF correctly, one must be familiar with the native properties and their structure or implement their own inference engine to override the default one.



#### 3.4.5.1 Quantifiers



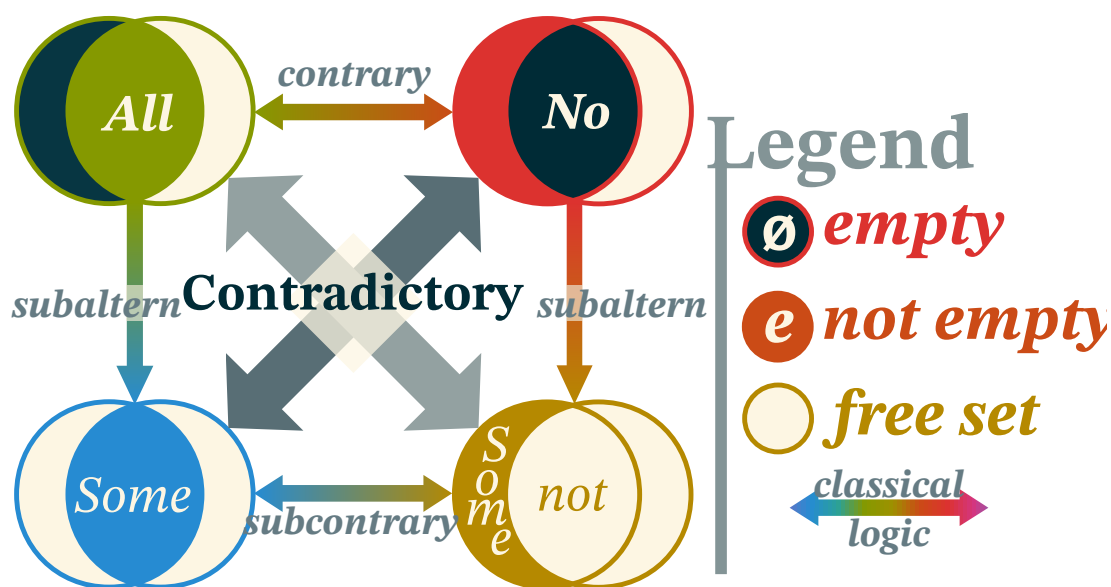
In SELF quantifiers differ from their mathematical counterparts. The quantifiers are special entities that are meant to be of a generic type that matches any entities including quantifiers. There are infinitely many quantifiers in SELF but they are all derived from a special one called the *solution quantifier*. We mentioned it briefly during the definition of the grammar  $\mathbb{Q}_0$ . It is the language equivalent of  $\mu^*$  and is used to extract and evaluate reified knowledge (see section 3.4.1.2).

**Example 15.** The statement `bob is <SOLVE>(x)` will give either a default container filled with every value that the variable `x` can take or if the value is unique, it will take that value. If there is no value it will default to `<NULL>`, the exclusion quantifier.

How are other quantifiers defined? We use a definition akin to Lindstöm quantifiers (1966) which is a generalization of counting quantifiers (Gradel et al. 1997). Meaning that a quantifier is defined as a constrained range over the quantified variable. We suppose five quantifiers as existing in SELF as native entities.

- The **solution quantifier** `<SOLVE>` noted  $\S$  in classical mathematics, turns the expression into the possible value range of its variable. It is like replacing it by the natural expression “those  $x$  that”.
- The **universal quantifier** `<ALL>` behaves like  $\forall$  and forces the expression to take every possible value of its variable.
- The **existential quantifier** `<SOME>` behaves like  $\exists$  and forces the expression to match *at least one* arbitrary value for its variable.
- The **uniqueness quantifier** `<ONE>` behaves like  $!\exists$  and forces the expression to match *exactly one* arbitrary value for its variable.
- The **exclusion quantifier** `<NULL>` behaves like  $\neg\exists$  and forces the expression to not match the value of its variable.

The last four quantifiers are inspired from Aristotle’s square of opposition (D’Alfonso 2011).



**Figure 3.4:** Aristotle’s square of opposition

In SELF, quantifiers are not always followed by a quantified variable and can be used as a value. In that case the variable is simply anonymous. We use the exclusion quantifier as a value to indicate that there is no value, sort of like `null` or `nil` in programming languages.



**Example 16.** If we want to express the fact that a glass of water is not empty we can write either `glass contains ~(~);` or `glass ~(contains) ~` with `<NULL> = ~`. This shows that `<NULL>` is used for negation and to indicate the absence of value.

This property is quite handy as it require only one symbol and allows for complex constructs that are difficult to explain using available paradigms.

In ?? 3.5, we present an example file that is meant to define most of the useful native properties along with default quantifiers.

```
1 * =? ;
2 ?(x) = x; //Optional definition
3 ?~ = { };
4 ?_ ~(=) ~;
5 ?!_ = {_};
6
7 (*e, !T) : (e :: T); *T : (T :: Type);
8 *T : (Entity / T);
9
10 :: :: Property(Entity, Type);
11 (__) :: Statement;
12 (~, !, _, *) :: Quantifier;
13 ( )::Group;
14 { }::Set;
15 [ ]::List;
16 < >::Tuple;
17 Collection/(Set,List,Tuple);
18 0 :: Integer; 0.0::Float;
19 '\0'::Character; ""::String;
20 Literal/(Boolean, Integer, Float, Character, String);
21
22 (*e, !(s::String)) : (e named s);
23 (*e(p), !p) : (e param p);
24 *(s p o):((s p o) subject s),((s p o) property p),((s p o) object o));
```

**Listing 3.5:** The default lang.w file.

At section 3.4.5.1, we give the first statement that defines the solution quantifier's symbol. The reason this first statement is shaped like this is that global statements are always evaluated to be a true statement. Since domains are sets of statements, this means that anything equaling the solution quantifier at this level will be evaluated as a domain. This is because the entity is a domain **by structure**. If it is a single entity then it becomes synonymous to the entire SELF domain and therefore contains everything. We can infer that it becomes the universal quantifier.

If it is a string literal, then it must be either a file path or URL or a valid SELF expression.

**Example 17.** Using the first statement, we can include external domains akin to the `import` directive in Java. Writing `"path/lang.w" = ? ;` as a first statement will make the process parse the file located at `path/lang.w` and insert it at this spot.

All statements up to section 3.4.5.1 are quantifiers definitions. On the left side we got the quantifier symbol used as a parameter to the solution quantifier using the operator notation. On the right we got the domain of the quantifier. The exclusive quantifier

has as a range the empty set. For the existential quantifier we have only a restriction of it not having an empty range. At last, the uniqueness quantifier got a set with only one element matching its variable (noting that anonymous variables doesn't match necessarily other anonymous variables in the same statement).

In ?? 3.5 the type hierarchy can be illustrated by the figure 3.5. It consists of entities that are either parameterized or not and that has a value or not.

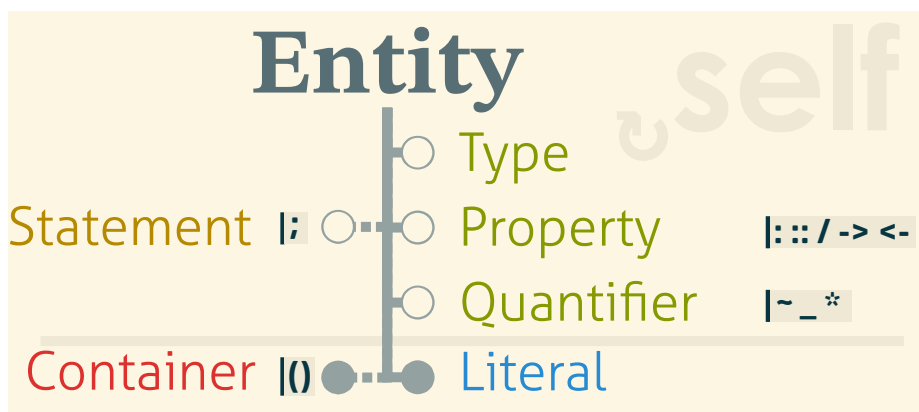


Figure 3.5: Hierarchy of types in SELF

**TODO: Explain figure**

### 3.4.5.2 Inferring Native Properties

All native properties can be inferred by structure using quantified statements. Here is the structural definition for each of them:

- $=$  (at section 3.4.5.1) is the equality relation given in the first statement.
- $\subseteq$  (at section 3.4.5.1) is the first property to relate a particular type to all types. That type becomes the entity type.
- $\mu^*$  (at section 3.4.5.1) is the solution quantifier discussed above given in the first statement.
- $\mu$  is represented using containers.
- $\nu$  (at section 3.4.5.1) is the first property affecting a string literal uniquely to each entity.
- $\rho$  (at section 3.4.5.1) is the first property to effect to all entities a possible parameter list.
- $:$  (at section 3.4.5.1) is the first property that matches every entity to a type.
- $\chi$  (at section 3.4.5.1) is the first property to match for all statements.

We limit the inference to one symbol to eliminate ambiguities and prevent accidental re-definition of native properties. This also improves performance as the inference is stopped after finding a first matching entity that can be used programmatically using a single constant.

### 3.4.6 Extended Inference Mechanisms

In this section we present the default inference engine. It is quite limited since it is meant to be universal and the goal of SELF is to provide a framework that can be used by specialists to define and code exactly what tools they need.

Inference engines need to create new knowledge but this knowledge shouldn't be simply merged with the explicit user provided domain. Since this knowledge is inferred, it is not exactly part of the domain but must remain consistent with it. This knowledge is stored in a special scope dedicated to each inference engine. This way, inference engines can use defeasible logic or have dynamic inference from any knowledge insertion in real time.

#### 3.4.6.1 Type Inference

Type inference works on matching types in statements. The main mechanism consists in inferring the type of properties in a restrictive way. Properties have a parameterized type with the type of their subject and object. The goal is to make that type match the input subject and object.

For that we start by trying to match the types. If the types differ, the process tries to reduce the more general type against the lesser one (subsumption-wise). If they are incompatible, the inference uses some light defeasible logic to undo previous inferences. In that case the types are changed to the last common type in the subsumption tree.

However, this may not always be possible. Indeed, types can be explicitly specified as a safeguard against mistakes. If that's the case, an error is raised and the parsing or knowledge insertion is interrupted.

#### 3.4.6.2 Instanciation

Another inference mechanism is instantiation. Since entities can be parameterized, they can also be defined against their parameters. When those parameters are variables, they allow entities to be instantiated later.

Since entities are immutable, updating their instance can be quite tricky. Indeed, parsing happens from left to right and therefore an entity is often created before all the instantiation information are available. Even harder are completion of definition in several separate statements. In all cases, a new entity is created and then the inference realize that it is either matching a previous definition and will need to be merged with the older entity or it is a new instance and needs all properties duplicated and instantiated.

This gives us two mechanisms to take into account: merging and instanciating.

Merging is pretty straightforward: the new entity is replaced with the old one in all of the knowledge graph. containers, parameterized entities, quantifiers and statements must be duplicated with the correct value and the original destroyed. This is a heavy



and complicated process but seemingly the only way to implement such a case with immutable entities.

Instantiating is similar to merging but even more complicated. It starts with computing a relation that maps each variable that needs replacing with their grounded value. Then it duplicates all knowledge about the parent entity while applying the replacement map.

## 3.5 Example

In the following section, a use case of the framework will be presented. First we have to explain a few notions.

### 3.5.1 Modality of Statements

In the field of logic there exists one special flavor of it called *modal logic*. It lays the emphasis upon the qualifications of statements, and especially the way they are interpreted. This is a very appropriate example for SELF. The modality of a statement acts like a modifier, it specifies a property regarding its plausibility, origin or validity.

**Example 18.** In the figure 3.6, we present a case of three persons gossiping, Alice, Becky and Carol. The presentation is inspired by the work of Schwarzentruher (2018). Here is a list of the statements in this example:

- Alice said to Becky that Carol should *probably* change her style from  $C_1$  to  $C_2$ .
- Becky said to Alice that she finds the Carol's style *usually* good.
- Alice told Carol that Becky told her that she should *sometimes* change her style to  $C_2$ .

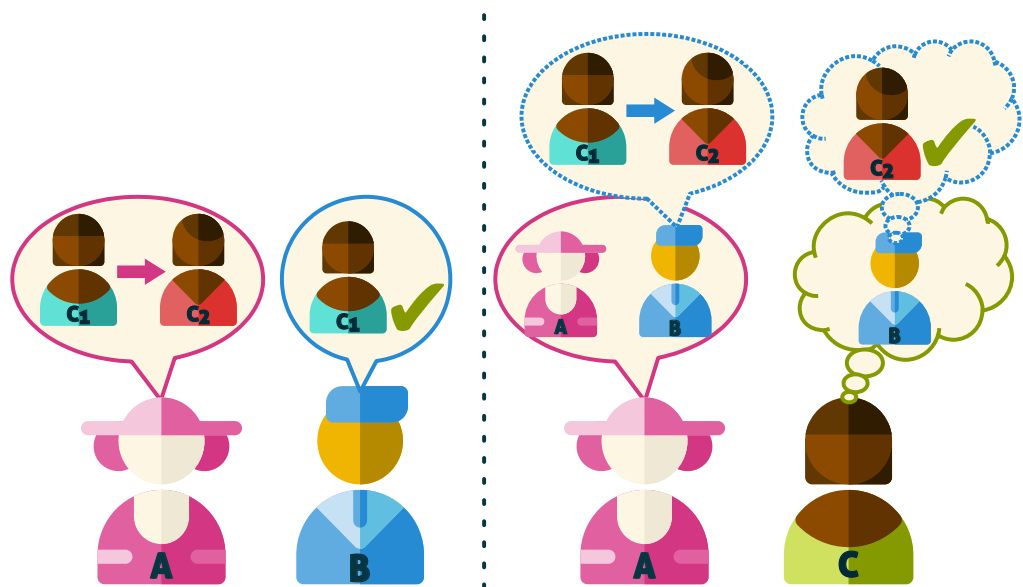
The following statement can be inferred:

- Carol *possibly* thinks that Becky thinks that the style  $C_2$  is *often* good.

In the example, all modalities are *emphasized*. One can notice an interesting property of these statements in that they are about other statements. This kind of description is called *higher order knowledge*.

### 3.5.2 Higher order knowledge

SELF is based on the ability to easily process higher order knowledge. In that case the term *order* refers to the level of abstraction of a statement (Schwarzentruher 2018). For such usages, a hypergraph structures such as SELF is using is a clear advantage in terms of expressivity and ease of manipulation of those statements. This is due to the higher dimensionality of sheaves (and by extension hypergraphs) that makes meta-statement as simple to express as any other statement. This chain of abstractions using meta-statements is where the higher order knowledge is encoded. ???



**Figure 3.6:** Example of modal logic propositions: Alice gossips about what Beatrice said about Claire

ajouter numéro + légende à chacun  
des 2,

In the following listings, we present the previous example using RDF and SELF to describe knowledge of the gossip.

```
1 @prefix : <http://genn.io/self/gossip#> .
2 @prefix owl: <http://www.w3.org/2002/07/owl#> .
3 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4 @prefix xml: <http://www.w3.org/XML/1998/namespace> .
5 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
6 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
7 @base <http://genn.io/self/gossip> .
8
9 <http://genn.io/self/gossip> rdf:type owl:Ontology ;
10                               owl:imports rdf: .
11
12 :modality rdf:type owl:AnnotationProperty ;
13           rdfs:range :Modality .
14
15 :told_a rdf:type owl:ObjectProperty .
16 :told_b rdf:type owl:ObjectProperty .
17 :told_c rdf:type owl:ObjectProperty .
18 :Modality rdf:type owl:Class .
19 :good rdf:type owl:NamedIndividual .
20 :is rdf:type owl:NamedIndividual ,
21     rdf:Property .
22 :probably rdf:type owl:NamedIndividual ,
23           :Modality .
24 :s1 rdf:type owl:NamedIndividual ,
25     rdf:Statement ;
26     rdf:object :c2 ;
27     rdf:predicate :worsethan ;
28     rdf:subject :c ;
29     :modality :probably .
30 :s2 rdf:type owl:NamedIndividual ;
31     rdf:object :good ;
32     rdf:predicate :is ;
33     rdf:subject :c ;
34     :modality :usually .
35 :s3 rdf:type owl:NamedIndividual ,
36     rdf:Statement ;
37     rdf:object :s4 ;
38     rdf:predicate :told_a ;
39     rdf:subject :b .
40 :s4 rdf:type owl:NamedIndividual ,
41     rdf:Statement ;
42     rdf:object :c2 ;
43     rdf:predicate :should ;
44     rdf:subject :c ;
45     :modality :sometimes .
46 :should rdf:type owl:NamedIndividual ,
47         rdf:Property .
48 :sometimes rdf:type owl:NamedIndividual ,
49           :Modality .
50 :told_a rdf:type owl:NamedIndividual ,
51         rdf:Property .
52 :usually rdf:type owl:NamedIndividual ,
53         :Modality .
54 :worsethan rdf:type owl:NamedIndividual ,
55           rdf:Property .
56 :a rdf:type owl:NamedIndividual ;
57   :told_b :s1 ;
```

```

58      :told_c :s3 .
59 :b rdf:type owl:NamedIndividual ;
60      :told_a :s2 .
61 :c rdf:type owl:NamedIndividual .
62 :c2 rdf:type owl:NamedIndividual .

```

```

1 "lang.s" = ? ;
2 a told(b) probably(c worsethan ctwo);
3 b told(a) usually(c is good);
4 a told(c) (b told(a) sometimes(c should ctwo));

```

It is obvious that the SELF version is an order of magnitude more concise than RDF to express modal logic. The 4 lines of SELF are **equivalent** to the 62 lines of RDF. In the RDF version we use the reified statements :s1, :s2, :s3 and :s4 along with a :modality annotation to express high order knowledge and modalities. In SELF, everything is inferred by structure and one can start exploiting their database right away.

CONCLUSION !!!!