Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol:

GOS
A new declarative tool for modelling and solving *CSPs* to *SAT*

Document: Report

Alumne: Roger Generoso Masós

Tutor: Jordi Coll i Mateu Villaret
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea: Llenguatges i Sistemes Informàtics

Convocatòria (mes/any)

Juny 2020

Acknowledgments

My most sincere thanks to my two supervisors, Jordi Coll and Mateu Villaret, for introducing me to the exciting world of constraint solving problems and for proposing me this project that I have enjoyed throughout the time that I have been doing it.

Contents

List of Figures

List of Tables

Listings

1. Introduction

When somebody wants to solve a problem, surely, the most common approach in the programming world is to use an imperative programming language and define an algorithm with the steps to solve it. But there are many alternatives to that.

Constraint Satisfaction Problems (CSP)s are a type of problems in which variables are defined and, by applying constraints, you try to limit the domain of this variables until you reach a solution, but without proposing any specific algorithm to solve it. This kind of problems are easily modelled with declarative programming languages.

Declarative programming languages attempt to describe what the program must accomplish in terms of the problem domain, rather than describe how to accomplish it as a sequence of the programming language primitives. This is in contrast with imperative programming, which implements algorithms in explicit steps.

A subset of declarative languages are *modelling* languages. This project will be focused on this subset and the main purpose will be create a new declarative programming language for modelling any CSP to Boolean Satisfiability (SAT).

One of the most successful methodologies for solving CSP relies on the conversion into SAT problems. The advantage is the wide availability of free and efficient SAT-solvers.

A SAT problem contains a formula built on a set of boolean variables, which can take only value *true* (or 1) and *false* (or 0). A solution to SAT problem is an assignment of values *true/false* to the logical variables, such that all clauses are satisfied.

Currently there exist some declarative languages to model CSPs, such as *MiniZinc* (Stuckey et al., 2018), *ESSENCE*' (Frisch et al., 2008), *Picat* (Zhou & Kjellerstrand, 2016), *WSimply* (Ansótegui et al., 2013) and many others, which have their own compiling and solving systems. Some of these systems support automatic reformulation to SAT. However, one of the current research lines of the Logic and Programming (L∧P) research group of the *University of Girona* is to find efficient SAT encodings of particular constraints. For this reason, the L∧P group is interested in having its own SAT declarative modelling language and compiler that can be directly integrated with their SAT encoding systems.

The name chosen for the tool is *GOS* (**G**irona **O**ptimization **S**ystem)[1] and the language

[1]It is true that the final result is not (yet) compatible with the optimization of CSPs, but this name was decided in agreement with the supervisors, considering that one of the first future tasks for this work would be add support to optimization.

that *GOS* will use is *BUP*. As a curiosity, the name choice lies in the fact that in *Catalan*, my mother tongue, *GOS* means *dog* and, also in Catalan, the onomatopoeia that describes the sound that dogs make (the language they speak) is *BUP-BUP!* (*woof-woof!*).

Therefore, *GOS* (*dog*) is the name of the tool implemented and *BUP* (*woof*) is the language defined that is used by *GOS*. Apart from that, a real *dog* (*GOS*) can be considered a *tracker*, and that is also what the project is about: looking for a solution to CSPs.

1.1. Motivations

After enjoy studying *Declarative Programming* and *Compilers* subjects, the *Logic and Programming* (L∧P) research group of the University of Girona proposed me a project that was a mix of the two subjects: creating a compiler for a new declarative language to make up for the need of L∧P group of a language with high expressiveness level to encode CSP to SAT to be used in their current research. I could not miss this opportunity.

By doing this project I hope to gain a broad understanding of how compilers work, in addition to get a deeper knowledge of modelling with SAT.

I hope the result of this project to be a tool that can be used by anyone who wants to easily encode with SAT.

1.2. Purpose

The main purpose of the project is to obtain a new declarative programming language for modelling any CSP to SAT. To achieve this, it will be necessary to think and define a new programming language allowing the user define CSPs as SAT in a declarative way, with a higher level of expressiveness, and implement a compiler that integrates a SAT-solver to obtain the solution of the modelled problems. The goal is to provide an easy gateway for modelling with SAT, that has extremely efficient solvers.

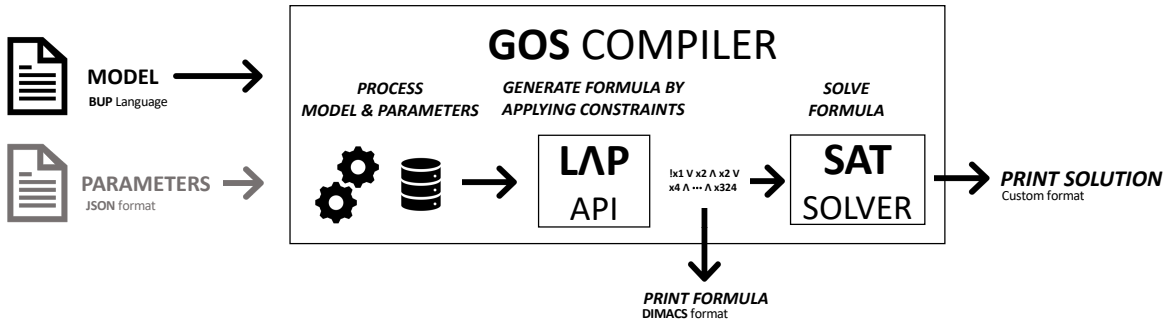


Figure 1.1: High level *GOS* tool schema

The figure 1.1 is a high level schema of the project. We could exemplify this schema modelling the *Sudoku* problem.
The **model file** will be written using the *BUP* language and it would define:

- Necessary parameters (initially fixed sudoku values, board size,...).
- Variables used to model the problem.
- Constraints applied over those variables (no repeated numbers in each row, column and sub-square, initial values must be respected,...)
- Output format.

The following code is a real example of using *BUP* modelling the Sudoku problem (model file):

```
viewpoint:
    var p[9][9][9];
    param int iniSudoku[9][9];

constraints:
    forall(i in 0..8, j in 0..8){
        EO(p[i][j][_]); // One value per cell
        AMO(p[i][_][j]); // Each value one time per row
        AMO(p[_][i][j]); // Each value one time per column
    };
    //Each value one time per block
    forall(i in [0,3,6], j in [0,3,6], k in 0..8){
        AMK([p[i+l][j+g][k] | l in 0..2, g in 0..2], 1);
    };
    //Initialize input fixed sudoku values.
    forall(i in 0..8, j in 0..8){
        if(iniSudoku[i][j] != 0){
            p[i][j][iniSudoku[i][j]-1];
        };
    };
output:
    "Sudoku solution: \n";
    [ k+1 ++ " " ++ ((j+1) % 3 == 0 ? " " : "") ++ (j==8 ? (i+1) % 3 == 0 ? "\n↩
    ↪ \n": "\n" : "") | i in 0..8, j in 0..8, k in 0..8 where p[i][j][k]];
```

The model has a parameter `iniSudoku`, that is the initial sudoku to solve, and an array of variables **p**, where `p[i][j][k]` is true when the cell with row `i` and column `j` has the value `k`.

The **parameters file** would give the values to the required model parameters. The following listing is an example of sudoku instance (parameters file):

```
{
    "iniSudoku" : [
        [8, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 3, 6, 0, 0, 0, 0, 0],
        [0, 7, 0, 0, 9, 0, 2, 0, 0],
        [0, 5, 0, 0, 0, 7, 0, 0, 0],
        [0, 0, 0, 0, 4, 5, 7, 0, 0],
```

```
        [0, 0, 0, 1, 0, 0, 0, 3, 0],
        [0, 0, 1, 0, 0, 0, 0, 6, 8],
        [0, 0, 8, 5, 0, 0, 0, 1, 0],
        [0, 9, 0, 0, 0, 0, 4, 0, 0]
    ]
}
```

GOS compiler will process these data and, by using the *L∧P API*[2], will generate the propositional formula as a result of applying the defined constraints over the model variables.
Once the formula is generated, the compiler can print the formula in a standard format, *DIMACS*, or apply the solver to get a solution. Basically, the process would be like doing a compilation of a high-level language (in our case, *BUP*), to assembler (where SAT is the assembler, the lowest-level language), using *GOS* compiler.

The following constraint makes sure that the initial values of the sudoku are respected.

```
//Initialize input fixed sudoku values.
forall(i in 0..8, j in 0..8){
    if(iniSudoku[i][j] != 0){
        p[i][j][iniSudoku[i][j]-1];
    };
};
```

The translation of this constraint to a SAT propositional formula, given the example instance: `p[0][0][7]` $\wedge$ `p[1][2][2]` $\wedge$ `p[1][3][5]` $\wedge \cdots \wedge$ `p[5][7][3]`[3]. This ensures row 1 (0+1), column 1 (0+1) have the value 8 (7+1) and the row 1, column 3 have the value 3,...

Given the model and the sudoku instance previously defined, *GOS* generates the following Boolean Formula, expressed in a standard format, *DIMACS*. The user could decide whether print the formula or solve the problem.

```
p cnf 4536 11361
-1 730 0
-2 730 0
-1 -2 731 0
1 -731 0
2 -731 0
1 2 -730 0
-3 732 0
-4 732 0
-3 -4 733 0
3 -733 0
4 -733 0
(...)
```

In this example it is shown a chunk of the whole output formula, that consists on 4536 variables and 11361 clauses.

[2]L∧P is a research group of the University of Girona that has developed an API to manage *Satisfiability Modulo Theories (SMT)* formulas (an extension of *SAT*). This API also integrates different solvers (among them SAT specific solvers).

[3]Note that the array indexes range from 0 to *n-1*, where *n* is the array size. So, for example, the row mapped as 0 will be the 1th in the real board.

1.3. Objectives

To reach the main purpose, many objectives must be satisfied:

- Design a programming language to improve the expressiveness when encoding any CSP to SAT.
- Implement a compiler for this language to be able to parse any input and give it semantic meaning.
- Integrate the compiler with the API implemented by L∧P group in order to use the solvers to solve the modelled problems.

My personal objectives developing this project are get a deepen knowledge in:

- how compilers work.
- SAT encoding.
- declarative and formal languages.

To achieve these goals I added some training periods to the project scheduling, in order to get the necessary knowledge to carry out this project (see Section 4).

1.4. Contextual description

To do this project, I start with the knowledge obtained from the subjects of compilers and declarative programming, and the support of the *Logic and Programming* research group of the *Universitat de Girona*.

The state of the art is the following:

- There exist many tools for text recognition to handle free-context grammars: Another Tool for Language Recognition (ANTLR) is the one used in this project.
- There exist several languages and compilers for modelling CSPs, *MiniZinc*, *ESSENCE'*, *Picat*, *WSimply*. The project purpose is define a language fully oriented to SAT.
- There exist many efficient solvers for SAT: *MiniSAT* is the one used in this project because it is already supported in the L∧P research group API.

The result obtained of this project will not only assist the research of L∧P group but also give to the community a new tool to easily model CSPs to SAT in a declarative way.

In addition it will be used in the *Declarative Programming* subject of the computer science degree at *University of Girona*.

2. Feasibility Study

2.1. Technological viability

The parameters that make the project technologically viable are:

- Existence of text recognition tools to handle context-free grammars: *ANTLR* (Parr, 2013), Bison, Yacc...
- Existence of *Constraint Programming (CP)* modelling languages: *MiniZinc*, *ESSENCE'*, *Picat*, *WSimply*,...
- Existence of efficient open source solvers for SAT: *MiniSAT* (Eén & Sörensson, 2004),...
- Experience in the L∧P research group developing languages: *WSimply* (Ansótegui et al., 2013)
- Experience in the L∧P research group developing compilers: *fzn2smt* (Bofill et al., 2012)

2.2. Economical viability

Due to the nature of this project, similarly as most software development related projects, the required set of tools is small. While many projects need additional support to be developed (scanners, sensors,...), in our case only a computer with basic software development tools is needed, so we will focus on analyzing the cost involved in using it, along with the cost of human resources.

In terms of human resources, however, a task-level analysis needs to be done as shown in the Gantt chart (see figure 4.1)

For the economical viability of the project phases, it is supposed someone working 4 hours/day 3 days/week and the following prices per hour:

Compiler designer 25€/hour[1]

Programmer 15€/hour

Analyst 20€/hour

[1]It is not an easy task to estimate the cost per hour of a compiler designer profile since it's not a typical profile but a very specialized one.

Task	Profile	Hours	Cost
Project scope definition	Analyst	48	960€
Study of LRT	Compiler designer	48	1.200€
Input format definition	Analyst	36	720€
Grammar definition	Compiler designer	48	1.200€
Symbol Table[2]	Analyst & Programmer	120	2.100€
Semantic behaviour[2]	Compiler designer & Programmer	168	3.360€
Solver integration	Programmer	24	360€
Error handling	Compiler designer & Programmer	24	480€
Output	Compiler designer & Programmer	36	720€
		552h	**11.100€**

Table 2.1: Economic human costs

Apart from the human cost also exists an infrastructure cost. To carry out this project, I have tried to minimize the price of the necessary infrastructure by using free software.

Item	Cost
MacBook Pro "Core i5" 2.3Hz Mid-2017	1.400€
ANTLR 4.0	0€
JetBrains CLion	140€[3]
Visual Studio Code	0€
Github	0€
Fork	0€
Lucidchart	9€[3]
MiniSAT	0€
LaTeX	0€
	1.549€

Table 2.2: Economic infrastructure costs

Note that the only essential equipment is the computer and the rest is free with a student license.

The total cost of the project is 11.100€ + 1.549€ = **12.649**€

[2]During this task, It is supposed to be working 6 hours/day 4 day/week
[3]Free with student licence

3. Methodology

The methodology is the process that defines how tasks are organized and distributed while developing an application.

The methodology I have followed is a kind of ***SCRUM*** based on **Iterative and Incremental Development (IID)**. New features are added to each iteration, and the product resulting from the iteration is a prototype. The prototype is the application with a subset of all the functionalities to be implemented:

1. Collect and study the requirements.
2. Study the different frameworks, libraries and data structures.
3. Choose frameworks, libraries and data structures to use.
4. General design of the system according to the requirements.
5. Iteration: Development of a set of functionalities.
6. Test what has been developed. If the test is not passed or the requirements are not satisfied, return to the last iteration of development, otherwise, pass to the next iteration.

The iterative process continues until all requirements are satisfied. The documentation of the project is done while project development. At the end of the last iteration it is verified if the documentation is valid or need to be updated.

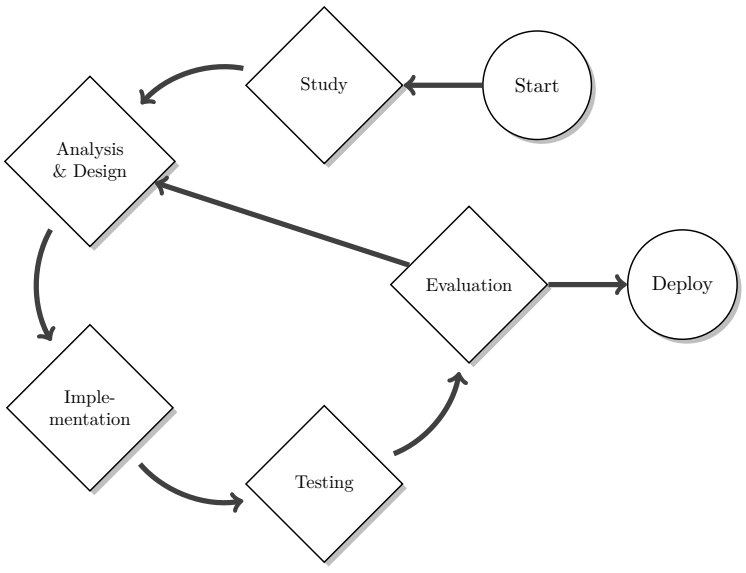


Figure 3.1: IID flow chart

The IID method that has been followed is a kind of ***SCRUM*** adapted to individual work, where sprints (described in section 4.1) had different duration. Each sprint was broken down into a subset of tasks and subtasks, which were the ones that were reviewed weekly with the supervisors. As it was impossible to do daily brief meetings, we did them weekly and they served to explain the tasks that were done, those that had to be done and make forecasts of possible setbacks.

When *sprint* was started, all the tasks and sub-tasks to perform were added on *Trello*, a platform to manage the status of those tasks:

- **Backlog**: Tasks not started.
- **Waiting**: Tasks blocked until the completion of other tasks.
- **In progress**: Tasks being done.
- **Review**: Tasks done waiting for review with the supervisors.
- **Done**: Tasks finished and reviewed.

When a sprint was finished, we did a meeting to validate the work and to schedule the sub-tasks of the next sprint.

4. Project planning

This chapter describes the planned schedule to develop the project.

4.1. Working plan

This section describes the different parts into which the project was divided (*SCRUM* sprints).

Project election and scope definition The first step consisted of defining with the supervisors what we wanted to achieve and the scope of the project.

Study of Language Recognition Tools alternatives Research about the different Language Recognition Tools (LRT) and learn how to use using its good practices.

Definition of input files format Search the best way to organize the input and the possible params of the model.

Grammar definition Read up on good practices on grammar definition. Define the grammar according to the requirements.

Lexical Analysis Research about regular expressions and define the tokens and reserved words of the language.

Syntactic Analysis Define syntactic rules draw from tokens to represent complex structures of the language.

Symbol Table implementation Implement a way to store the symbols of the language (variables, types, parameters), and its related information.

Implementation of semantic behaviour Research how to explore the input tree generated by following the syntactic rules. Give the expected semantic behaviour to the grammar.

Solver integration Research about different SAT solvers and it's implementations, and find a way to integrate into the project. After obtaining the result, integrate a SAT solver to get a result of the modelled problem.

Error Handling Analysis and study of the different ways of throwing input errors, both semantic and syntactic, and implement it.

Definition of output format Define a way to allow the user to customize the output of the modelled problem.

4.2. Schedule

The following table 4.1 shows the initial planned schedule of the project.

	2019				2020							
	December				January				February			
Tasks / Weeks	01	02	03	04	05	06	07	08	09	10	11	12
Project definition												
Study of LRT												
Input format definition												
Grammar definition												
Symbol Table												
Semantic behaviour												
Solver integration												
Error handling												
Output												

	2020											
	March				April				May			
Tasks / Weeks	13	14	15	16	17	18	19	20	21	22	23	24
Project definition												
Study of LRT												
Input format definition												
Grammar definition												
Symbol Table												
Semantic behaviour												
Solver integration												
Error handling												
Output												

Table 4.1: Gantt chart: Project tasks scheduling

5. Work framework and preliminary concepts

This chapter sets the basis on the necessary theoretical concepts that a reader of this project has to know to understand the context and the *state-of-the-art* of most of the technologies used.

The first part explains how a compiler of any language works. After this, the second part will give the theoretical base of what we want to achieve with the compiler developed in this project. The third part consists of how a SAT problem could be solved (our compiler output). Finally, the last part gives a brief overview about declarative programming.

5.1. Regular Expressions (REGEX)

Regular expressions define formal languages as sets of strings over a finite alphabet. Let σ denote a selected alphabet. Then $\emptyset$ is a regular expression that denotes the empty set and ϵ is a regular expression that denotes the set containing the empty string as its only element.

If $c \in \sigma$, then c is a regular expression that denotes the set whose only element is string c. If p and q are regular expressions denoting sets *L(p)* and *L(q)*, then

- $p \mid q$ is a regular expression denoting the set *L(p)* $\cup$ *L(q)*, where $\cup$ denotes the union.
- pq is a regular expression denoting the set of all concatenations of m and n, where $m \in$ *L(p)* and $n \in$ *L(q)*.
- p^* is a regular expression denoting closure of *L(p)*, that is, the set of zero or more concatenations of strings from *L(p)*

A language is regular if and only if some regular expression describes it.
A regular expression could also be seen as a Finite State Machine (FSM). Figure 5.1 is an example of FSM that accepts the regular expression *(bb)**, i.e. even number of b's.

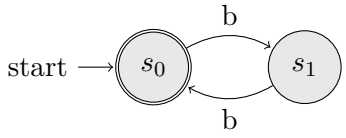


Figure 5.1: Regular expression example as a Finite State Machine

5.2. Context-Free Grammars (CFG)

Definition 5.1 (**Grammar**). *A grammar is a way of specify a language. Given a language L and a word w lets you know if w belongs to the L language or not.*

Formally a grammar G is

$$G = (\Sigma_T, \Sigma_N, s_0, R)$$

where

- $\boldsymbol{\Sigma_T}$ is the set of terminal symbols.
- $\boldsymbol{\Sigma_N}$ is the set of non-terminal symbols.
- $\boldsymbol{s_0}$ is the initial symbol of a grammar. $s_0 \in \Sigma_N$
- $\boldsymbol{R}$ is the set of production rules.

The production rules are $\alpha \rightarrow \beta$ where:

- $\alpha \ \epsilon \ (\Sigma_T \cup \Sigma_N)^+$
- $\beta \ \epsilon \ (\Sigma_T \cup \Sigma_N)^+$

Definition 5.2 (**Context-Free Grammars (CFG)**). *CFG are grammars where the context does not matter, i.e., a non-terminal symbol is always derived in a set of possible ways and always in the same set. Formally expressed as:* $\alpha \ \epsilon \ \Sigma_N$

Example 5.1 (CFG: Properly closed parenthesis chain)**.**

$S \rightarrow ()$
$S \rightarrow (\ S \ )$
$S \rightarrow S \ S$

5.3. Language Recognition

It is unquestionable the importance of being able to recognise a text that follows specific patterns in order to get some utility. For example:

- A programming language: Through written code in this language, we can get an executable file with the initial requirements.
- A protocol that recognises a particular data format in order to write, read, and share data.

The task of a compiler is usually divided into 3 steps (Stephen A. Edwards, 2007). Figure 5.2 shows a schema of those compiler steps (phases).

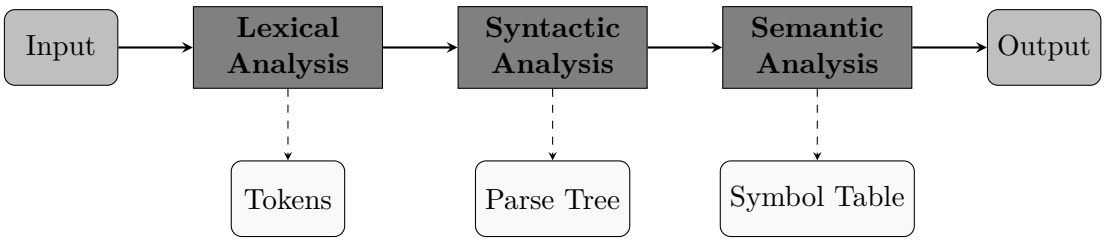


Figure 5.2: Compiler phases flow

Lexical Analysis (Scanning) The first step consists of determining if the words written in the code are correct. For example, in a natural language like English, the word "*h!ello*" would not be accepted despite the characters are part of the language. Other words as "*asdfg*" would neither pass the filter since you could not found them in the dictionary.

Take notice that this step only performs lexical analysis, so texts as "*hello hello good the*" would be accepted. Since a text is usually provided in the form of a string, this step is also used to group the relevant characters to form words in the language, forming what we will call ***tokens***.

The method used to do so is by using **Regular Expressions** (see Section 5.1). The regular expressions are a sequence of characters that define a search pattern.
For example "$(h|H)ell(o+)$" recognises the word "*hello*" with the first letter in uppercase or lowercase and a number greater than 0 of *"o"* in the end.

Syntactic Analysis (Parsing) Once we know that the words that make up the text belong to the language, we need to determine if they are presented in a logical order from a syntactic point of view. For example, although sentences can be formulated correctly in many ways, the typical format *Subject + Verb + Predicate* is an example of a syntactic rule that the text should follow to pass this stage.

So, texts like "*three blue hello look go*" would not be correct, while others like "*the ant sculpted a planetary apple*" would. Note, however, that the sentence formed may not make sense.

Since in this step it is checked if the structure of the text is correct, it is usually used to organise the tokens in a way where the structure is implicit, and help us to process in the following steps. For example, in the form of a tree, thus obtaining what we know as a syntactic tree.

The parsing phase works in a similar way than the scanning phase but using a CFG (see Section 5.2) instead of regular expressions.

Listing 5.1: Simple grammar example that recognise arithmetic expressions

```
EXPR     => EXPR + TERM
EXPR     => EXPR - TERM
EXPR     => TERM
TERM     => TERM * FACTOR
TERM     => TERM / FACTOR
TERM     => FACTOR
FACTOR   => (EXPR)
FACTOR   => identifier
```

The generated tree of the example grammar 5.1 respects the associativity and precedence of arithmetic expressions.

Semantic Analysis Once the text has been verified to be written correctly and structured logically, it remains to be seen if it makes sense, and what we want to do with it. This step is less general than the previous ones, as the more specific aspects of each particular language come into play, and there is no secure method for deciding whether a text makes sense or not.

This step usually consists of visiting the structure generated during the parsing, and performing the necessary checks and actions during the process. Apart from this, in this phase it is used to fill the *Symbol Table*, that it is a data structure where each identifier (a.k.a. *symbol*) in a program's source code is associated with information relating to its declaration.

The follwing Figure 5.3 is a schema of the explained compiler phases.

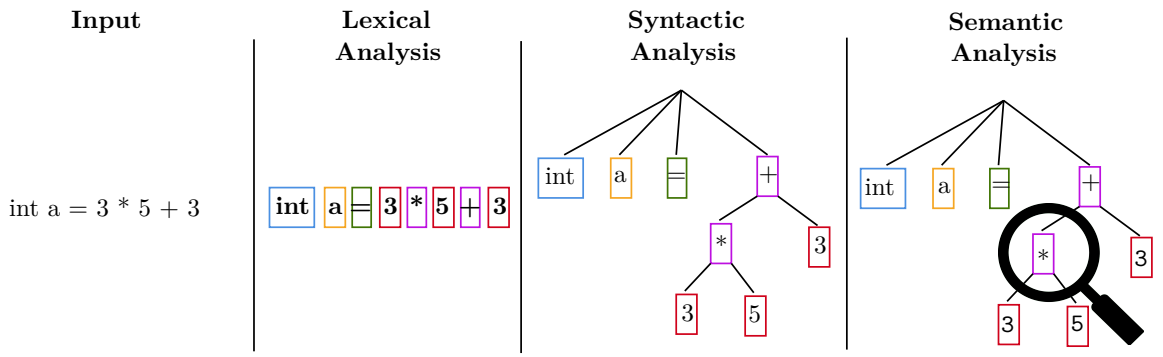


Figure 5.3: Compiler phases graphic chart

5.3.1. Error recovery strategies

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

5.4. Constraint Satisfaction Problems (CSP)

CSP are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria.

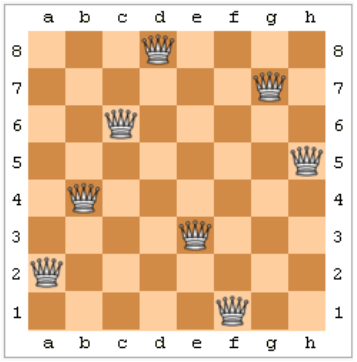
Formally, a constraint satisfaction problem is defined (Russell & Norvig, 2010) as a triple $\langle X, D, C \rangle$, where:

$X = \{X_1, \ldots, X_n\}$ is a set of variables,
$D = \{D_1, \ldots, D_n\}$ is a set of their respective domains of values, and
$C = \{C_1, \ldots, C_m\}$ is a set of constraints

Each variable X_i can take one of the values in the nonempty domain D_i. Evey constraint $C_j \in C$ is in turn a pair $\langle t_j, R_j \rangle$, where $t_j \subset X$ is a subset of k variables and R_j is a k-ary relation on the corresponding subset of domains D_j. An evaluation of the variables is a function from a subset of variables to a particular set of values in the corresponding subset of domains. An evaluation v satisfies a constraint $\langle t_j, R_j \rangle$ if the values assigned to the variable t_j satisfies the relation R_j.

An evaluation is consistent if it does not violate any of the constraints. An evaluation is complete if it includes all variables. An evaluation is a solution if it is consistent and complete; such an evaluation is said to solve the constraint satisfaction problem.

Example 5.2. *Procaccia (2008) The eight queens puzzle is the problem of putting eight chess queens on an 8 × 8 chessboard, such that none of them is able to capture any other using the standard chess queen's moves. The queens must be placed in such a way that no two queens would be able to attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.*
Variables x1..x8 represent the location of the queens. D = (1, a)..(8, h). An example constraint for queens 1 and 2 is: $C_{1,2} = \langle(1,a),(2,c)\rangle$, $\langle(1,a),(2,d)\rangle$, $\langle(1,b),(2,d)\rangle$,...



In other words, a CSP consists of a set of variables, ranging on domains, and subject to constraints.
In this project, we will focus on one of the ways of modeling and solving CSP: Boolean Satisfiability (SAT).

5.4.1. Boolean Satisfiability (SAT)

One methodology for solving Constraint Satisfaction Problems (CSP) (treated in this project) relies on the conversion into SAT problems. The advantage is the wide availability of free and efficient SAT solvers.

A SAT problem (Bessière, 2007) contains a formula built on a set of variables, which can take only value *true* (or 1) and *false* (or 0). This formula is often required to be in Conjunctive Normal Form (CNF). A solution to SAT problem (a model) is an assignment of values *true/false* to the logical variables, such that the formula evaluates to true.

5.4.1.1. Conjunctive Normal Form (CNF)

The CNF Satisfiability Problem (CNF-SAT) is a version of the Satisfiability Problem, where the Boolean formula is specified in CNF, that means that it is a conjunction of clauses, where a ***clause*** is a disjunction of literals, and a literal is a variable or its negation. For example:

$$(x_1 \vee x_2) \ \wedge \ (\neg x_2 \vee x_3 \vee x_4) \ \wedge \ (\neg x_1 \vee x_4) \tag{5.1}$$

Here x_1, x_2, x_3, x_4 are Boolean variables to be assigned, $\neg$ means negation (logical not),$\vee$ means disjunction (logical or), and $\wedge$ means conjunction (logical and). One may note that the formula 5.1 is satisfiable, because on $x_1 = true$, $x_2 = false$, $x_3 = false$, and $x_4 = true$ it takes on the value true. If a formula is not satisfiable, it is called unsatisfiable, that means that it takes on the value false on any combination of values of its variables.

Any propositional formula can be transformed to CNF in a polynomial time.

5.5. SAT solving

To satisfy a CNF, all the clauses must be satisfied. A first inefficient approach to find a satisfying assignment (a model) could be using a truth table. The table 5.1 shows an example of how to solve the formula $(\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})$ using a truth table.

x_1	x_2	x_3	$(\overline{x_1} \vee x_2 \vee \overline{x_3})$	$(x_1 \vee \overline{x_2} \vee \overline{x_3})$	$(\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})$
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	1	1
0	**1**	**1**	1	**0**	**0**
1	0	0	1	1	1
1	**0**	**1**	**0**	1	**0**
1	1	0	1	1	1
1	1	1	1	1	1

Table 5.1: Example of SAT solving through a truth table

In the example 5.1 there exist two interpretations, for $x_1 = 0, x_2 = 1, x_3 = 1$ and $x_1 = 1, x_2 = 0, x_3 = 1$ that falsify the formula, since this assignation does not satisfy the clauses. The rest of the interpretations are valid (a model) of the formula.

The number of interpretations is exponential in the number of variables, and therefore constructing the whole truth table is extremely inefficient. Most SAT-solvers implement more efficient algorithms based on backtracking schemes. The fundamentals of these algorithms are explained below to get a basic idea of how they work but no deep details are given since the goal of this project is not implementing a solver.

Resolution Rule Is a solid rule that generates a new clause (C) that is logic consequence of the conjunction of two origin clauses ($A \wedge B$).

$$\frac{\mathbf{A} : x \vee a_1 \vee \cdots \vee a_n, \quad \mathbf{B} : \overline{x} \vee b_1 \vee \cdots \vee b_m}{\mathbf{C} : a_1 \vee \cdots \vee a_n \vee b_n \vee \cdots \vee b_m} \tag{5.2}$$

Figure 5.4: Resolution rule

Every model (solution) of $A \wedge B$ is a model of C. If an interpretation does not satisfy C neither satisfies $A \wedge B$. There exist methods to determine the satisfiability of a formula based only on applications of the resolution rule. These methods are not used in practice, since they have many drawbacks such as the difficulty to retrieve a model if the instance is satisfiable. However, the resolution rule is crucial for modern sat solvers, since it is the inference mechanism used in the learning process of CDCL algorithm (described below).

Backtracking It is an approach using brute force. As long as there are variables to assign, we try an assignment, we check the correctness, and continue or backtrack.

Having this clauses: $C_1 : \overline{x_1} \vee x_4 \quad C_2 : \overline{x_2} \vee \overline{x_4} \quad C_3 : \overline{x_3} \vee x_4 \quad C_4 : x_1 \vee x_5 \quad C_5 : x_1$

Algorithm steps:

1. While remain variables to assign and there are not violated clauses, decide a literal.
2. When a clause C is violated:
   a) If we have some decision to invert, backtrack to the last decision.
   b) If we do not have any decision, unsatisfiability is proved.
3. If we have assigned all the variables without violating any clause, we have proved the satisfiability by obtaining a model.

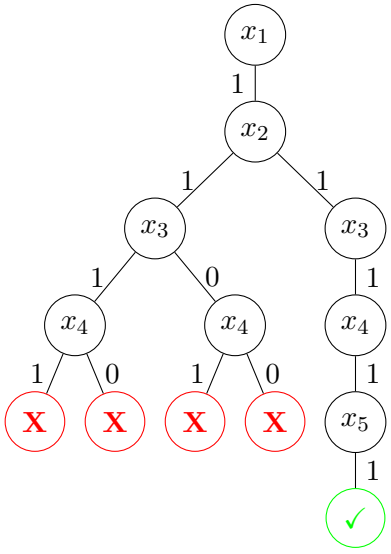


Figure 5.5: Backtracking tree example

DPLL This method is an improvement of the backtracking method. It can be divided in two parts:

- **Unit propagation** Uses two rules to simplify the set of clauses S:
  1. A unitary clause $L \in S$ is chosen and we apply consecutively the following two rules:
     - **Unit subsumption:** All clauses subsumed by L are removed from S, i.e., those that contain the literal L (including clause L itself).
     - **Unit resolution:** It is removed $\overline{L}$ from all the clauses of S.
  2. Repeat until there are no more unit clauses in S.
- **Division:** After the simplification process, if the algorithm has not stopped, then S does not contain unit clauses. So, it is selected a literal $L \in S$ and constructed the following sets: $S \cup L$ and $S \cup \overline{L}$:
  1. It is applied recursively unit propagation to $S \cup L$ (selecting necessarily the unit clause L)
  2. If $S \cup L$ is unsatisfiable, apply the process to $S \cup \overline{L}$.

Conflict-Driven Clause Learning (CDCL) Besides using DPLL, CDCL solver involves a number of additional key techniques:

- Learning new clauses from conflicts during backtrack search.
- Exploiting structure of conflicts during clause learning
- Conflict driven decision heuristic.

- Using lazy data structures for the representation of formulas.
- Periodically restarting backtrack search.
- Additional techniques, including deletion policies for learnt clauses.

5.6. Declarative programming

Definition 5.3 (Declarative programming)**.** *(Wikipedia, 2020a) Declarative programming contrasts with imperative and procedural programming. Declarative programming is a non-imperative style of programming in which programs describe their desired results without explicitly listing commands or steps that must be performed. Functional and logical programming languages are characterized by a declarative programming style. In logical programming languages, programs consist of logical statements, and the program executes by searching for proofs of the statements.*

The different types of declarative languages are:

Constraint programming Constraint programming states relations between variables in the form of constraints that specify the properties of the target solution. The set of constraints is solved by giving a value to each variable so that the solution is consistent with the maximum number of constraints. Constraint programming often complements other paradigms: functional, logical, or even imperative programming.

Domain-specific languages Well-known examples of declarative domain-specific languages (DSLs) include the yacc parser generator input language, QML, the Make build specification language, Puppet's configuration management language, regular expressions, and a subset of SQL (SELECT queries, for example). Many markup languages such as HTML, MXML, XAML, XSLT or other user-interface markup languages are often declarative. HTML, for example, only describes what should appear on a webpage - it specifies neither control flow for rendering a page nor the page's possible interactions with a user.

Logic programming Logic programming languages such as Prolog state and query relations. The specifics of how these queries are answered is up to the implementation and its theorem prover, but typically take the form of some sort of unification. Like some functional programming, many logic programming languages permit side effects, and as a result are not strictly declarative.

Functional programming Is a programming paradigm that treats computing as a process of applying functions, avoiding changeable data with its state changes. Functional programming is based on lambda calculus. The difference between mathematical function and the concept of function used in imperative programming is that imperative functions have side effects, changing the value of objects. already calculated, showing a lack of referential integrity, because the same expression can have different values at different times, depending on the state of execution of the program. Examples of functional programming languages are Haskell, Scala,...

6. System requirements

System requirements can be split into functional and nonfunctional requirements.

6.1. Functional requirements

The functional requirements are the following:

- The user must be able to model a CSP-SAT using the language defined.
- The user must be able to set the parameters values of the model through an input file.
- The user must be notified of all lexical, syntactic or semantic errors in their model or input file.
- The user must be able to get a solution of the model, if exists.
- The user must be able to get the generated *CNF* of the modelled problem.
- The user must be able to define a custom output.
- The user must be able to generate the mathematic documentation of the model in LaTeX-format.

6.2. Nonfunctional requirements

6.2.1. Hardware requirements

OS Tested only in MacOS Catalina 10.15, but it should should work on Windows or any distribution of Linux.

CPU Any CPU should be compatible, but it is recommended a powerful one.

RAM Any RAM size is compatible, but is recommended a minimum of 8GB.

6.2.2. Software requirements

The software required to execute the project is the following [1]:

- Another Tool for Language Recognition (ANTLR) 4
- *CMake*
- *GNU Compiler Collection (GCC) 4.8*

[1] The code used for installing the tool already includes the L∧P research group API (see Section 9.5) and the binaries of the SAT solver (see Section 7.4).

7. Studies and decisions

7.1. *Another Tool for Language Recognition (ANTLR)*

ANTLR is a parser generator developed and maintained by Terence Parr at the University of San Francisco for reading, processing, executing, or translating structured text or binary files. The last stable version is *ANTLR 4* and it is useful for constructing:

- Lexers (lexical analysis)
- Parsers (syntactic analysis)
- Tree walkers (used in semantic analysis)

ANTLR is a *Java* application but it can generate *tree walkers* in different target languages: *Java*, *C++*, *C#*, *Python*, *JavaScript*,...

7.1.1. Lexical Analysis

The section 5.3 explains how Language Recognition Tools (LRT) work. Here is explained how ANTLR do this and its added features. Table 7.1 shows the lexical rules of ANTLR.

Expression	Result
'c'	Recognizes the character c
(e)	Recognizes e
e_1 e_2	Recognizes e_1 and then e_2
e_1 | e_2	Recognizes e_1 or e_2
'c1'..'c2'	Recognizes all ASCII characters between c_1 and c_2
~'c'	Recognizes a different character from c
~('c_1' | 'c_2' | 'c_3')	Recognizes a character different than c_1, c_2, and c_3
~('c_1'..'c_2')	Recognizes a character different than the characters between c_1 and c_2
e?	Recognizes e or nothing
e*	Recognizes zero or more occurrences of e
e+	Recognizes one or more occurrences of e
'abcde'	Recognizes string abcde
'\"	Recognizes a apostrophe
'\n'	Recognizes a new line
'\r'	Recognizes a carry return
'\t'	Recognizes a tab
' '	Recognizes a whitespace
EOF or '@'	Recognizes the end of file
.	Recognizes any character

Table 7.1: ANTLR lexical recognition rules

The part of grammar that determines lexical rules is made up of associations between token name (identifier) and regular expressions. This way, when a string matches a regular expression, it is transformed into the corresponding token.

It is possible that the same string fits the definition of more than one token; in this case it will be transformed to the first token corresponding to a regular expression that accepts the string.

The Figures 7.1 and 7.2 show two examples of valid and invalid input to lexical rules.

```
TWO : '2';
DIGIT: '0'..'9';
```

```
8 2 2 3
```

```
DIGIT TWO TWO DIGIT
```

Figure 7.1: Valid example of lexical rules, input and output

```
TWO : '2';
DIGIT: '0'..'9';
```

```
8 2 2 3 hello
```

```
ERROR - No rules
fitting with 'hello'
```

Figure 7.2: Invalid example of lexical rules, input and output

7.1.2. Syntactic Analysis

The other set of rules that ANTLR have are the syntactic ones: these determine, once the lexical analysis has transformed the entry into a chain of tokens, how they are structured, in order to generate tree that represents the input file we processed.

The production rules in table 7.2, are written with EBNF notation:

Expression	Result
T	Recognizes the token T
(e)	Recognizes e
e_1 e_2	Recognizes e_1 and then e_2
e_1 | e_2	Recognizes e_1 or e_2
e?	Recognizes e or nothing
e*	Recognizes zero or more occurrences of e
e+	Recognizes one or more occurrences of e
EOF or '@'	End of file

Table 7.2: ANTLR syntactical recognition rules

To differentiate a syntactic rule from a lexicon, the name of a token is usually indicated in capital letters and the rule in lowercase.

```
DIGIT: '0'..'9';
sum : sum '+' DIGIT
      | DIGIT;
```

```
1+2+3
```

```
sum + DIGIT(3)
    | sum + DIGIT(2)
        | DIGIT(1)
```

Figure 7.3: Example of ANTLR syntactic rules, input and output

Many LRT restrict the use of certain patterns of syntactic rules in grammar. For example, in Figure 7.3, the *sum* rule is recursive to the left, i.e., the first token or rule tries to match itself. In some implementations, this would cause an infinite loop in the code. The version of ANTLR used in the project supports this pattern, among other advantages. The figure 7.4 shows the parse tree generated of the example in figure 7.3

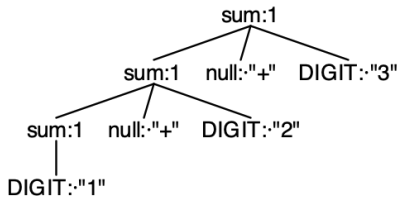


Figure 7.4: Example of ANTLR parser tree result

7.1.3. Semantic Analysis

Once our grammar is defined, we can run ANTLR. It will generate source code files, in our case C++ (see Section 7.1.3.2 for more information on target selection).

These files consist of a set of objects appropriate for performing semantic analysis; in essence, a tree where the nodes contain all the necessary information about the input file, and functions to traverse such a tree.

Assuming you want to process a grammar called *Test*, described in a file called, necessarily, *Test.g4*, this files are always generated:

- ***Test.tokens***: Mapping between token identifiers and constant int values.
- ***TestLexer.tokens***: Similar to *Test.tokens* but including extra tokens with the purpose of manage *Lexer* environment.
- ***TestLexer.h and TestLexer.cpp***: Lexical analysis related methods.
- ***TestParser.h and TestParser.cpp***: Syntactic analysis and parsing related methods.

Two more files are also generated, but they depend on how you want to explore the generated parsing-tree for the semantic analysis. ANTLR offers two alternatives: *Visitors* or *Listeners*.

7.1.3.1. Listeners vs Visitors

This section explains the main differences between the ways that *ANTLR* gives to explore the generated parse-trees.

Listeners If this approach is chosen, the syntax tree is visited automatically. The generated classes provide an interface of methods that are called when interacting with a node, either at the time of entry or at the time of exit. The figure 7.5 shows a flow example of an *ANTLR* listener.

Listing 7.1: ANTLR generated **listener** for Figure 7.3 grammar

```
class  TestBaseListener : public TestListener {
public:
  virtual void enterSum(TestParser::SumContext * /*ctx*/) override {}
  virtual void exitSum(TestParser::SumContext * /*ctx*/) override {}

  virtual void enterEveryRule(antlr4::ParserRuleContext*) override {}
  virtual void exitEveryRule(antlr4::ParserRuleContext*) override {}
  virtual void visitTerminal(antlr4::tree::TerminalNode*) override {}
  virtual void visitErrorNode(antlr4::tree::ErrorNode*) override {}
};

```

Figure 7.5: *ANTLR* listener schema

Visitors This other method involves, as the name suggests, explicitly visiting the nodes in the tree. The process is not automatic, so it is usually more flexible in controlling which parts to visit, and which ones to ignore; by using functions such as *visit(ParseTree* tree)* or *visitChildren(RuleNode* node)*.
Visitors also allows to return a value after visiting a node and get it on the parent avoiding create global variables to store data. The figure 7.6 shows a flow example of an *ANTLR* vistor.

Listing 7.2: ANTLR generated **visitor** for Figure 7.3 grammar

```
class  TestBaseVisitor : public TestVisitor {
public:
  virtual antlrcpp::Any visitSum(TestParser::SumContext*ctx) override {
    return visitChildren(ctx);
  }
};

```

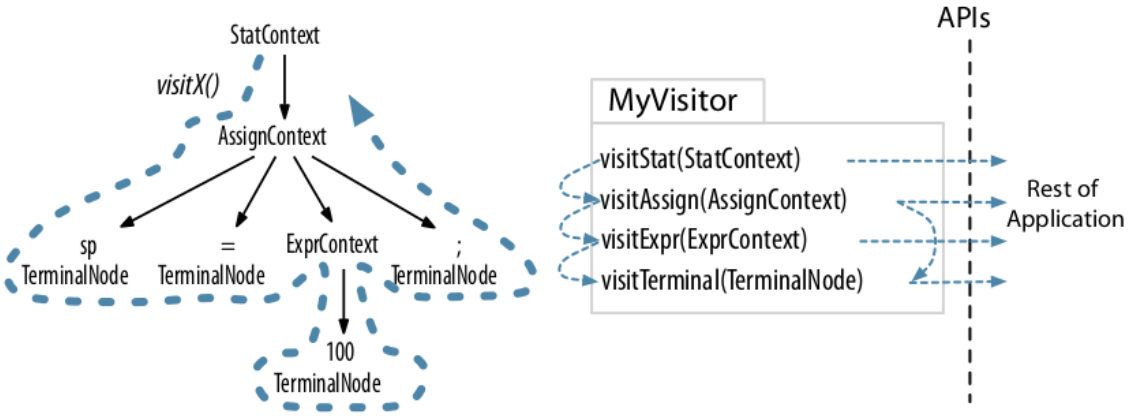


Figure 7.6: *ANTLR* visitor schema

The *context* elements generated as parameters of visitor and listener methods describe the current node. It has information about the real values of the node tokens and also information about the row and column of the current node in the input file, e.g., to make error handling easier.

The second paradigm, *Visitors*, has been chosen for this project. It is the one that best suits our needs, as we will have to go through the tree more than once and in different orders.

Thus, two additional classes are generated in out example *Test* grammar:

- ***TestBaseVisitor***
- ***TestVisitor***

The last step is create a class that extends and overwrites the base visitor's methods and give them the logic wanted according to the semantic meaning you want to achieve.

7.1.3.2. Target: *C++* vs *Java*

ANTLR allows targeting with multiple languages, although Java is the default.

One of the main differences between Java and C++ is that the first was born as an interpreted language while the second as a compiled language. Compiled languages are translated into machine code trough a compiler. This process generates a file that can be directly executed by the CPU. Interpreted languages are compiled in a platform independent language (bytecode), which can be executed only by means of an interpreter (e.g. JVM).
Apart from that, Gherardi et al. (2012) conclude in his report that "*The results obtained (...) have shown that Java is from 2.72 to 5.61 times slower than C++.*"

Efficiency is important in problem-solving tools, so C++ target is the best option for this project.

7.1.4. *ANTLR* vs other LRT

Apart from ANTLR, there exists other LRT like *Bison, yacc, flex,....* Here is a brief summary of the comparison between them and ANTLR (Federico, 2019):

Stability and Development of New Features

- *Flex* and *Bison* are stable and maintained software but there is no active development. C++ support can be of limited quality.
- *ANTLR* is actively developed and new features are added periodically.

Separation between Grammar and Code

- Flex and Bison maintain an old-school design with little support for readability or productivity.
- *ANTLR* is a modern parsing generator tool with a design that favours a portable grammar, usable for multiple target languages.

Features of lexing

- *Flex* supports regular expressions to define rules, which works for most elements, but adds complexity.
- *ANTLR* supports context-free expression to define rules, which simplifies defining some common elements.

Features of parsing

- *Bison* supports two parsing algorithms that cover all ranges of performance and languages. It gives cryptic error messages.
- *ANTLR* supports one algorithm that works for all languages with usually a great performance.

Documentation

- *Flex* and *Bison* have smaller and fractured communities, but they have good documentation
- *ANTLR* has a great community and a good documentation.

7.2. *JSON* as input format

JSON (acronym for *JavaScript* Object Notation) is an open text-based standard designed for human-readable data exchange.

It derives from the JavaScript script language, to represent simple data structures and associative lists, called objects. Despite its relationship to JavaScript, it has implementations for much of the programming language.

Listing 7.3: *JSON* grammar

```
grammar JSON;

json : value;

obj : '{' pair (',' pair)* '}'
    | '{' '}';

pair : STRING ':' value;

arr : '[' value (',' value)* ']'
    | '[' ']';

value
   : STRING
   | NUMBER
   | obj
   | arr
   | 'true'
   | 'false'
   | 'null';
```

The main reason of using *JSON* as input format is because it is a widely used standard and it has a simple grammar. Moreover, JSON easily represents tuples and arrays, which are also the main data structures used in *BUP* language.

```
{
    "id" : 10
    "fruit": "Apple",
    "sizes": ["Large", "Small"],
    "color": "Red"
}
```

7.3. Allow *structs* as data type

Initially the idea arose that our modelling language could be object-oriented. It was discarded because currently there not exist any object-oriented declarative modelling language and it would surely require a much longer research task than the estimated for an end-of-grade

project.

But it was an idea that we liked and we decided to do a first iteration on this path allowing to declare structs. Currently there is no declarative constraint programming language that allows declaring structs as a data type.

In this language *structs* are called *entities* (see Section 8.1.5.2 for further information).

7.4. *MiniSAT* as SAT solver

The project objective is to define a new declarative modelling language oriented to *SAT* and it not focused on the solving part. There already exist many efficient SAT solvers for that.

Luckily, one of my supervisors, Jordi Coll, member of the L∧P research group at the *University of Girona*, developed a C++ library that aggregates many Satisfiability Modulo Theories (SMT) solvers (an extension of SAT that allows predicates more expressive than propositional formulas) and defines a generic interface to interact with them (see 9.5 for more information about the library).

Currently, the only strictly-SAT solver integrated in this library is MiniSAT[1] (Eén & Sörensson, 2004), and therefore it is the one used in this project. However, which solver is used is completely transparent to the user of the library, and any new SAT solver integrated in the library will also be available in *GOS*.

7.5. Use of *CMake* to build the project

One of the project goals is implement a compiler that could be used in any computer architecture and OS.
CMake is a build-system generator (not a build system, though). It generates input files for build generators (such as *make*, *ninja*, *xcode*,...), is cross-platform and it has *C++* support.

CMake does proper dependency management and prevents manually managing include directories (especially transitive include directories), linker command lines, etc. (see Appendix A Install and Run Instructions to check how easy it is to build the project using *CMake*).

[1]The *API* also supports *Yices* and other solvers, but the binaries are not included in the project code.

8. *BUP*: Language Specification

This chapter contains the specification of the *BUP* programming language, the language of *GOS*. *BUP* is a *SAT*-oriented declarative *CSP* modelling programming language. Similarly as we do with imperative languages, we want models written in *BUP* language to be as much reusable as possible, i.e., we want to separate the problem definition from the data of a particular instance.

For this reason we deal with CSPs using two distinct files: the **model file** (see 8.1), which describes the semantics of the problem at hand by means of defining the needed variables, parameters and constraints, and the **parameters file** (see 8.2), which describes the values of the parameters of the particular instance we want to solve.

8.1. Model file

The model file structure is divided in four blocks:

- Entity definition block, where entities are defined.
- Viewpoint block, where variables and parameters are declared.
- Constraints block, where constraints are defined.
- Output block, where custom output is defined.

Listing 8.1: *BUP*: Basic model file structure

```
1 <entityDefinitionBlock>?
2 <viewpointBlock>
3 <constraintsBlock>
4 <outputBlock>?
```

8.1.1. Entity definition block

This block is used to define the new entities to be used at the viewpoint.

Listing 8.2: *BUP*: Entity definition block

```
1 <<entityDefinition>;>*
```

8.1.2. Viewpoint block

This block is used to define the variables and parameters used to model the problem.

Listing 8.3: *BUP*: Viewpoint block

```
viewpoint : <<
      <variableDeclaration>
    | <parameterDeclaration>
    | <entityDeclaration>
    | <arrayDeclaration>
>;>*
```

8.1.2.1. Variable declaration

Listing 8.4: *BUP*: Variable declaration

```
var bool? <ident>
```

8.1.2.2. Parameter declaration

Listing 8.5: *BUP*: Parameter declaration

```
param <int | bool> <ident>
```

8.1.2.3. Entity declaration

Listing 8.6: *BUP*: Entity declaration

```
<entity_ident> <ident>
```

8.1.2.4. Array declaration

Array indexes range from 0 to n-1, where n is the array size.

Listing 8.7: *BUP*: Array declaration

```
<
      <varDeclaration>
    | <paramDeclaration>
    | <entityDeclaration>
> <[<int_expr>]>*
```

8.1.3. Constraints block

Listing 8.8: *BUP*: Constraints definition block

```
constraints: <<constraint>;>*
```

8.1.4. Output block

Listing 8.9: *BUP*: Output block

```
output: <<string>;>*
```

8.1.5. Data

BUP has two types of data, according to its nature:

- **Parameters**: Instance related. Used to describe the instance specification.
- **Variables**: Model related. We must differentiate this variables from the imperative languages variables. SAT variables are always boolean, but you cannot do arithmetic operations or assign them. Instead, you can define constraints over variables, expressed as propositional formulas. They are strictly designed for defining the SAT model and their value can only be retrieved in the *output-block* (see 8.1.4) specification, where their value can be accessed similarly as is done with boolean parameters.

Both, parameters and variables, could be declared individually or using data structures: multidimensional arrays or defined types (entities).

8.1.5.1. Basic types

The basic types are those with which you can define **parameters**:

int Represents an integer value.

bool Represents a boolean. It can take values *true* or *false*.

8.1.5.2. Defined types: Entities

BUP allows defining new entities consisting of a tuple of data elements. These data elements can contain both parameters and variables. The only restriction is that entity types used in the definition of an element must have been declared previously.

Listing 8.10: *BUP*: Entity definition

```
<identifier> {
  <
      <parameter declaration>
    | <variable declaration>
    | <entity declaration>
    | <array declaration>
  >+
}
```

8.1.5.3. *n*-dimentional arrays

It is allowed to define *n*-dimentional arrays of any basic type, variable or entity. The arrays index range from 0 to *m-1*, where *m* is the length of the indexed dimension of the array.

8.1.6. Identifiers

The identifiers are words without whitespaces. They can only contain alphanumeric characters and underscores. Identifiers cannot start with a number.

Listing 8.11: *BUP*: Identifiers

```
<'a'..'z' | 'A'..'Z' | '_'> <'a'..'z' | 'A'..'Z' | '_' | '0'..'9'>*
```

8.1.7. Comments

Code could be commented using the following two methods:

- `//` to comment one line.
- `/*` ... `*/` to comment multiple lines.

8.1.8. Expressions

Expressions are only permitted between parameters of basic types (see Basic types):
An expression could be:

- A value of basic type.
- A data access (see 8.1.9)
- A list aggregation operation (see 8.1.10.1)
- An operation between one or more expressions. In table 8.1 we can find all implemented operators with their associativity and types. They are also sorted according to their priority.

Operator	Associativity	Input type	Output type
`not`	-	*bool*	*bool*
`/`, `%`, `*`	Left	*int*	*int*
`+`, `-`	Left	*int*	*int*
`<`, `<=`, `>`, `>=`	Left	*int*	*bool*
`==`, `!=`	Left	*int or bool*	*bool*
`and`, `or`	Left	*bool*	*bool*
if-then-else	Right	*int or bool*	*int or bool*

Table 8.1: *BUP*: Expresion operators priority

- An *if-then-else* structure, expressed as:

Listing 8.12: *BUP: Range list*

```
<bool_expr> ? <expr1> : <expr2>
```

If `bool_expr` evaluates `true` resolves `expr1` else `expr2`.

8.1.9. Data access

Accesses can be used to retrieve a value (in parameters or in variables when they already have a value, i.e. in the output section), or to impose constraints on variables.

8.1.9.1. Identifier access

Listing 8.13: *BUP*: Identifier access

```
<ident>
```

8.1.9.2. Array index access

Listing 8.14: *BUP*: Array index access

```
 <
      <ident>
    | <entityAttrAccess>
    | <arrayIndexAccess>
> [<int_expr>]
```

8.1.9.3. Matrix row access

BUP allows generating a list by accessing a row of a matrix using the operator _ (see section 8.1.10)

Listing 8.15: *BUP*: Matrix row access

```
 <
      <ident>
    | <entityAttrAccess>
    | <arrayIndexAccess>
> [_] <[<int_expr>]>*
```

8.1.9.4. Entity attribute access

Listing 8.16: *BUP*: Entity attribute access

```
 <
      <ident>
    | <entityAttrAccess>
    | <arrayIndexAccess>
>.<ident>
```

8.1.10. Lists

Lists can be used to generate ranges to loop over arrays in *forall* structures. It is also allowed applying constraints over lists of variables, i.e., cardinalities (see 8.1.11.2), *and* constraints (see 8.1.11.1.3), *or* constraints (see 8.1.11.1.4),...

A list could be:

- **Range list**

Listing 8.17: *BUP*: Range list

```
<int_expr>..<int_expr>
```

  - `1..5` generates the list *[1,2,3,4,5]*.
  - `1..15/5` generates the list *[1,2,3]*.

- **Comprehension list**

Listing 8.18: *BUP*: Comprehension list

```
[<<expr> | <clause>> | <ident> in <list> (,<ident>} in <list>)* <where ↩
    ↪ <bool_expr>>?]
```

  - `[i | i in 1..5]` generates the list *[1,2,3,4,5]*.
  - `[i*j | i in 1..3, j in 1..3]` generates the list *[1,2,3,2,4,6,3,6,9]*.
  - `[i*j | i in 1..3, j in 1..3 where i < j]` generates the list *[2,3,6]*.

- **Explicit list**

Listing 8.19: *BUP*: Explicit list

```
[<<expr> | <clause>> <, <<expr> | <clause>>*]
```

  - `[1,2,3,4,5]` generates the list *[1,2,3,4,5]*.
  - `[1,2*4,7,14/2+1,76]` generates the list *[1,8,7,8,76]*[1].
  - `[a | b, a, c -> d]` generates the list $[a \vee b,\ a,\ \overline{c} \vee d]$.

- **One-dimensional array**

Listing 8.20: *BUP*: Comprehension list

```
<matrixRowAccess>
```

Given an `int arrayX[3]`:

  - `arrayX` generates the list [arrayX[0], arrayX[1], arrayX[2]]

Given an `int arrayX[3][3][3]`:

  - `arrayX[0][_][0]` generates the list *[arrayX[0][0][0], arrayX[0][1][0], arrayX[0][2][0]]*
  - `arrayX[_][1][0]` generates the list *[arrayX[0][1][0], arrayX[1][1][0], arrayX[2][1][0]]*

8.1.10.1. List Aggregation Operators

8.1.10.1.1. `length` The `length` operator returns the size of a list.

[1] All list elements must have the same type.

Listing 8.21: *BUP*: List aggregate `length`

```
length( <list> )
```

8.1.10.1.2. `sum`

The `sum` operator returns the sum of a list of `int`.

Listing 8.22: *BUP*: List aggregate `sum`

```
sum( <list_int> )
```

8.1.10.1.3. `max`

The `max` operator returns the maximum number of a list of `int`.

Listing 8.23: *BUP*: List aggregate `max`

```
max( <list_int> )
```

8.1.10.1.4. `min`

The `min` operator returns the minimum number of a list of `int`.

Listing 8.24: *BUP*: List aggregate `min`

```
min( <list_int> )
```

8.1.11. Constraints

A constraint can be:

- Propositional Formula
- Cardinality Constraint
- `forall` structure
- `if` structure

8.1.11.1. Propositional Formula

Although the GOS compiler can recognize syntactically any propositional formula, the BUP language only permits those formulas with trivial translation to CNF (see 5.4.1.1), more precisely, formulas that can be translated to a linear number of clauses without adding auxiliary variables. We have taken this decision for the first version of BUP[2], but since GOS's parser already supports any kind of propositional formula, it would be easy to support any propositional formula in future versions if desired.

For instance, given the following propositional formula

```
a & b & c | d & e
```

A translation to *CNF* without introducing auxiliary variables would be

```
(a | d) & (a | e) & (b | d) & (b | e) & (c | d) & (c | e)
```

It generates a quadratic number of clauses. This propositional formula is **not** semantically in *BUP* language. Given this other formula:

```
a & b & c | d
```

The translation to *CNF* would be

```
(a | d) & (b | d) & (c | d)
```

It generates a linear number of clauses. This propositional formula is allowed in *BUP* language.

Therefore, the *GOS* compiler does a semantic check to ensure that the formulas contained in a *BUP* file fulfil certain properties. The semantic rules that *GOS* applies over Boolean operations are described in the following subsections.

[2]This decision was taken mainly for the sake of a clear correspondence with the BUP file and the generated SAT formulas. Also, good reformulations of more complex formulas involve challenges such as detection of common sub-expressions, which are out of the scope of this project.

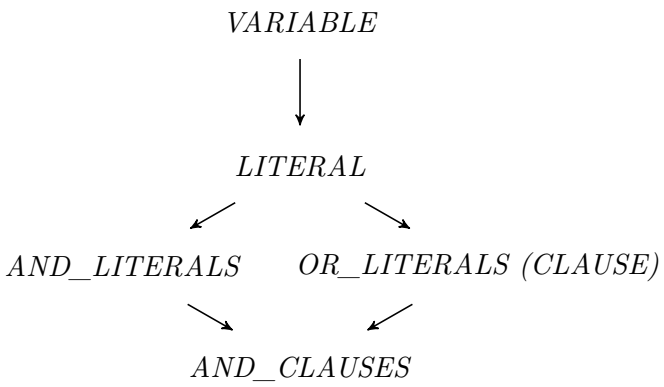


Figure 8.1: *BUP*: Propositional formulas operands

Figure 8.1 shows how different operands could be treated when constructing Boolean Formulas:

- *LITERAL* is a *VARIABLE* or its negation.
- *AND_LITERALS* is an and operation between literals. A *LITERAL* is a particular case of *AND_LITERALS* where there is only one literal.
- *OR_LITERALS* is an or operation between literals. A *LITERAL* is a particular case of *OR_LITERALS* where there is only one literal.
- *AND_CLAUSES* is an and operations between clauses. *OR_LITERALS* is a particular case of *AND_CLAUSES* where there is only one clause. *AND_LITERALS* is a particular case of *AND_CLAUSES* where all the clauses are unitary.

These are the operators and their precedence:

Name	Operator	Associativity	Precedence
Negation	`!`	-	1
And	`&`	Left	2
Or	`|`	Left	3
Implication	`->`	Left	4
Double Implication	`<->`	Left	5

Table 8.2: *BUP:* Propositional formula operators

8.1.11.1.1. Variable

Access to a declared variable:

Listing 8.25: *BUP*: Variable

```
1 <identAccess>
```

8.1.11.1.2. Negation The negation operator has the following truth table:

a	!a
0	**1**
1	**0**

Table 8.3: Negation truth table

The allowed operations using `!` operator are:

Expression	Result
! *LITERAL*	*LITERALS*
! *AND_LITERALS*	*OR_LITERALS*
! *OR_LITERALS*	*AND_LITERALS*

Table 8.4: *BUP*: Allowed operations with !

8.1.11.1.3. *And* The and operator has the following truth table:

a	b	a & b
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

Table 8.5: *And* truth table

The *And* operation could also be constructed through a list using the operator `&&` and a list of clauses. As it is a unary operator, `&&` has the precedence in the same level as the negation operator (see operator precedence on Table 8.2)

Listing 8.26: *BUP*: `&&` operator

```
1 &&( <list> )
```

The allowed operations using `&` operator are[3]:

[3]The result of using the `&&` operator to a list of clauses will be equivalent to applying `&` operator to all the elements.

Expression	Result
OR_LITERALS & *OR_LITERALS*	*AND_CLAUSES*
OR_LITERALS & *AND_LITERALS*	*AND_CLAUSES*
AND_LITERALS & *OR_LITERALS*	*AND_CLAUSES*
AND_LITERALS & *AND_LITERALS*	*AND_LITERALS*
AND_CLAUSES & *AND_CLAUSES*	*AND_CLAUSES*
AND_CLAUSES & *AND_CLAUSES*	*AND_CLAUSES*

Table 8.6: *BUP*: Allowed operations with &

8.1.11.1.4. *Or* The or operator has the following truth table:

a	b	a | b
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

Table 8.7: *Or* truth table

The *Or* operation also could be constructed through a list using the operator || and a list of clauses. As it is a unary operator, || has the precedence in the same level as the negation operator (see operator precedence on Table 8.2)

Listing 8.27: *BUP*: || operator

```
1 ||( <list> )
```

The allowed operations using | operator are[4]:

Expression	Result
LITERAL | *LITERAL*	*OR_LITERALS*
OR_LITERALS | *OR_LITERALS*	*OR_LITERALS*
OR_LITERALS | *AND_LITERALS*	AND_CLAUSES

Table 8.8: *BUP*: Allowed operations with |

8.1.11.1.5. Implication The implication operator has the following truth table:

[4]The result of using the || operator to a list of clauses will be equivalent to applying | operator to all the elements.

a	b	a -> b
0	0	**1**
0	1	**1**
1	0	**0**
1	1	**1**

Table 8.9: *Implication* truth table

The allowed operations using `->` operator are:

Expression	Result
AND_LITERALS `->` *OR_LITERALS*	*OR_LITERALS*
OR_LITERALS `<-` *AND_LITERALS*	*OR_LITERALS*
LITERAL `->` *AND_LITERALS*	*AND_CLAUSES*
AND_LITERALS `<-` *LITERAL*	*AND_CLAUSES*

Table 8.10: *BUP*: Allowed operations with `->`

8.1.11.1.6. Double implication The double implication operator has the following truth table:

a	b	a <-> b
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

Table 8.11: *Double implication* truth table

The allowed operations using `<->` operator are:

Expression	Result
LITERAL `<->` *AND_LITERALS*	*AND_CLAUSES*
AND_LITERALS `<->` *LITERAL*	*AND_CLAUSES*
LITERAL `<->` *OR_LITERALS*	*AND_CLAUSES*
OR_LITERALS `<->` *LITERAL*	*AND_CLAUSES*

Table 8.12: *BUP*: Allowed operations with `<->`

8.1.11.2. Cardinality constraints

Apart from simple boolean formulas, *BUP* accepts *Cardinality Constraints* in the model specification, that are later automatically translated to *CNF* by *GOS*. These kind of constraints state that at most (at least, or exactly) k out of a propositional literals list can be true.

Listing 8.28: *BUP*: Exactly-K constraint

```
EO( <list> )
```

Listing 8.29: *BUP*: At-Most-K constraint

```
AMO( <list> )
```

Listing 8.30: *BUP*: At-Least-K constraint

```
ALO( <list> )
```

Listing 8.31: *BUP*: Exactly-K constraint

```
EK( <list>, <intExpression> )
```

Listing 8.32: *BUP*: At-Most-K constraint

```
AMK( <list>, <intExpression> )
```

Listing 8.33: *BUP*: At-Least-K constraint

```
ALK( <list>, <intExpression> )
```

8.1.11.3. *forall* structure

BUP language support *forall* structures used to loop lists and add constraints to the model.

Listing 8.34: *BUP*: *forall* constraint

```
forall ( <ident> in <list> <, <ident> in <list> >*) {
    <constraint>*
};
```

8.1.11.4. *if* structure

BUP supports *if* structures used to conditionally add constraints to the model.

Listing 8.35: *BUP*: *if* constraint

```
<if ( <boolExpression> ) { <constraint>* } >
<else if ( <boolExpression> ) { <constraint>* } >*
<else { <constraint>* } >?
```

8.1.12. Strings

BUP allows string **only** in the output block (see Section 8.1.4). A string could be:

- An **explicit string** by adding quotes at the beginning and at the end of the text:

Listing 8.36: *BUP*: Explicit string

```
ESCAPED_QUOTE : '\\"';
TK_STRING :   '"' ( ESCAPED_QUOTE | ~('"') )*? '"';
```

- A **string concat** using *++* operator:

Listing 8.37: *BUP*: String concat

```
<string> <++ <string>>*
```

- A **string between parenthesis**:

Listing 8.38: *BUP*: String between parenthesis

```
( <string> )
```

- A **ternary** operation having a string expression to the both sides:

Listing 8.39: *BUP*: Ternary operation string

```
<boolExpression> ? <string1> : <string2>
```

- **A variable**[5] **or param access** automatically casted to string (see 8.1.9)
- An **expression**[6] (see 8.1.8) automatically casted to string.
- A **list** (see 8.1.10) automatically casted to string.

8.2. Parameters file

The data file must contain the value for all the **parameters** declared in the *viewpoint block* (see 8.1.2) in *JSON* format.

Listing 8.40: *GOS*: Example parameters *JSON* file using basic types, arrays and entities

```
{
    "nSize" : 2,
    "mSize" : 1,
    "board": [
        [
            {
```

[5] Strings are only allowed in *output block*, where variables are treated as bool basic type

[6] Variables are automatically casted to bool basic type to do operations between expressions in the *output-block*

```
                "valueA" : 1,
                "valueB" : 2,
                "b" : true
            }
        ],
        [
            {
                "valueA" : 25,
                "valueB" : 10,
                "b" : false
            }
        ]
    ]
}
```

String and *null* values from the *JSON* standard are not allowed since the basic types of *BUP* are *int* and *bool* (see 8.1.5.1).

9. Analysis and design of the system

9.1. Main flow

1. Read and store input data (see 9.1.1).
   a) Stop if there are errors.
2. Parse and generate entities definition types (see 9.1.2).
   a) Stop if there are errors.
3. Parse variables declarations (see 9.1.3) and parse parameters declarations and assign them with input data (see 9.1.4).
   a) Stop if there are errors.
4. Parse constraints and generate the boolean formula. (see 9.1.5).
   a) Stop if there are errors.
   b) If *show-formula* flag is active: Print the CNF formula in *DIMACS* format (see 9.1.6).
   c) Else apply the SAT solver to the generated formula and get the result (see 9.1.7).
      i. If the formula is satisfable:
         A. If exists an output block: Parse custom output and show result (see 9.1.8.2)
         B. Else show default output (see 9.1.8.1).
      ii. Else show *UNSAT*.

The figure 9.1 shows the main flow assigning each task to the layer that effectuates it.

9.1.1. Input data read

The first the compiler does is to read the input data file. The reason of reading the input data before the entities and parameter declaration is to allow declare variables or parameters using previously declared parameters, e.g., to indicate an array dimension using a parameter.

Thus, what *GOS* firstly does is generate an auxiliary data structure containing the input *JSON* data. The Figure 9.2 shows the implemented data structure class diagram.

Once generated the input visitor from the *JSON grammar* (see *JSON* grammar in Appendix B and input visitor generation on Section 9.3.1) the auxiliary structure is filled in order to store the given input data.

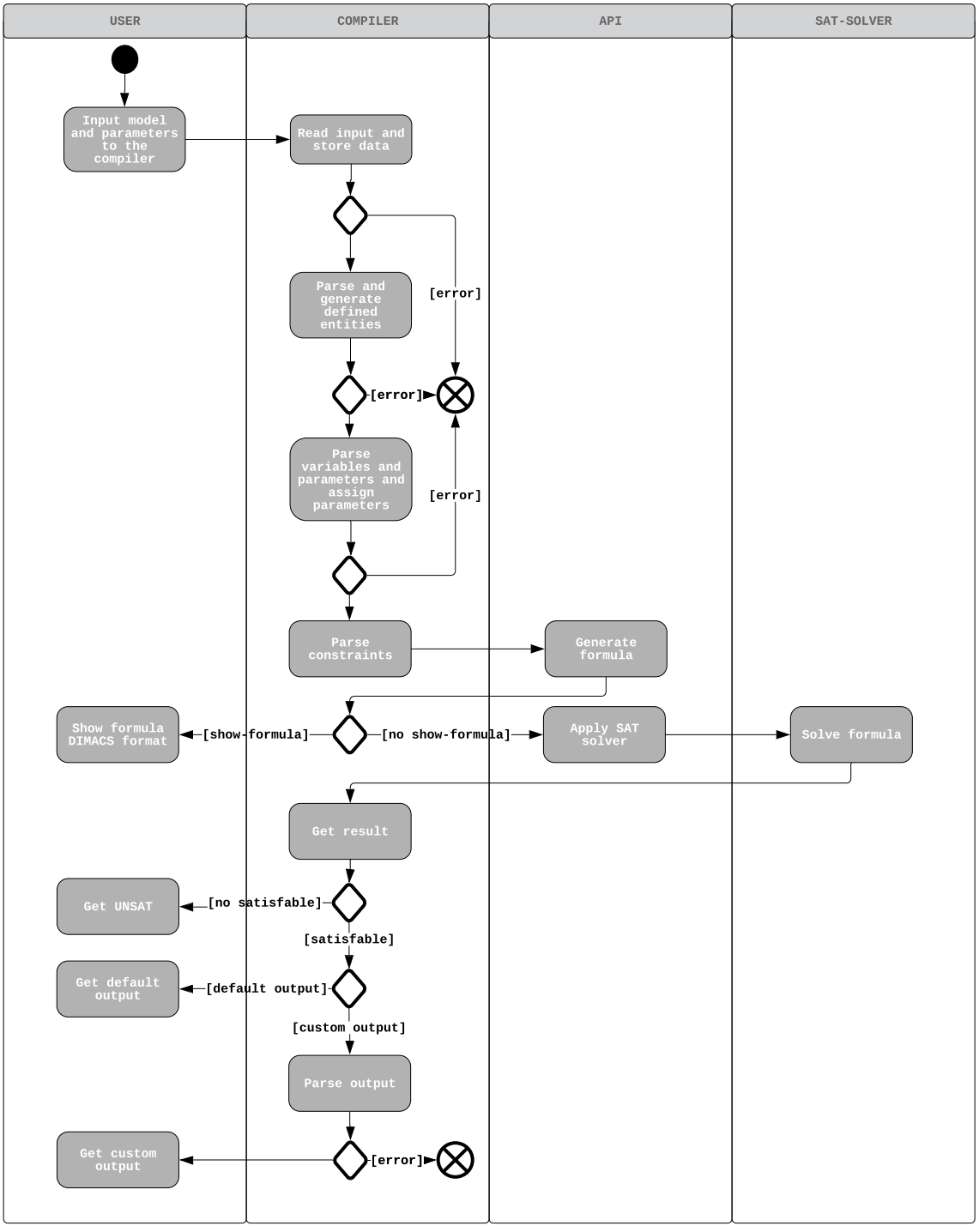


Figure 9.1: Tool layered flow scheme

Our root object will be a *ParamJSON* and each attribute will be added recursively using the *add(Param a)* method while visiting each node found in the input file.

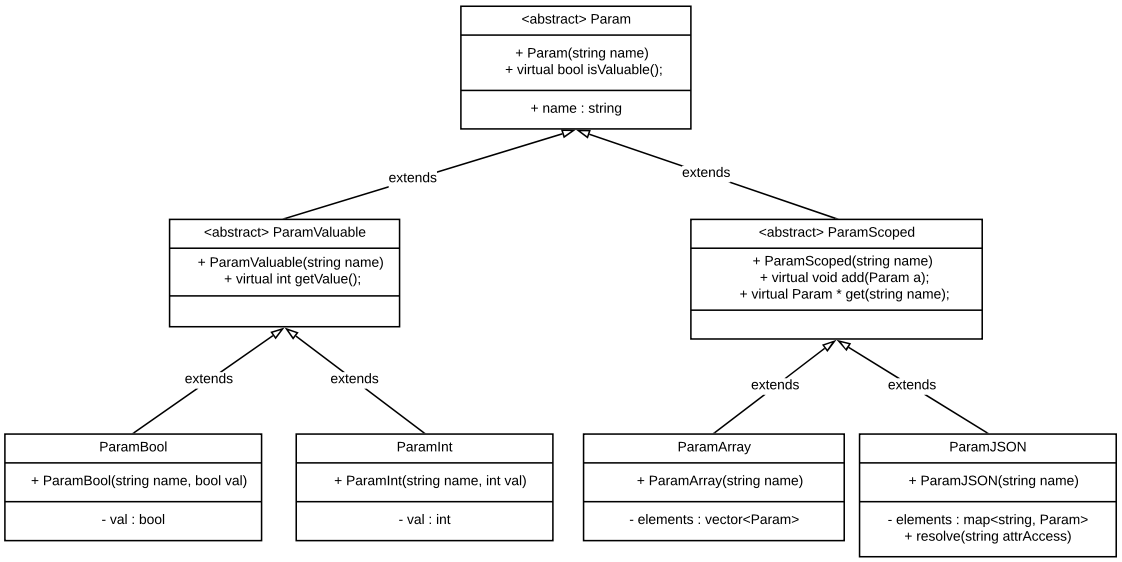


Figure 9.2: *Compiler flow:* auxiliary data structure to store input *JSON* data

9.1.2. Entity definition

The step before declaring variables and parameters is to check if it exists any new entity defined by the user and save the definition into the *Symbol Table* in order to be able to declare symbols using this new type.

9.1.3. Variable declaration

Once entering the *viewpoint-block*, *GOS* declares the model resolution variables and adds its reference to the *SMTFormula* (see 9.5).

9.1.4. Parameter declaration and assignation

At the same time as the variable declaration, *GOS* also declares the model parameters and assigns them with the values stored int the previously generated input data structure (see 9.1.1).

9.1.5. Formula generation

Once all the parameters and variables are declared and initialized, it is time to generate the formula by applying the constraints defined in the *constraints-block*.

9.1.6. Print CNF formula: *DIMACS* format

If the compiler is running using the *show-formula* flag (see 9.6.1), the compiler will not get a solution and instead it will print the *CNF* formula in *DIMACS* format.

This format is widely accepted as the standard format for boolean formulas in CNF. A *DIMACS* file starts with comments (each line starts with `c`). The number of variables and the number of clauses is defined by the line `p cnf` *`variables clauses`*

Each of the next lines specifies a clause: a positive literal is denoted by the corresponding number, and a negative literal is denoted by the corresponding negative number. The last number in a line should be zero. For example,

```
c A sample .cnf file.
p cnf 3 2
1 -3 0
2 3 -1 0
```

9.1.7. Solver application

Once the formula has been obtained by applying all the constraints, the solver (in our case, MiniSAT) is run to obtain a model that satisfies all the constraints. If the solver obtain a solution we have proved by giving a model that the formula is *SATisfable*. On the other hand, if the solver does not find a solution, we can assure that the formula is unsatisfable.

9.1.8. Output

9.1.8.1. Default output

The *output-block* is optional. In the case that the user has not set a custom output and the formula is *SATisfable*, the output will be the default. The default output consists on showing the value of all the declared variables in the viewpoint-block and their values given by the solver.

9.1.8.2. Custom output

In the case that the user has defined a custom output and the formula is *SATisfable*, the shown output would be the defined in the *output-block*.

9.2. Symbol Table

The *Symbol Table* is the data structure in charge of managing all symbols in the code. To implement the *Symbol Table* it was taken as a starting point the *Symbol Table for Data Aggregate pattern* of the book *Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages* (Parr, 2010, chapter 7).

Although it was used as a basis, the final result bears little resemblance to the pattern proposed in the book.

The implementation of the *SymbolTable* in this project proposes a new framework in which symbols are mixed with its values to simplify the recursive hierarchy and name resolution methods. Having a single hierarchy of symbols and values makes it possible to explore the parsing tree in a elegant way in accordance with the *SymbolTable*.

The Figure 9.3 show the complete class diagram of the implemented Symbol Table.

The *SymbolTable* constructor define a global *Scope* and initalizes all pre-defined types as *static* constants to ensure a unique instance and offers the possibility to be accessed without having to explore the *SymbolTable* structure.

Listing 9.1: *SymbolTable*: Constructor

```
class SymbolTable {
    static const int tCustom = 0;
    static const int tArray = 1;
    static const int tInt = 2;
    static const int tBool = 3;
    static const int tVarBool = 4;
    static const int tString = 5;

    static BuiltInTypeSymbol *_integer;
    static BuiltInTypeSymbol *_boolean;
    static BuiltInTypeSymbol *_varbool;
    static BuiltInTypeSymbol *_string;

    GlobalScope * gloabls;

    SymbolTable(){
        gloabls = new GlobalScope();
        this->gloabls->define(_integer);
        this->gloabls->define(_boolean);
        this->gloabls->define(_varbool);
        this->gloabls->define(_string);
    }
}

BuiltInTypeSymbol * SymbolTable::_integer = new BuiltInTypeSymbol("int", ↩
    ↪ SymbolTable::tInt);
BuiltInTypeSymbol * SymbolTable::_boolean = new BuiltInTypeSymbol("bool", ↩
    ↪ SymbolTable::tBool);
BuiltInTypeSymbol * SymbolTable::_varbool = new BuiltInTypeSymbol("varbool", ↩
    ↪ SymbolTable::tVarBool);
BuiltInTypeSymbol * SymbolTable::_string = new BuiltInTypeSymbol("string", ↩
    ↪ SymbolTable::tString);
```

9.2.1. Scope

A *Scope* refers to the visibility of variables and params in one part of a program to another part of that program. In our project, a *Scope* is a container of Symbols and is implemented

as an interface, so all the classes that implements all methods of *Scope* could be considered a *Scope*:

string getScopeName() Returns the name of the *Scope*.

string getFullName() Returns the name of the *Scope* concatenated to rest of enclosing scopes names hierarchy.

Scope * getEnclosingScope() Returns the parent *Scope*, if exists.

void define(Symbol * sym) Add *sym* to the *Scope*.

Symbol * resolve(string name) Resolves the symbol named *name* in the current scope.

map<string, Symbol*> getScopeSymbols() Returns all symbols in the *scope*.

9.2.1.1. *GlobalScope*

A *GlobalScope* is a kind of scope that its variables must be accessible from anywhere in the code, e.g., type or entities definitions,... Our *SymbolTable* have only one *GlobalScope* where basic types, defined entities and first level parameters and variable declarations are stored.

The *GlobalScope* does not have an enclosing scope to enforce the resolution method explore only the names defined in the current scope.

9.2.1.2. *LocalScope*

A *LocalScope* is a kind of scope that its variables are only available in a current block of code. After exiting this block of code, the variables declared inside will be no longer accessible.

In our case, we can not define new variables or parameters outside viewpoint block and *LocalScope*s are limited to the temporary identifiers created on, e.g., *forall* structures (see 8.1.11.3).

The *LocalScope* has always an enclosing scope, so the name resolution method explores the name in the current scope and, if not exists, then call the resolution method of the enclosing scope. This allow not only resolve the current scope defined names but also the previously defined parameters and variables in the *GlobalScope* or other nested enclosing *LocalScope*s.

9.2.2. *Symbol*

In this project, the *Symbol* is the base of almost everything. A *Symbol* could be a *Type*, a *ScopedSymbol* or a *ValueSymbol*. A *Symbol* is stored (and could be resolved) in a *Scope*. All *Symbol*s have a name and a *Type*.

9.2.2.1. *Type*

A *Type* is a kind of *Symbol* that defines the characteristics of a parameter or variable. It exists two kind of *Type*s:

BuiltInTypeSymbol Defines the pre-defined types of the language (*int, bool, variable* and *string*). They are initalized when constructing the *SymbolTable* (see 9.2).

ScopedSymbol A *ScopedSymbol* could be an *ArraySymbol* or a *StructSymbol* and they also could be considered *Type*s (see 9.2.2.3).

9.2.2.2. *ValueSymbol*

A *ValueSymbol* is considered a **leaf** in our hierarchy, i.e., it must contain the current value of the parameter or variable. For instance, having declared *param int a[3][3]*, *param int b*, *EntityA c*[1] and *var d*.

- ***a[0][0]*** is a *ValueSymbol* (and an *AssignableSymbol*).
- ***a[0]*** is **NOT** a *ValueSymbol*: is an *ArraySymbol*.
- ***b*** is a *ValueSymbol* (and an *AssignableSymbol*).
- ***c.a*** is a *ValueSymbol* (and an *AssignableSymbol*).
- ***c*** is **NOT** a *ValueSymbol*: is a *StructSymbol*.
- ***d*** is a *ValueSymbol* (and a *VariableSymbol*).

A *ValueSymbol* is a *VariableSymbol* or an *AssignableSymbol*.

9.2.2.2.1. *VariableSymbol* A *VariableSymbol* contains the reference to the a concrete model resolution variable declared as a `var` in the *viewpoint-block*. You can resolve the reference to the variable in order to apply constraints but the model value is not accessible until the *output-block*.

9.2.2.2.2. *AssignableSymbol* An *AssignableSymbol* contains the value of the `param`s declared in the *viewpoint-block* and assigned through input data file. You can resolve their name and access to their value from anywhere in the code.

9.2.2.3. *ScopedSymbol*

A *ScopedSymbol* is the most complex *Symbol*. It could be considered a *Type*, e.g, defining a custom entity, and it could be considered a Scope since it implements the *Scope* interface. In this case, the method *Symbol* resolve(string name)* of the *Scope* interface is used to resolve entity attributes or the array indexes.

A *ScopedSymbol* can be a *Type* when defining an *Entity* and this type could be used in the *viewpoint-block* to declare concrete instances.

9.2.2.3.1. *ArraySymbol* Used to create array declarations of elements of any *Type*.

9.2.2.3.2. *StructSymbol* Used to create custom entity definitions (*entity-block*) and entity declarations (*viewpoint-block*).

[1] *EntityA* is a user-defined entity having only one *param int a* attribute.

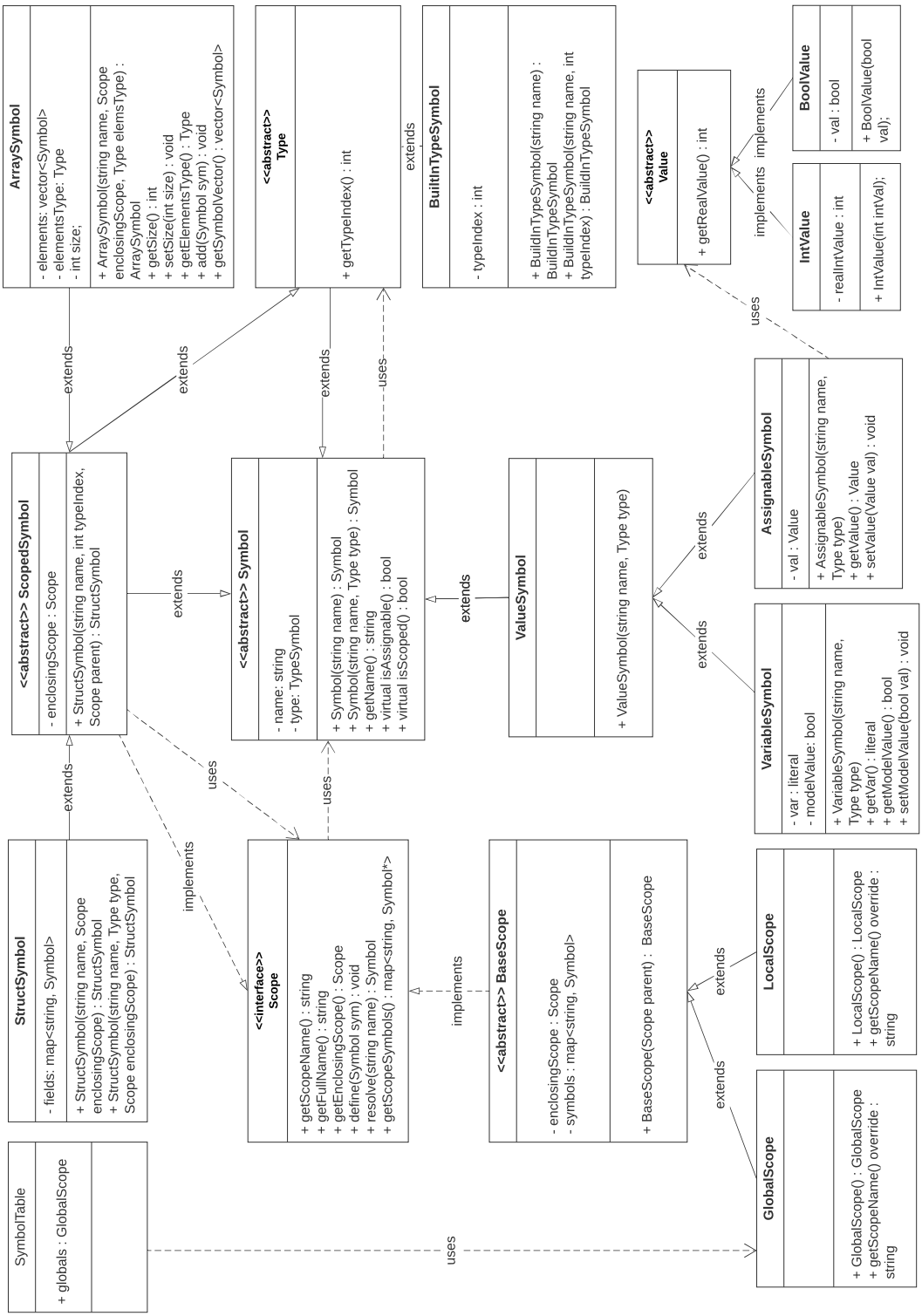


Figure 9.3: *Symbol Table:* Class diagram

9.3. Visitors

A visitor is the structure that allows to explore the generated parsing tree given an *ANTLR* grammar and an input text (see Section 7.1.2 Syntactic Analysis).

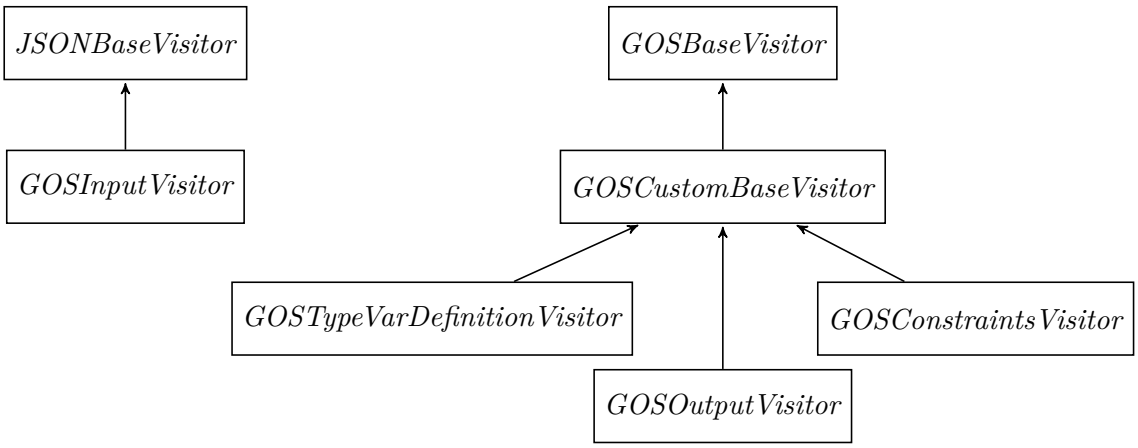


Figure 9.4: Visitors schema

It is defined a Visitor hierarchy to give the wanted behaviour in each block of the code by overriding the *visit methods*.

9.3.1. *GOSInputVisitor*

GOSInputVisitor explores the input *JSON* parsing-tree nodes and generates a auxiliar structure to store the parameter assignation of the input. (see *Input data read* on Section 9.1.1).

Listing 9.2: *GOSJSONInputVisitor*

```
class GOSJSONInputVisitor : public JSONBaseVisitor {
private:
    ParamJSON *base;
    ParamScoped *current;

public:
    GOSJSONInputVisitor() {
        base = new ParamJSON("base");
        current = base;
    }
    antlrcpp::Any visitValue(JSONParser::ValueContext *ctx) override {...}
    antlrcpp::Any visitPair(JSONParser::PairContext *ctx) override {...}
    antlrcpp::Any visitArr(JSONParser::ArrContext *ctx) override {...}
    antlrcpp::Any visitJson(JSONParser::JsonContext *ctx) override {...}
};
```

9.3.2. *GOSBaseVisitor*

The *GOSBaseVisitor* is the default *ANTLR* parsing-tree visitor of the grammar and the model input.

9.3.2.1. *GOSCustomBaseVisitor*

GOSCustomBaseVisitor overrides the methods of *GOSBaseVisitor* that have common behaviour everywhere, e.g., expressions, identifier access, lists,..

Listing 9.3: GOSBaseVisitor

```
class GOSCustomBaseVisitor : public GOSBaseVisitor {

protected:
    bool accessingNotLeafVariable = false;
    SymbolTable *st;
    SMTFormula *_f;
    Scope *currentScope;
    Scope *currentLocalScope = nullptr;

public:

    explicit GOSCustomBaseVisitor(SymbolTable *symbolTable, SMTFormula *f)
    {
        this->st = symbolTable;
        this->_f = f;
        this->currentScope = this->st->gloabls;
    }

    antlrcpp::Any visitExprTop(GOSParser::ExprTopContext *ctx) {...}
    ( ... )
    antlrcpp::Any visitExprSumDiff(GOSParser::ExprSumDiffContext *ctx){...}
    ( ... )
    antlrcpp::Any visitVarAccess(GOSParser::VarAccessContext *ctx) {...}
    ( ... )
    antlrcpp::Any visitRangList(GOSParser::RangListContext *ctx) {...}
    ( ... )
};
```

The *GOSCustomBaseVisitor* attributes are protected to allow subclasses access to the *SymbolTable* and the *SMTFormula*.

9.3.2.1.1. *GOSTypeVarDefinitionVisitor* Adds the custom entities definitions and the variable and parameter declarations to the *SymbolTable*. Receives the auxiliar input structure generated by the *GOSInputVisitor* in order to assign the values to the parameter declarations.

It overrides the dataAccess method from the *GOSCustomBaseVisitor* to get the data from the auxiliar input structure (see 9.1.1).

Listing 9.4: *GOSTypeVarDefinitionVisitor*

```
class GOSTypeVarDefinitionVisitor : public GOSCustomBaseVisitor {
private:
    ParamJSON *params;

public:
    explicit GOSTypeVarDefinitionVisitor(SymbolTable *symbolTable, SMTFormula *f, ParamJSON *params)
            : GOSCustomBaseVisitor(symbolTable, f) {
        this->params = params;
    }

    antlrcpp::Any visitEntityDefinitionBlock(GOSParser::EntityDefinitionBlockContext *ctx) override {...}

    antlrcpp::Any visitViewpointBlock(GOSParser::ViewpointBlockContext *ctx) override {...}

    antlrcpp::Any visitVarDefinition(GOSParser::VarDefinitionContext *ctx) override {...}

    antlrcpp::Any visitParamDefinition(GOSParser::ParamDefinitionContext *ctx) override {...}

    antlrcpp::Any visitVarAccess(GOSParser::VarAccessContext *ctx) override {...}

    (...)
}
```

9.3.2.1.2. *GOSConstraintsVisitor* Visits the *constraints-block* nodes and adds the variables constraints to the *SMTFormula*[2]

Listing 9.5: *GOSConstraintsVisitor*

```
class GOSConstraintsVisitor : public GOSCustomBaseVisitor {
public:
    explicit GOSConstraintsVisitor(SymbolTable *symbolTable, SMTFormula *f) : GOSCustomBaseVisitor(symbolTable, f) {}

    antlrcpp::Any visitConstraint(GOSParser::ConstraintContext *ctx) override {...}

    antlrcpp::Any visitIfThenElse(GOSParser::IfThenElseContext *ctx) override {...}

    antlrcpp::Any visitForall(GOSParser::ForallContext *ctx) override {...}

```

[2]Although the used api is *SMT*-oriented, it is fully compatible to encode SAT. SMT is an extension of SAT that allows predicates more expressive than propositional formulas.

```
    (...)
}
```

9.3.2.1.3. *GOSOutputVisitor* If the user has defined a custom output, it generates the output stream. This visitor also overwrites the *dataAccess* method to allow access model variables **values** (if the model is satisfable) and treat it as a base type bool symbol.

Listing 9.6: *GOSOutputVisitor*

```
class GOSOutputVisitor : public GOSCustomBaseVisitor {

public:
    GOSOutputVisitor(SymbolTable *symbolTable, SMTFormula *f) : ↩
    ↪ GOSCustomBaseVisitor(symbolTable, f) {}

    antlrcpp::Any visitOutputBlock(GOSParser::OutputBlockContext *ctx) override↩
    ↪  {...}

    antlrcpp::Any visitString(GOSParser::StringContext *ctx) override {...}

    (...)
}
```

9.4. Error handling

The different ways to handle errors in a compiler are explained in the Section 5.3.1. *GOS* uses different strategies depending on the phase of the compiler.

9.4.1. Lexical Errors

The errors that can be detected in the lexical analysis are few (strange characters, misspelled constants,...). The strategy used in the *lexical phase* is ***panic*** (stop the execution when an error is found).

9.4.2. Syntactic Errors

In the syntactic phase, *GOS* use the default *ANTLR* strategy. When the parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon, etc.
If *ANTLR* can recover the input, the errors are shown but the execution does not stop. On the other hand, if *ANTLR* can not recover the input, the error is shown and the execution stops.

9.4.3. Semantic Errors

This kind of errors are related to the given behaviour of the compiler.*GOS* have fully customized semantic errors. A new C++ exception type, *GOSException*, has been implemented,

which is a class that unifies the output of the compiler errors.

Listing 9.7: *GOSException*

```
class GOSException : public exception {
private:
    int line;
    int column;
    string message;

public:
    GOSException(int line, int pos, const string &message) :
        line(line), column(pos), message(message) {
        SymbolTable::errors = true;
    }

    string getErrorMessage(){
        string error =  string("ERROR on line ") + to_string(line) + ":" + ↩
    ↪ to_string(column) + "\n\t" + message;
        return error;
    }
};
```

GOSException extends the default C++ exception class and defines a default method *getErrorMessage* that returns a *string* with the unified error outptut style:

Listing 9.8: Default error message

```
ERROR on line <line>:<col>:
    <message>
```

Exceptions are throwed and catched using *try-catch* blocks set in top-level nodes of the parsing-tree.

There exists a *GOSException* repository where there are defined all the defined exceptions extending *GOSException*:

Listing 9.9: GOSBadAccessException

```
class GOSBadAccessException : public GOSException {
public:
    GOSBadAccessException(int line, int pos, const string &badAccess) :
        GOSException(
                line,
                pos,
                "Invalid access: \"" + badAccess + "\" is not a variable or ↩
    ↪ param"
        ) {}
};
```

Listing 9.10: GOSInvalidFormulaException

```
class GOSInvalidFormulaException : public GOSException {
public:
    GOSInvalidFormulaException(int line, int pos, string formula, string ↩
    ↪ message = "") :
        GOSException(
                line,
                pos,
                "Invalid formula \"" + formula + "\": " + message
            ) {}
};

};
```

9.5. *SMT* api

The solving part of *GOS* uses a *SMT API*[3] implemented by one of my supervisors and member of L∧P research group in the University of Girona, Jordi Coll, that gives methods to define SMT (in our case SAT) formulas, create custom encodings and solve with a set of *SMT solvers* (in our case *MiniSAT*).

GOS generates the Propositional Formula by visiting the nodes of the parsing-tree and calling methods of *SMTFormula* to define variables and apply constraints. Once the formula is fully generated, it is sent to the solver to get a model.

9.5.1. *SMTFormula*

SMTFormula is used to define variables and apply constraints. The API uses custom defined classes *boolvar* (represents a variable), *literal* and *clause*.

Listing 9.11: *SMTFormula api used methods*

```
    //Get the trivially false variable
    boolvar falseVar();

    //Get the trivially true variable
    boolvar trueVar();

    //Get a new unnamed Boolean variable
    boolvar newBoolVar();

    //Add the empty clause to the formula
    void addEmptyClause();

    //Add clause 'c' to the formula
    void addClause(const clause &c);

    //All all the clauses in 'v' to the formula
```

[3] Although the used api is *SMT*-oriented, it is fully compatible to encode SAT. SMT is an extension of SAT that allows predicates more expressive than propositional formulas.

```
17      void addClauses(const std::vector<clause> &c);
18
19      //Adds at-least-K constraint on the literals in 'v'
20      void addALK(const std::vector<literal> & v, int K);
21
22      //Adds at-most-K constraint on the literals in 'v'
23      void addAMK(const std::vector<literal> & v, int K, CardinalityEncoding enc
        ↪ = CARD_SORTER);
24
25      //Adds exactly-K constraint on the literals in 'v'
26      void addEK(const std::vector<literal> & v, int K);
```

9.5.2. *GOSEncoding*

To interact with the solver, the api uses an abstract class *Encoding* with some pure-virtual methods to implement by the subclasses:

- *SMTFormula * encode(int LB = INT_MIN, int UB = INT_MAX)*
- *bool printSolution(std::ostream & os) const*

In our case, the encoded formula is generated externally (using visitors) and the *encode()* method only returns the generated *SMTFormula*.

Listing 9.12: *GOSEncoding*

```
1  class GOSEncoding : public Encoding {
2
3  private:
4      SMTFormula *f;
5      SymbolTable *st;
6
7      void fillModelValuesResult(Scope *currentScope, const EncodedFormula
       ↪ formula, const vector<bool> & bmodel) {...}
8
9      void printModelSolution(Scope *currentScope, ostream &os, string prefix = "
       ↪ ") const {...}
10
11 public:
12
13     GOSEncoding(SMTFormula *formula, SymbolTable *st) {
14         this->f = formula;
15         this->st = st;
16     }
17
18     SMTFormula *encode(int LB = 0, int UB = 0) override {
19         return f;
20     }
21
```

```
    bool printModelSolution(ostream &os) const {
        printModelSolution(this->st->gloabls, os);
        return true;
    }

    bool printSolution(ostream &os) const override {
        //GOS uses its custom method to print solution.
        return true;
    }

    void setModel(const EncodedFormula &ef, int lb, int ub, const vector<bool> &bmodel, const vector<int> &imodel) override {
        fillModelValuesResult(this->st->gloabls, ef, bmodel);
    }
};
```

9.5.3. Controller

Once the formula and the encoding are created, *GOS* uses the the *BasicController* to run the solver.

BasicController basically uses the solving arguments (see 9.6.2) and the encoding to run. The other parameters are *SMT*-related and dora not matter in our case.

Listing 9.13: *BasicController*

```
class BasicController {
 protected:
    int LB;
    int UB;
    Encoding * encoding;
    bool minimize;
    SolvingArguments * sargs;

public:
    BasicController(SolvingArguments * sargs, Encoding * enc, bool minimize, int lb, int ub);
    virtual ~BasicController();

    static void afterSatisfiabilityCall(int lb, int ub, Encoder * encoder);
    static void afterNativeOptimizationCall(int lb, int ub, Encoder * encoder);
    static void onNewBoundsProved(int lb, int ub);
    static void onSATSolutionFound(int & lb, int & ub, int & obj_val, Encoding * encoding);
    static void onProvedOptimum(int opt);
    static void onProvedSAT();
    static void onProvedUNSAT();

    virtual void run();
};
```

The use of *BasicController* is as simple as call the *run()* and it will run the solver according to the *SolvingArguments*.

9.6. *GOS* compiler arguments

Apart from the *SMT* api, there is implemented an argument manager to configure the solver. The main reason of that is give a standard input to the compiler.
In our case, we will force to use *MiniSAT* as a solver and set the solver to return the first solution found (optimization formula use is not allowed).

GOS will require only the input model file and the param file to run.

Listing 9.14: *GOS main: argument management*

```
int main(int argc, char **argv) {
    Arguments<ProgramArg> *pargs = new Arguments<ProgramArg>(
            {
                    arguments::arg("modelfile", "Instance file path."),
                    arguments::arg("datafile", "Input params file path."),
            },
            2,
            {
                    arguments::bop("pf", "print-formula", SHOWFORMULA, false,
                    "Print CNF formula"),
            },
            "Solve CSP to SAT"
    );
    SolvingArguments*sargs =SolvingArguments::readArguments(argc, argv, pargs);
    bool showFormula = pargs->getBoolOption(SHOWFORMULA);

    //Force use MiniSAT
    SolvingArg solver = sargs->getOptionRef("-s");
    sargs->setOption(solver, (string) "minisat");

    //Force no optimization
    SolvingArg optimize = sargs->getOptionRef("-o");
    sargs->setOption(optimize, (string) "check");

    //Configure formula print if flag pf active.
    if (showFormula) {
        SolvingArg print = sargs->getOptionRef("-e");
        sargs->setOption(print, true);
        SolvingArg format = sargs->getOptionRef("-f");
        sargs->setOption(format, (string) "dimacs");
    }

    string inputStr = readFile(pargs->getArgument(1));
    string modelStr = readFile(pargs->getArgument(0));

    GOSCompiler *compiler = new GOSCompiler(inputStr, modelStr, sargs);
```

```
37      compiler->run();
38
39      return 0;
40  }
```

Arguments checks the correctness of input parameters and gives commands to customize the *SolvingArguments*.

9.6.1. *print-formula* flag

The compiler allows to be executed using the `-pf=1` or `--print-formula=1` option to print the generated CNF formula in *DIMACS* format instead of applying the solver (see 9.1.6).

9.6.2. *SolvingArguments*

Solving arguments are those arguments that are allowed as a compiler options that could change the solver behaviour. For further information type "-h" as a compiler parameter and it will print all the options[4].

[4]It is not recommended to change the default solving arguments since *GOS* compiler have only a subset of the *SMT* api and not all the options are available. Apart from this, The only solver implemented for *SAT* is *MiniSAT*.

10. Deploying

In this section it is explained the two ways that the compiler has been deployed.

10.1. Downloadable compiler

The last version of the compiler is published on my GitHub repository and it is simple to clone, install and use in any OS (see Appendix A Install and Run Instructions). Figure 10.1 shows how works the online version and the flow between the components it has.

10.2. Online compiler

GOS also has an online version. This was not initially defined as a requirement of the project but it turns out to seem interesting to reach a greater amount of users and was included in the lists of tasks and objectives of the project.

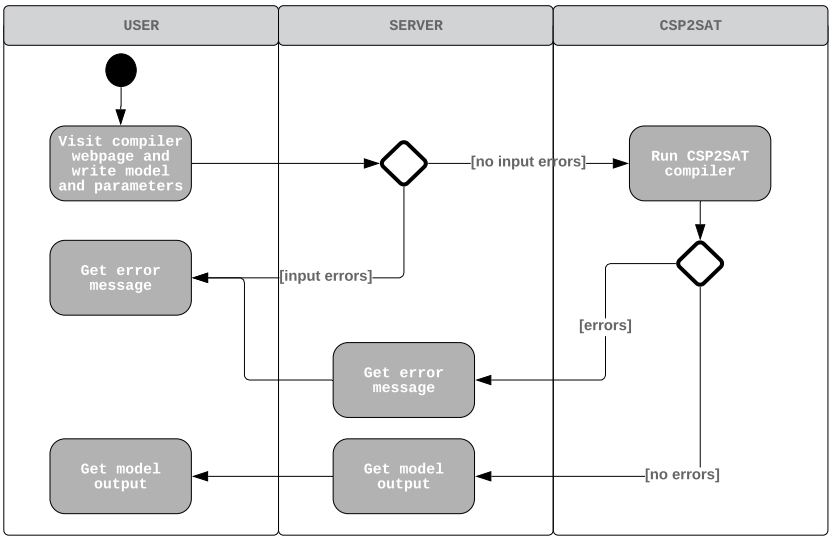


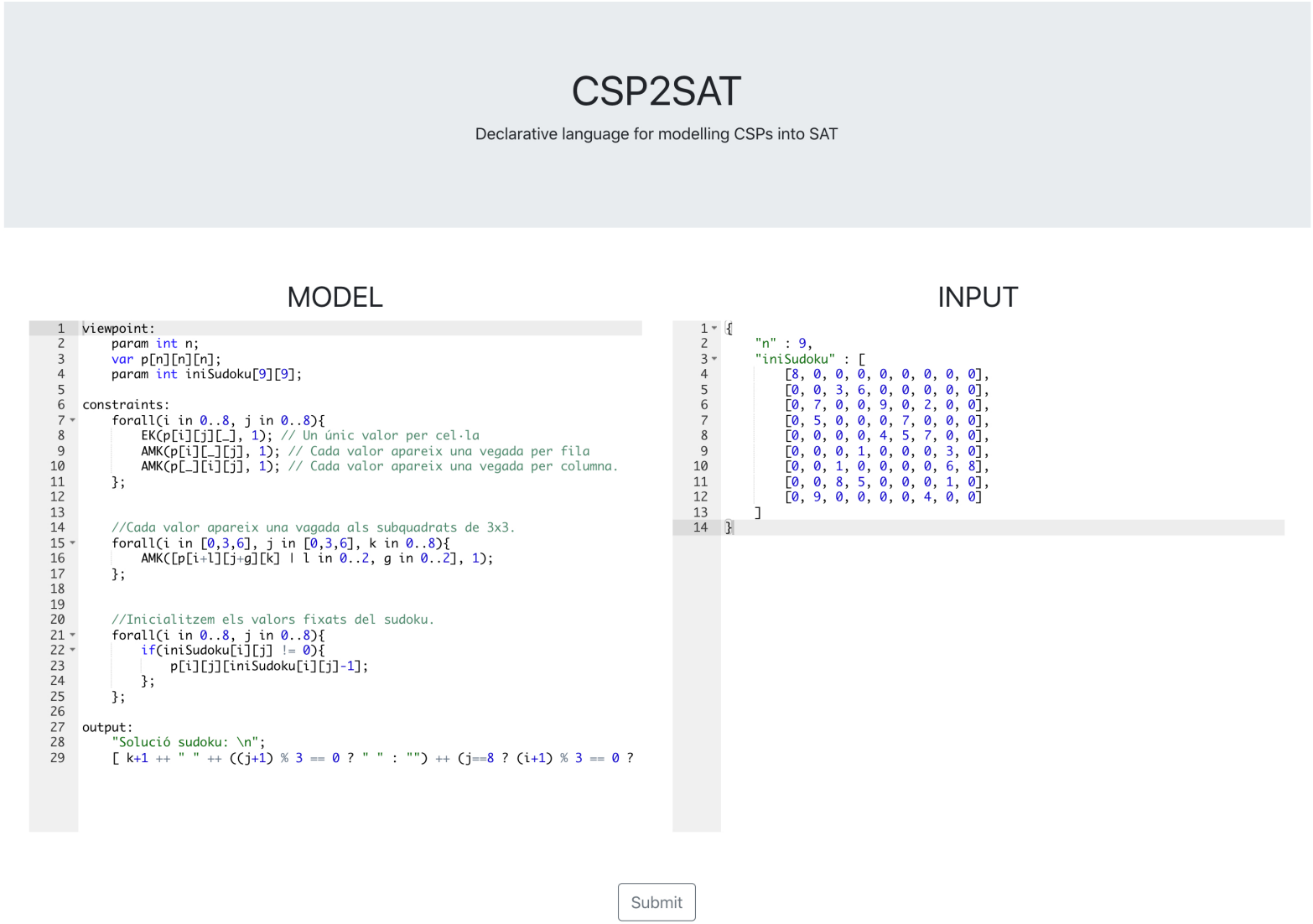
Figure 10.1: *GOS* online version flow

To implement the online editor it was used:

Frontend: *ReactJS* *Frontend* is the visual part: contains two code editors to be able to write the input model and the params and a button to get the solution. It is only responsible of sending the model to the server and receiving and displaying the response.

Backend (server): *Spring Boot* The *Spring Boot* server listens to requests from the frontend and returns the solution by executing *GOS* compiler.

At the time of writing this project it is published in a free *Google Cloud* instance using an *apache2* server. It can be accessed here (Figure 10.2 shows a screenshot of the webpage). In the future will be considered allocating this online version in the webpage of the L∧P research group in order to make easy the use, specially, for the students of *Declarative Programming* subject in the *University of Girona*.




```
viewpoint:
    param int n;
    var p[n][n][n];
    param int iniSudoku[9][9];

constraints:
    forall(i in 0..8, j in 0..8){
        EK(p[i][j][_], 1); // Un únic valor per cel·la
        AMK(p[i][_][j], 1); // Cada valor apareix una vegada per fila
        AMK(p[_][i][j], 1); // Cada valor apareix una vegada per columna.
    };


    //Cada valor apareix una vagada als subquadrats de 3x3.
    forall(i in [0,3,6], j in [0,3,6], k in 0..8){
        AMK([p[i+l][j+g][k] | l in 0..2, g in 0..2], 1);
    };


    //Inicialitzem els valors fixats del sudoku.
    forall(i in 0..8, j in 0..8){
        if(iniSudoku[i][j] != 0){
            p[i][j][iniSudoku[i][j]-1];
        };
    };

output:
    "Solució sudoku: \n";
    [ k+1 ++ " " ++ ((j+1) % 3 == 0 ? " " : "") ++ (j==8 ? (i+1) % 3 == 0 ?
```

```
{
    "n" : 9,
    "iniSudoku" : [
        [8, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 3, 6, 0, 0, 0, 0, 0],
        [0, 7, 0, 0, 9, 0, 2, 0, 0],
        [0, 5, 0, 0, 0, 7, 0, 0, 0],
        [0, 0, 0, 0, 4, 5, 7, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 3, 0],
        [0, 0, 1, 0, 0, 0, 0, 6, 8],
        [0, 0, 8, 5, 0, 0, 0, 1, 0],
        [0, 9, 0, 0, 0, 0, 4, 0, 0]
    ]
}
```

Figure 10.2: *GOS* online version

11. Results

11.1. *BUP* programming language

BUP is a new declarative programming language for modelling *CSP* and solve them using SAT.

As far as we know, there are no precedents of declarative modelling languages allowing to define tuples. *BUP* permits create complex data structures (called *entities*) that allow you to group params and variables in a common framework. In next iterations of the language (see Future Work on Section 13) it is possible to consider the idea of extending the use of tuples to object orientation, allowing for instance to include constraints over objects in the definition of the entities

To sum up, *BUP* is a language that has emerged to improve expressiveness when encoding any CSP to SAT by allowing:

- Define int or bool parameters
- Use *forall* structures to loop over parameters.
- Use *if* structures for conditionally apply constraints.
- Generate clause lists by using comprehension lists.
- Translate automatically any allowed formula to CNF.
- Customize the output when the model is satisfable.
- Produce a clear CNF encoding resulting of the individual conjunction of CNFs resulting from translating each particular constraint.
- Be easily extendible to support further constraints implemented in the SMT API.

11.2. *GOS* compiler

The *GOS* compiler allows to use the defined language, *BUP*, to solve *CSP*s. Given a *BUP* model and a *JSON* file with the data of a particular instance at hand, *GOS* compiler makes the translation to *SAT* and gives the option to print the resulting formula in a standard format, *DIMACS*, or to obtain a solution by using *MiniSAT SAT*-solver.

There are two ways of using *GOS* compiler:

- By building the project using CMake

- By using the published online version

11.3. Model examples

11.3.1. Sudoku

Sudoku is a popular Japanese puzzle that is based on the logical placement of numbers. The goal of Sudoku is to fill in a 9×9 grid with digits so that each column, row, and 3×3 section contain the numbers between 1 to 9. At the beginning of the game, the 9×9 grid will have some of the squares filled in. Your job is to use logic to fill in the missing digits and complete the grid.
The mathematical model of the sudoku is the following:

Sets G = *Set of already placed numbers*

Variables $y_{ijk} = \begin{cases} 1, & \text{if element } (i,j) \text{ of the } n\text{x}n \text{ sudoku matrix contains the integer } k \\ 0, & \text{else} \end{cases}$

Constraints

Only one k in each colum

$$\sum_{i=1}^{n} y_{ijk} = 1 \qquad \forall_{i,j} \in 1..n$$

Only one k in each row

$$\sum_{j=1}^{n} y_{ijk} = 1 \qquad \forall_{i,j} \in 1..n$$

Only one k in each sub-matrix

$$\sum_{j=mq-m+1}^{mq} y_{ijk} \sum_{i=mq-m+1}^{mq} y_{ijk} = 1 \qquad \forall_k \in 1..n\,, \forall_{p,q} \in 1..m$$

Every position in matrix must be filled once

$$\sum_{k=1}^{n} y_{ijk} = 1 \qquad \forall_{i,j} \in 1..n$$

Given elements G in matrix are set “on”

$$y_{ijk} = 1 \qquad \forall_{i,j,k} \in G$$

A 9x9 sudoku can be modelled using *BUP* as:

Listing 11.1: Sudoku example: model

```
viewpoint:
    var p[9][9][9];
    param int iniSudoku[9][9];
```

```
4
5  constraints:
6      forall(i in 0..8, j in 0..8){
7          EO(p[i][j][_]); // One value per cell
8          AMO(p[i][_][j]); // Each value one time per row
9          AMO(p[_][i][j]); // Each value one time per column
10     };
11     //Each value one time per block
12     forall(i in [0,3,6], j in [0,3,6], k in 0..8){
13         AMK([p[i+l][j+g][k] | l in 0..2, g in 0..2], 1);
14     };
15     //Initialize input fixed sudoku values.
16     forall(i in 0..8, j in 0..8){
17         if(iniSudoku[i][j] != 0){
18             p[i][j][iniSudoku[i][j]-1];
19         };
20     };
21
22 output:
23     "Sudoku solution: \n";
24     [ k+1 ++ " " ++ ((j+1) % 3 == 0 ? " " : "") ++ (j==8 ? (i+1) % 3 == 0 ? "\n↩
       ↪ \n": "\n" : "") | i in 0..8, j in 0..8, k in 0..8 where p[i][j][k]];
```

And the following input parameters:

Listing 11.2: Sudoku example: parameters input

```
1 {
2     "n" : 9,
3     "iniSudoku" : [
4         [8, 0, 0, 0, 0, 0, 0, 0, 0],
5         [0, 0, 3, 6, 0, 0, 0, 0, 0],
6         [0, 7, 0, 0, 9, 0, 2, 0, 0],
7         [0, 5, 0, 0, 0, 7, 0, 0, 0],
8         [0, 0, 0, 0, 4, 5, 7, 0, 0],
9         [0, 0, 0, 1, 0, 0, 0, 3, 0],
10        [0, 0, 1, 0, 0, 0, 0, 6, 8],
11        [0, 0, 8, 5, 0, 0, 0, 1, 0],
12        [0, 9, 0, 0, 0, 0, 4, 0, 0]
13    ]
14 }
```

The solution obtained is:

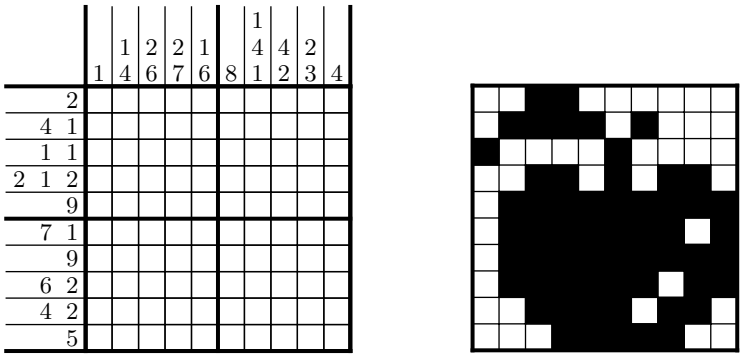
Listing 11.3: Sudoku example: solution

```
1 c restarts 3
2 c decisions 490
3 c propagations 108196
4 c conflics 291
5 c stats 0;0;0.043765;-1;4536;11361;3;-1;-1;490;108196;291;-1;-1;
6 v
7 s SATISFIABLE
8 Solució sudoku:
```

```
9
10 8 1 2  7 5 3  6 4 9
11 9 4 3  6 8 2  1 7 5
12 6 7 5  4 9 1  2 8 3
13
14 1 5 4  2 3 7  8 9 6
15 3 6 9  8 4 5  7 2 1
16 2 8 7  1 6 9  5 3 4
17
18 5 2 1  9 7 4  3 6 8
19 4 3 8  5 2 6  9 1 7
20 7 9 6  3 1 8  4 5 2
```

11.3.2. Nonogram

Nonograms, also known as Picross or Griddlers, are image logic puzzles in which cells in a grid must be colored or blank according to the numbers on the side of the grid to reveal a hidden image. In this type of puzzle, numbers are a form of discrete tomography that measures the number of continuous lines of filled-in squares there are in any given row or column. For example, a track of "4 8 3" would mean that there are sets of four, eight, and three filled squares, in that order, with at least one blank square between successive groups.



We can encode *nonograms* with *BUP* using the following parameters and variables:

```
viewpoint:
    param int rowSize;
    param int colSize;
    param int maxNonos;

    param int rowNonos[rowSize][maxNonos];
    param int colNonos[colSize][maxNonos];

    var x[rowSize][colSize];
    var hasStartedRow[rowSize][maxNonos][colSize];
    var hasStartedCol[colSize][maxNonos][rowSize];
```

- `rowSize` is the grid row size.
- `colSize` is the grid column size.
- `maxNonos` is the maximum number of blocks that a column or a row could have
- `rowNonos` indicates the number of blocks and its sizes in each row.
- `colNonos` indicates the number of blocks and its sizes in eachcolumn.
- `x` is the result board. `x[i][j]` will be true if and only if row `i`, column `j` is coloured and false otherwise.
- `hasStartedRow[i][b][j]` will be true if and only if block `b` of row `i` has already started in the column `j`.

- `hasStartedCol[j][b][i]` will be true if block `b` of column `j` has already started in the row `i`.

The constraints are the same applied over columns and over rows. The following constrains are those applied over rows:

- If a block of a row has started at column i, it also must have started in column i+1.

```
//Order encoding
forall(i in 0..rowSize-1, b in 0..maxNonos-1){
    if(rowNonos[i][b] != 0){
        forall(j in 0..colSize-2){
            hasStartedRow[i][b][j] -> hasStartedRow[i][b][j+1];
        };
    }
    else{
        &&( [!hasStartedRow[i][b][j] | j in 0..colSize-1] );
    };
};

```

- A block mush have started soon enough to fit in the row.

```
forall(i in 0..rowSize-1, b in 0..maxNonos-1){
    if(rowNonos[i][b] != 0){
        hasStartedRow[i][b][colSize-rowNonos[i][b]];
    };
};

```

- `x[i][j]` must be true if it is colored.

```
//Channelling between hasStarted and x
forall(i in 0..rowSize-1, b in 0..maxNonos-1){
    if(rowNonos[i][b] != 0){
        forall(j in 0..colSize-1){
            if(j >= rowNonos[i][b]){
                x[i][j] <- hasStartedRow[i][b][j] & !hasStartedRow[i][b][j↩
    ↪ -rowNonos[i][b]];
            }
            else {
                x[i][j] <- hasStartedRow[i][b][j];
            };
        };
    };
};

```

- The number of cells true in the row `i` must be the sum of the length of the blocks in row `i`

```
forall(i in 0..rowSize-1){
    EK(x[i], sum(rowNonos[i]));
};

```

- Block `b` must start before block `b+1`

```
forall(i in 0..rowSize-1, b in 0..maxNonos-2){
    if(rowNonos[i][b+1] != 0){
        forall(j in 0..colSize-1){
            if(j-rowNonos[i][b]-1 >= 0){
                hasStartedRow[i][b+1][j] -> hasStartedRow[i][b][j-rowNonos[i][b]-1];
            }
            else {
                !hasStartedRow[i][b+1][j];
            };
        };
    };
};

```

Figure 11.1 shows an example of *Nonogram* instance that could be found in the project source code examples.

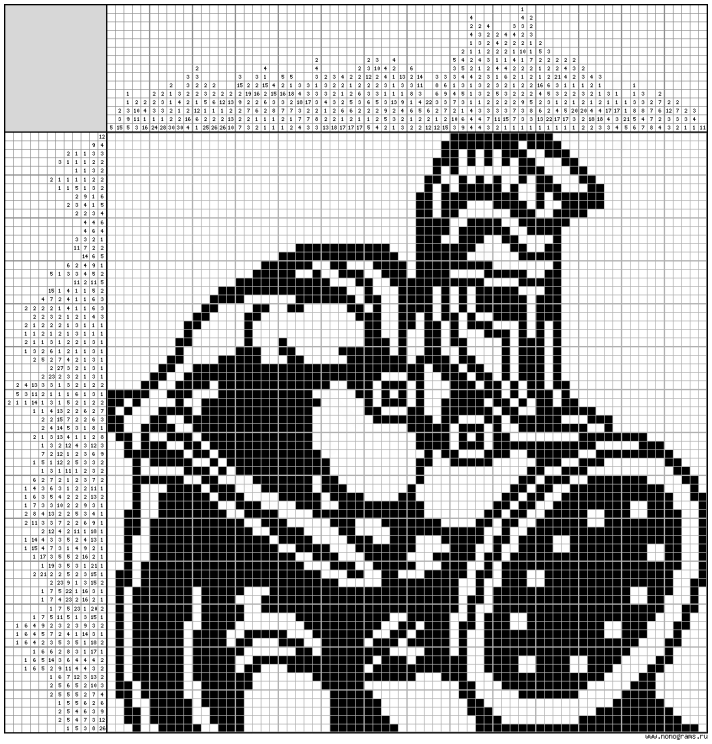


Figure 11.1: Nonogram: gladiator example

11.3.3. Multi-Skill Project Scheduling Problem (MSPSP)

The MSPSP problem is a generalization of RCPSP. The *resource-constrained project scheduling problem* (Kolisch & Sprecher, 1997) is a classical well-known problem in operations research. A number of activities have to be scheduled. Each activity has a duration and cannot be interrupted. There are a set of precedence relations between pairs of activities which state that the second activity must start after the first has finished. The set of precedence relations are usually given as a directed acyclic graph (DAG), where the edge (u,v) represents a precedence relation where u must finish before v begins. The DAG contains two additional activities with duration 0, the source and sink, where the source is the first activity and sink is the last activity (these are *dummy* activities).

There are a set of renewable resources. Each resource has a maximum capacity and at any given time slot no more than this amount can be in use. Each activity has a demand (possibly zero) on each resource. The dummy source and sink activities have zero demand on all resources.

However, in the MSPSP (Coll, 2019) the activities do not directly ask for resources but they ask for skills. These skills are supplied by renewable resources, and every resource is specialised to master a subset of the skills. A clear example is that the resources are workers. A worker can master many skills, and he/she can perform a different skill on each activity. The resource constraints state that one resource (worker) can only work at one skill of one activity at a time, and that a resource can only supply skills that it masters. The resources are unary, i.e. they can only supply one unit of skill at a time, but the activities may require many units of each skill. Also, the set of resources that an activity is using cannot change at any moment of execution, i.e. the resource usage of the activities is also non-preemptive.

The problem is usually stated as an optimisation problem where the makespan (i.e. the completion time of the sink activity) is minimised. *GOS* still does not support optimization, so the solution will be a model that satisfies all the constraints.

MSPSP could be defined as a tuple (V,p,E,R,L,m,b) where:

V is a set of activities A.

p is a vector of naturals, with p_i being the duration of A_i.

E is a set of paris of activities representing end-start precedence relations: $(A_i,A_j) \in E$ iff the execution of activity A_i must precede the activity A_j.

R is a set of unary renewable resources.

L is a set of skills.

$\mathbf{m_{r,l}}$ is a matrix of Booleans, with $m_{i,j}$ being true iff resource R_i masters skill L_j.

$\mathbf{b_{v,l}}$ is a matrix of naturals, where $m_{k,l}$ represents the number of resources mastering skill L_l that activity A_i requires during its execution.

11.3.3.1. Model

This section is a walkthrough explaining how this problem can be modelled with *BUP* and solved using *GOS* step by step.

11.3.3.1.1. Viewpoint The first step consists in defining all the necessary parameters and variables in the viewpoint block:

Listing 11.4: *MSPSP* model: parameters

```
viewpoint:
    param int UB;
    param int nActivities;
    param int nResources;
    param int nSkills;
    param int duration[nActivities+2];
    param int demand[nActivities+2][nSkills];
    param bool successors[nActivities+2][nActivities+2];
    param bool mastersSkill[nResources][nSkills];
```

All the `param`s will be filled with the data in the input file. We define a UB (upper bound) to set the allowed given time to schedule the whole project. The rest of the parameters are the number of activities, resources and skills, the duration of the activities, the skills required for each activity, the activities precedences (which activities must be finished before the start of other activities), and the skills that the available resources masters.

Listing 11.5: *MSPSP* model: variables

```
    var hasStarted[nActivities+2][UB+1];
    var isRunning[nActivities+2][UB];
    var usesResourceForSkill[nActivities+2][nResources][nSkills];
    var usesResourceAtTime[nActivities+2][nResources][UB];
    var usesResource[nActivities+2][nResources];
```

`var`s does not have value when they are declared and they are used to define the model in which the constraints will be applied.

- `hasStarted`: `hasStarted[i][j]` will be true if the activity `i` has started before `j` unit time.
- `isRunning`: `isRunning[i][j]` will be true if the activity `i` is running in the `j` unit time.
- `usesResourceForSkill`: `usesResourceForSkill[i][j][k]` will be true if activity `i` uses the resource `j` for the skill `k`.
- `usesResourceAtTime`: `usesResourceAtTime[i][j][k]` will be true if activity `i` uses resource `j` in `k` unit time.
- `usesResource`: `usesResource[i][j]` will be true if activity `i` uses the resource `j`.

11.3.3.1.2. Constraints The *constraints-block* always start with `constraints:` The constraints to modelate *MSPSP* are:

- Start *dummy*[1] activity starts at time 0, and is never running (due to duration 0).

```
//Dummy start activity
forall(t in 0..UB-1){
    hasStarted[0][t];
    !isRunning[0][t];
};
hasStarted[0][UB];
```

- End activity has started at UB, and is never running.

```
//Dummy end activity
forall(t in 0..UB-1){
    !hasStarted[nActivities+1][t];
    !isRunning[nActivities+1][t];
};
hasStarted[nActivities+1][UB];
```

- *Dummy* activities do not consume resources.

```
forall(r in 0..nResources-1){
    !usesResource[0][r];
    !usesResource[nActivities+1][r];
    forall(s in 0..nSkills-1){
        !usesResourceForSkill[0][r][s];
        !usesResourceForSkill[nActivities+1][r][s];
    };
    forall(t in 0..UB-1){
        !usesResourceAtTime[0][r][t];
        !usesResourceAtTime[nActivities+1][r][t];
    };
};
```

- When an activity starts, it must be started until the end.

```
//Order encoding
forall(i in 1..nActivities, t in 0..UB-1){
    hasStarted[i][t] -> hasStarted[i][t+1];
};
```

- An activity `i` is running for a time period of `duration[i]` after it has started, and is not running at other times.

[1]The model has two *dummy* activities that do not consume resources and skills to indicate the start and the end.

```
//Channelling between hasStarted and isRunning
forall(i in 1..nActivities, t in 0..UB-1){
    if(t >= duration[i]){
        isRunning[i][t] <-> hasStarted[i][t] & !hasStarted[i][t-duration[i
    ↪ ]];
    }
    else{
        isRunning[i][t] <-> hasStarted[i][t];
    };
};
```

- `usesResource[i][r]` is true if and only if activity `i` uses resource `r` for some skill `s`.

```
//Chanelling between usesResource and usesResourceForSkill
forall(i in 1..nActivities, r in 0..nResources-1){
    usesResource[i][r] <-> ||(usesResourceForSkill[i][r]);
};
```

- `usesResourceAtTime[i][r][t]` is true if and only if activity `i` uses resource `r` and is running at time `t`

```
//Chanelling between usesResource, isRunning and usesResourceAtTime
forall(i in 1..nActivities, r in 0..nResources-1, t in 0..UB-1){
    usesResourceAtTime[i][r][t] <-> usesResource[i][r] & isRunning[i][t];
};
```

- The activities must respect the precedences (other activities must have started to start the current activity).

```
//Precedences
forall(i in 0..nActivities, j in 1..nActivities+1){
    if(successors[i][j]){
        forall(t in 0..UB-duration[i]-1){
            !hasStarted[i][t] -> !hasStarted[j][t+duration[i]+1];
        };
    };
};
```

- Resources can only perform skills that they master.

```
forall(i in 1..nActivities, s in 0..nSkills-1, r in 0..nResources-1){
    if(not mastersSkill[r][s]){
        !usesResourceForSkill[i][r][s];
    };
};
```

- Each activity uses as many resources for a skill as necessary.

```
// for a skill as required
forall(i in 1..nActivities, s in 0..nSkills-1){
    EK(usesResourceForSkill[i][_][s],demand[i][s]);
};
```

- Each resource supplies at most one skill to to each activity.

```
forall(i in 1..nActivities, r in 0..nResources-1){
    AMO(usesResourceForSkill[i][r]);
};
```

- Each resource works at most at one activity at a time.

```
forall(r in 0..nResources-1, t in 0..UB-1){
    AMO(usesResourceAtTime[_][r][t]);
};
```

11.3.3.1.3. Output The *output-block* is optional and it is used to define a custom output to the model solution. It always starts with the token `output:`.

```
output:
    "Schedule: \n";
    ["Activity " ++ i ++ " starts at time " ++ t ++ "\n" | i in 1..nActivities,
    ↪  t in 0..UB where (t == 0 ? true : (not hasStarted[i][t-1])) and
    ↪ hasStarted[i][t]];
    ["Activity " ++ i ++ " uses Resource " ++ r ++ " for Skill " ++ s ++ "\n"|
        i in 1..nActivities, r in 0..nResources-1, s in 0..nSkills-1 where
    ↪ usesResourceForSkill[i][r][s]];
```

In this case, the line 2 is the output title. *BUP* allows adding newlines and tabs using `\n` and `\t` respectively.

In the line 3, we generate a comprehension list iterating all activities and times to get the time at which each activity starts.

Line 3 is similar to line 4, but in this case iterating the resources used to supply the necessary skills to the activities.

11.3.3.2. Instance example

This section shows a concrete instance of the problem having:

- 20 activities
- 4 resources
- 10 skills

The way of express all of these data in *GOS* is by creating an input JSON file giving value to the parameters declared in the *BUP* model file:

```
{
    (...)
    "nActivities" : 20,
    "nResources" : 4,
    "nSkills" : 10,
    (...)
}
```

The table 11.1 describes the duration of each activity and the necessary skills to do it. It also includes two *dummy* activities (0 and 21) to indicate the start activity and the end activity. For example, the activity 1 has a duration of 4 time units and requires the skills 4 and 6 to be done. And the activity 13 has a duration of 2 time units and requieres two resources mastering skill 4.

		Skills									
Activity	**Duration**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0 (dummy)	0										
1	4				1		1				
2	2						1				
3	1								1		1
4	2			1					1		
5	3			1		1					
6	3		1								
7	2									1	1
8	4					1					1
9	1							1			
10	2	1									
11	1				1						
12	1					2				1	
13	2						2				
14	1										1
15	2										1
16	2			1					1		
17	4						2				
18	1		1					1			
19	3				1						1
20	2			1					1		
21 (dummy)	0										

Table 11.1: MSPSP instance example: Requiered skills for each activity.

```
{
    (...)
    "duration" : [0,4,2,1,2,3,3,2,4,1,2,1,1,2,1,2,2,4,1,3,2,0],
    "demand" : [
        [0,0,0,0,0,0,0,0,0,0],
        [0,0,0,1,0,1,0,0,0,0],
        [0,0,0,0,0,1,0,0,0,0],
        [0,0,0,0,0,0,0,1,0,1],
        [0,0,1,0,0,0,0,1,0,0],
        [0,0,1,0,1,0,0,0,0,0],
        [0,1,0,0,0,0,0,0,0,0],
```

```
        [0,0,0,0,0,0,0,0,1,1],
        [0,0,0,0,1,0,0,0,0,1],
        [0,0,0,0,0,0,1,0,0,0],
        [1,0,0,0,0,0,0,0,0,0],
        [0,0,0,1,0,0,0,0,0,0],
        [0,0,0,0,2,0,0,0,1,0],
        [0,0,0,0,0,2,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,1],
        [0,0,0,0,0,0,0,0,0,1],
        [0,0,1,0,0,0,0,1,0,0],
        [0,0,0,0,0,2,0,0,0,0],
        [0,1,0,0,0,0,1,0,0,0],
        [0,0,0,1,0,0,0,0,0,1],
        [0,0,1,0,0,0,0,1,0,0],
        [0,0,0,0,0,0,0,0,0,0]],
    (...)
}
```

The table 11.2 shows the activities that can not start until a predecesor activity ends. For example, activities 4, 5 and 6 could not start until the end of activity 1. In other way, after finishing activity 1, activities 4, 5 and 6 could start.

```
{
    (...)
    "successors" : [
        [0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,1,0,0],
        [0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0],
        [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0],
        [0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]],
    (...)
}
```

	Activity successors																				
Activity	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	21
0 (dummy)		■	■	■																		
1					■	■	■															
2												■						■		■		
3					■							■								■		
4													■				■					
5								■														
6								■	■	■												
7																			■	■	■	
8											■		■			■						
9											■					■						
10															■				■			
11														■							■	
12														■				■				
13																			■			
14																	■					
15																	■					
16																		■				
17																					■	
18																						■
19																						■
20																						■
21 (dummy)																						

Table 11.2: MSPSP instance example: Activity successors

The table 11.3 shows the skills mastered for each resource. For example, the resource 1 masters the skills 3, 5, 6, 9 and 10. The resource 1 could be used to supply one unit of the skills it masters to one activity.

	Skills									
Resources	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1			■		■	■			■	■
2		■				■		■	■	■
3	■		■				■	■	■	
4		■		■	■			■		■

Table 11.3: MSPSP instance example: Skills mastered for each resource

```
{
    (...)
    "mastersSkill" : [
        [0,0,1,0,1,1,0,0,1,1],
        [0,1,0,0,0,1,0,1,1,1],
        [1,0,1,0,0,0,1,1,1,0],
        [0,1,0,1,1,0,0,1,0,1]
    ]
}
```

After defining the input JSON file and the *BUP* model file you can run the *GOS* compiler

to obtain the model in the custom output format (defined in the model file).

```
Schedule:

Activity 1 starts at time 0
Activity 2 starts at time 8
Activity 3 starts at time 5
Activity 4 starts at time 7
Activity 5 starts at time 11
Activity 6 starts at time 0
Activity 7 starts at time 17
Activity 8 starts at time 4
Activity 9 starts at time 0
Activity 10 starts at time 9
Activity 11 starts at time 11
Activity 12 starts at time 10
Activity 13 starts at time 14
Activity 14 starts at time 12
Activity 15 starts at time 11
Activity 16 starts at time 14
Activity 17 starts at time 17
Activity 18 starts at time 20
Activity 19 starts at time 21
Activity 20 starts at time 22

Activity 1 uses Resource 1 for Skill 6
Activity 1 uses Resource 4 for Skill 4
Activity 2 uses Resource 2 for Skill 6
Activity 3 uses Resource 1 for Skill 10
Activity 3 uses Resource 3 for Skill 8
Activity 4 uses Resource 1 for Skill 3
Activity 4 uses Resource 3 for Skill 8
Activity 5 uses Resource 1 for Skill 5
Activity 5 uses Resource 3 for Skill 3
Activity 6 uses Resource 2 for Skill 2
Activity 7 uses Resource 3 for Skill 9
Activity 7 uses Resource 4 for Skill 10
Activity 8 uses Resource 2 for Skill 10
Activity 8 uses Resource 4 for Skill 5
Activity 9 uses Resource 3 for Skill 7
Activity 10 uses Resource 3 for Skill 1
Activity 11 uses Resource 4 for Skill 4
Activity 12 uses Resource 1 for Skill 5
Activity 12 uses Resource 2 for Skill 9
Activity 12 uses Resource 4 for Skill 5
Activity 13 uses Resource 1 for Skill 6
Activity 13 uses Resource 2 for Skill 6
Activity 14 uses Resource 4 for Skill 10
Activity 15 uses Resource 2 for Skill 10
Activity 16 uses Resource 3 for Skill 3
Activity 16 uses Resource 4 for Skill 8
```

```
Activity 17 uses Resource 1 for Skill 6
Activity 17 uses Resource 2 for Skill 6
Activity 18 uses Resource 3 for Skill 7
Activity 18 uses Resource 4 for Skill 2
Activity 19 uses Resource 2 for Skill 10
Activity 19 uses Resource 4 for Skill 4
Activity 20 uses Resource 1 for Skill 3
Activity 20 uses Resource 3 for Skill 8
```

		Start time																						
Activity	**Duration**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0 (dummy)	0																								
1	4	r1 s6 r4 s4	r1 s6 r4 s4	r1 s6 r4 s4	r1 s4 r4 s4																				
2	2								r2 s6	r2 s6															
3	1						r1 s10 r3 s8																		
4	2								r1 s3 r3 s8	r1 s3 r3 s8															
5	3												r1 s5 r3 s3	r1 s5 r3 s3	r1 s5 r3 s3										
6	3	r2 s2	r2 s2	r2 s2																					
7	2																		r3 s9 r4 s10	r3 s9 r4 s10					
8	4					r2 s10 r4 s5	r2 s10 r4 s5	r2 s10 r4 s5	r2 s10 r4 s5																
9	1	r3 s7																							
10	2										r3 s1	r3 s1													
11	1												r4 s4												
12	1											r1 s5 r2 s9 r4 s5													
13	2															r1 s6 r2 s6	r1 s6 r2 s6								
14	1													r4 s10											
15	2												r2 s10	r2 s10											
16	2															r3 s3 r4 s8	r3 s3 r4 s8								
17	4																		r1 s6 r2 s6	r1 s6 r2 s6	r1 s6 r2 s6	r1 s6 r2 s6			
18	1																					r3 s7 r4 s2			
19	3																						r2 s10 r4 s4	r2 s10 r4 s4	r2 s10 r4 s4
20	2																							r1 s3 r3 s8	r1 s3 r3 s8
21 (dummy)	0																								

Table 11.4: MSPSP instance example: result chart

The table 11.4 shows the result that the solver resolves. The model result satisfies the constraints:

- Resources are not repeated in the same time slot.
- Activities has the resources that masters the necessary skills.
- The activity precedences are respected.

12. Conclusions

The objective of this project was define a declarative language for modelling Constraint Satisfaction Problems (CSP) and solve them by generating an encoding to Boolean Satisfiability (SAT). This has been accomplished by defining the language *BUP* and implementing its compiler *GOS*.

BUP allows to model any *CSP* to *SAT* using simple and user-friendly declarative expressions and takes a step forward on the path of defining object-oriented declarative languages, by allowing to create structs, called *entities*.

Although the default SAT-solver is *MiniSAT* (Eén & Sörensson, 2004), the compiler uses an API that makes very easy to add new solvers.

Comparing with other constraint modelling languages, we could determine that *Minizinc* and *ESSENCE'* are the ones that most resemble *BUP*. These are Constraint Programming (CP) modelling languages that allow finite domain variables (not only boolean). The *Minizinc* compiler generates an intermediate code, FlatZinc, that could be solved with different backend solvers. In the contrary, *BUP* is a *CSP* modelling language over boolean variables, fully oriented to SAT.

This project has been achieved thanks to the knowledge acquired in the Computer Engineering degree, specially in the last year, in the *Computing* branch:

Declarative Programming: Applications This subject gave me the knowledge about declarative programming as well as a broad vision of how to model and solve *CSPs* using *SAT*. I also learned how to model using Minizinc(Stuckey et al., 2018), a tool for modeling *CSP*s as *Constraint Programming (CP)*.

Compilers This subject gave me knowledge about free-context grammars, regular expressions and how a compiler works and which analysis does in each phase.

From initial planning, has been discarding the option that the compiler automatically generates the mathematical documentation in LaTeX format. Consequently, it has been added as a point for future work. This point required a lot of work and we decided to prioritize the correct functioning of the compiler and the proper specification of the language. On the other hand, a new functionality not initially planned has been added: the online version of the compiler.

The table 12.1 shows the final project schedule considering deviations from initial planning (see the initial planning on 4.1).

	2019				2020							
	December				January				February			
Tasks / Weeks	01	02	03	04	05	06	07	08	09	10	11	12
Project definition												
Study of LRT												
Input format definition												
Grammar definition												
Symbol Table												
Semantic behaviour												
Solver integration												
Error handling												
Output												
Online version												

	2020											
	March				April				May			
Tasks / Weeks	13	14	15	16	17	18	19	20	21	22	23	24
Project definition												
Study of LRT												
Input format definition												
Grammar definition												
Symbol Table												
Semantic behaviour												
Solver integration												
Error handling												
Output												
Online version												

Table 12.1: Gantt chart: Project final schedule

GOS compiler could be used cloning the GitHub repository and following the steps described in the user manual (see Appendix A) or visiting the fully online version.

13. Future Work

This chapter explains how this project could be extended in a future giving ideas appeared during the implementation of the language.

13.1. Optimization

Although the name of the tool is *GOS* (Girona ***Optimization*** System), in the first iteration (the scope of this project) no support for optimization has been considered. Even so, the *API* that *GOS* uses, supports optimization, so it will be one of the first tasks to be done after the completion of this project.

The *API* supports *Weighted Partial MaxSAT* that is an extension of *SAT* where, apart from *hard* clauses (clauses that must be always satisfied), *soft* clauses can also be defined. These *soft* clause can be violated, each at its own associated cost. The goal is to satisfy all the *hard* clauses, and minimize the total cost of the unsatisfied *soft* clauses.

To implement this, it is only necessary to add support to *BUP* to define *soft* clauses with an associated cost.

13.2. Satisfiability Modulo Theories (SMT)

SAT uses propositional logic as the formalization language. That involves a high degree of efficiency but a lower expressiveness. In the other hand, *SMT* uses propositional logic and domain-specific reasoning that improves the expressivity.

SMT allows that the literals of the clauses are not just Boolean variables (or their negation), but also predicates of other theories.

Currently, the API only supports *Linear Integer Arithmetic* (LIA) theory, i.e. linear equations and inequalities with integer variables. For example, a clause could look like $2x_1+3x_2 \leq 4 \vee b$, where x_1 and x_2 are integer variables, and b is a Boolean variable. *BUP* language could be easily extended to support integer variables and *LIA* expressions.

Regarding optimization with *SMT*, the *API* already supports optimization of *LIA* expressions such as $x + 2y - z$, where x, y, z, are integer variables.

There are more *SMT* theories apart from *LIA* that, to be included to *BUP*, would first have to be supported by the *API*, such as *bitvectors*, uninterpreted functions with equality, etc.

13.3. Object-oriented language

As already discussed in other sections, *BUP* lays the foundation stone for creating object-oriented declarative modelling languages by allowing define a kind of *tuples* called *entities* to define parameters and variables.

A further step in this path would be to be able to add local constraints on the variables of an entity, making an entity have restrictions associated. So, in the moment of declaration of the entity in the model you not only would be declaring new variables and parameters but also some restrictions to the model.

The case of the sudoku, for example, you could define an entity called "row" that by itself had the restrictions of not repeating any number.

The possibilities are endless and there are no precedents in the languages that we currently have. Implementing it in this project was ruled out because the research task was very large and uncertain.

13.4. Pseudo-boolean constraints

Another interesting future work is to support *pseudo-boolean* constraints. *Pseudo-boolean* constraints are expressed as:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \,\#\, K$$

where $\# \in \{<, \leq, =, \geq, >\}$ and $\{a_1, a_2, \ldots, a_n\}$ and K are integer values and $\{x_1, x_2, \ldots, x_n\}$ are boolean variables.

The L∧P *API* used to generate the SAT encoding already allows pseudo-booleans constraints and it would not imply a lot of work for *GOS* to support this feature. The *API* allows the inclusion of pseudo-boolean constraints given a K and two lists: a list of coefficients and a list of variables.
Thus, *GOS* could support *pseudo-boolens* in two ways:

- As a predicate: `PB(<listOfSortedCoefs>, <listOfSortedVariables>, K)`
- As an expression: $a_1x_1 + a_2x_2 + \cdots + a_nx_n \,\#\, K$

13.5. Functions

Some models have complex constraints that are repeated in different parts of the model. Having the possibility of defining functions (predicates) that given certain input adds constraints to the model, would simplify many models.

An example of that could be a custom implementation of a cardinality constraint, by defining and using it in the model.

13.6. Different implementations of cardinality constraints

The API used to generate the SAT encoding allows different implementations of some predefined constraints. In the end of this project, *GOS* use the default encodings to translate all the cardinality constraints.

A future work of this project could be add support to being able to select the wanted implementation when you are creating the model. It is as simple as add a new optional parameter or annotation to the *constraint* call allowing the user specify the implementation wanted.

13.7. Mathematical model documentation with LaTeX

This task was initially in the scope of the project, but it was ruled out because the correct functioning of the language and the compiler was prioritized.

It consists on adding an option to the compiler to generate the mathematical documentation of the model given a model file. This could be done by creating a new ANTLR visitor that explores the already generated parse-tree and generates LaTeX code when visiting viewpoint and constraint block nodes.

Furthermore, it could be added a new type of comments, for example, starting with @ like JavaDoc does, that would be ignored by the rest of visitors and would allow the user add more information in the generated documentation file.

Bibliography

Add Gritman, Anthony Ha, Tony Quach, D. W. (2017). *Conflict Driven Clause Learning* (Tech. Rep.).

Ansótegui, C., Bofill, M., Palahí, M., Suy, J., & Villaret, M. (2013, nov). Solving weighted CSPs with meta-constraints by reformulation into satisfiability modulo theories. *Constraints*, *18*(2), 236–268. doi: 10.1007/s10601-012-9131-1

Bessière, C. (2007). Principles and practice of constraint programming–CP 2007: 13th international conference, CP 2007, Providence, RI, USA, September 23-27, 2007 proceedings. *Lecture Notes in Computer Science*(4741), xv, 887. doi: 10.1007/978-3-540-74970-7

Bofill, M., Palahí, M., Suy, J., & Villaret, M. (2012, jul). Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, *17*(3), 273–303. doi: 10.1007/s10601-012-9123-1

Büning, H. K., & Schmitgen, S. (1986). *Prolog*. Wiesbaden: Vieweg+Teubner Verlag. Retrieved from `http://link.springer.com/10.1007/978-3-322-92747-7` doi: 10.1007/978-3-322-92747-7

Coll, J. (2019). *Scheduling Through Logic-Based Tools* (Doctoral dissertation, Universitat de Girona). Retrieved from `http://hdl.handle.net/10803/667963http://creativecommons.org/licenses/by/4.0/deed.ca`

Crockford, D. (2006). The application/json media type for javascript object notation (json). *RFC 4627*.

Eén, N., & Sörensson, N. (2004). An extensible SAT-solver. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *2919*, 502–518. doi: 10.1007/978-3-540-24605-3_37

Federico, T. (2019). Why you should not use (f)lex, yacc and bison - Federico Tomassetti - Software Architect.

Frisch, A. M., Harvey, W., Jefferson, C., Martínez-Hernández, B., & Miguel, I. (2008). Essence: A constraint language for specifying combinatorial problems. *Constraints*, *13*(3), 268–306.

Gherardi, L., Brugali, D., & Comotti, D. (2012). A java vs. c++ performance evaluation: a 3d modeling benchmark. In *International conference on simulation, modeling, and programming for autonomous robots* (pp. 161–172).

Kolisch, R., & Sprecher, A. (1997). PSPLIB - A Project Scheduling Problem Library. *European Journal of Operational Research*, *96*(1), 205-216.

Parr, T. (2010). *Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages*.

Parr, T. (2013). *The definitive antlr 4 reference*. Pragmatic Bookshelf.

Procaccia, A. D. (2008). *Mathematical Foundations of AI* (Tech. Rep.).

Russell, S., & Norvig, P. (2010). *Artificial Intelligence A Modern Approach Third Edition*. Prentice Hall. doi: 10.1017/S0269888900007724

Stephen A. Edwards. (2007). *COMS W4115 Programming Languages and Translators*. Retrieved from `http://www1.cs.columbia.edu/{~}sedwards/classes/2007/w4115-fall/index.html`

Stuckey, P. J., Marrioo, K., & Tack, G. (2018). *MiniZinc Handbook Release 2.2.1* (Tech. Rep.).

Wikipedia. (2020a). *Declarative programming — Wikipedia, the free encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Declarative%20programming&oldid=960959661`. ([Online; accessed 09-May-2020])

Wikipedia. (2020b). *LL parser — Wikipedia, the free encyclopedia*. `http://en.wikipedia.org/w/index.php?title=LL%20parser&oldid=956137467`. ([Online; accessed 04-May-2020])

Zhou, N. F., & Kjellerstrand, H. (2016). The picat-sat compiler. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 9585, pp. 48–62). Springer Verlag. doi: 10.1007/978-3-319-28228-2_4

Acronyms

ANTLR	Another Tool for Language Recognition.
CDCL	Conflict-Driven Clause Learning.
CFG	Context-Free Grammars.
CNF	Conjunctive Normal Form.
CP	Constraint Programming.
CSP	Constraint Satisfaction Problems.
DPA	Deterministic Pushdown Automation.
IID	Iterative and Incremental Development.
LRT	Language Recognition Tools.
REGEX	Regular Expressions.
SAT	Boolean Satisfiability.
SMT	Satisfiability Modulo Theories.

A. Install and Run Instructions

There are two ways of using *GOS*:

Online version Click here and use *GOS* without installing anything.

Downloadable version There is a GitHub repository with the compiler binaries. Using *CMake* is really simple to get the executable.

To get the compiler executable follow the following steps:

1. Clone the *GOS* GitHub repository

```
git clone https://github.com/roger21gm/GOS
```

2. Change current directory to the cloned repository

```
cd GOS
```

3. Create a directory to build the project

```
mkdir build
```

4. Change current directory to the created build directory

```
cd build
```

5. Run CMake to generate the *makefile* according to your OS and hardware

```
cmake ..
```

6. Finally run the *makefile* to generate the executable

```
make
```

Once *make* process ends, the executable `GOS` is generated inside *build* directory. You can use the compiler by adding two input files:

- Model file written in *BUP* (see 8.1)
- Parameters file written in *JSON* (see 8.2)

Having the current directory inside the build directory, to run the compiler and get a solution only write:

```
./gos <path_to_model_file> <path_to_model_file>
```

On the other hand, if you want to get the CNF formula in *DIMACS* format include the option `-pf=1` or `--print-formula`:

```
./gos -pf=1 <path_to_model_file> <path_to_model_file>
```

B. *JSON* input grammar

```
/** Taken from "The Definitive ANTLR 4 Reference" by Terence Parr */

// Derived from http://json.org
grammar JSON;

json
   : value
   ;

obj
   : '{' pair (',' pair)* '}'
   | '{' '}'
   ;

pair
   : STRING ':' value
   ;

arr
   : '[' value (',' value)* ']'
   | '[' ']'
   ;

value
   : STRING
   | NUMBER
   | obj
   | arr
   | 'true'
   | 'false'
   | 'null'
   ;


STRING
   : '"' (ESC | SAFECODEPOINT)* '"'
   ;


fragment ESC
   : '\\' (["\\/bfnrt] | UNICODE)
   ;
```

```
fragment UNICODE
   : 'u' HEX HEX HEX HEX
   ;
fragment HEX
   : [0-9a-fA-F]
   ;
fragment SAFECODEPOINT
   : ~ ["\\\u0000-\u001F]
   ;


NUMBER
   : '-'? INT ('.' [0-9] +)? EXP?
   ;


fragment INT
   : '0' | [1-9] [0-9]*
   ;

// no leading zeros

fragment EXP
   : [Ee] [+\-]? INT
   ;

// \- since - means "range" inside [...]

WS
   : [ \t\n\r] + -> skip
   ;
```

C. *BUP* grammar

```
grammar BUP;

WS
   : [ \t\n\r] + -> skip
   ;

LINE_COMMENT : '//' ~[\r\n]* -> skip;

BLOCK_COMMENT : '/*' .*? '*/' -> skip;

// basic structure
TK_ENTITIES: 'entities';
TK_VIEWPOINT: 'viewpoint';
TK_CONSTRAINTS: 'constraints';
TK_OUTPUT: 'output';

TK_COLON: ':';
TK_SEMICOLON: ';';

TK_UNDERSCORE: '_';

TK_ASSIGN: ':=';

TK_PARAM: 'param';
TK_VAR: 'var';
TK_AUX: 'aux';

TK_CONSTRAINT: 'constraint';

TK_INT_VALUE: ('1'..'9')('0'..'9')* | '0';
TK_BOOLEAN_VALUE: 'true' | 'false';

TK_BASE_TYPE_INT : 'int';
TK_BASE_TYPE_BOOL : 'bool';

TK_IN: 'in';
TK_RANGE_DOTS: '..';

TK_IF: 'if';
TK_ELSEIF: 'else if';
TK_ELSE: 'else';

```

```
43 TK_LPAREN: '(';
44 TK_RPAREN: ')';
45
46 TK_LCLAUDATOR: '[';
47 TK_RCLAUDATOR: ']';
48
49 TK_LBRACKET: '{';
50 TK_RBRACKET: '}';
51
52
53 TK_COMMA: ',';
54 TK_DOT: '.';
55
56 TK_WHERE: 'where';
57
58 TK_FORALL: 'forall';
59
60
61 //EXPRESSIONS
62 TK_OP_AGG_SUM: 'sum';
63 TK_OP_AGG_LENGTH: 'length';
64 TK_OP_AGG_MAX: 'max';
65 TK_OP_AGG_MIN: 'min';
66
67 TK_OP_LOGIC_NOT: 'not';
68 TK_OP_LOGIC_AND: 'and';
69 TK_OP_LOGIC_OR: 'or';
70
71 TK_OP_ARIT_SUM: '+';
72 TK_OP_ARIT_DIFF: '-';
73 TK_OP_ARIT_MULT: '*';
74 TK_OP_ARIT_DIV: '/';
75 TK_OP_ARIT_MOD: '%';
76
77 TK_OP_REL_LT: '<';
78 TK_OP_REL_GT: '>';
79 TK_OP_REL_GE: '>=';
80 TK_OP_REL_LE: '<=';
81 TK_OP_REL_EQ: '==';
82 TK_OP_REL_NEQ: '!=';
83
84 TK_OP_IMPLIC_R: '->';
85 TK_OP_IMPLIC_L: '<-';
86 TK_OP_DOUBLE_IMPLIC: '<->';
87
88 TK_INTERROGANT: '?';
89
90 TK_CONSTRAINT_OR_PIPE: '|';
91 TK_CONSTRAINT_AND: '&';
92 TK_CONSTRAINT_NOT: '!';
93
```

```
TK_CONSTRAINT_AGG_EK : 'EK';
TK_CONSTRAINT_AGG_EO : 'EO';
TK_CONSTRAINT_AGG_ALK : 'ALK';
TK_CONSTRAINT_AGG_ALO : 'ALO';
TK_CONSTRAINT_AGG_AMK : 'AMK';
TK_CONSTRAINT_AGG_AMO : 'AMO';

TK_IDENT: ( ('a'..'z' | 'A'..'Z' | '_')('a'..'z' | 'A'..'Z' | '_' | '0'..'9')* ↩
    ↪ );


//OUTPUT
fragment ESCAPED_QUOTE : '\\"';
TK_STRING :   '"' ( ESCAPED_QUOTE | ~('"') )*? '"';

TK_STRING_AGG_OP: '++';

// SINTÀCTIC
bup: entityDefinitionBlock? viewpointBlock constraintDefinitionBlock ↩
    ↪ outputBlock?;

definition: varDefinition | paramDefinition;

entityDefinitionBlock: TK_ENTITIES TK_COLON entityDefinition* ;
entityDefinition: name=TK_IDENT TK_LBRACKET definition* TK_RBRACKET ↩
    ↪ TK_SEMICOLON;

viewpointBlock: TK_VIEWPOINT TK_COLON definition*;

constraintDefinitionBlock: TK_CONSTRAINTS TK_COLON constraintDefinition*;

outputBlock: TK_OUTPUT TK_COLON (string TK_SEMICOLON)*;

varDefinition: TK_VAR type=TK_BASE_TYPE_BOOL? name=TK_IDENT arrayDefinition ↩
    ↪ TK_SEMICOLON;
paramDefinition: (
        TK_PARAM type=(TK_BASE_TYPE_BOOL | TK_BASE_TYPE_INT)
        | type=TK_IDENT
    ) name=TK_IDENT arrayDefinition TK_SEMICOLON;

arrayDefinition: (TK_LCLAUDATOR arraySize=expr TK_RCLAUDATOR)*;


// EXPRESSIONS

expr:
    exprListAgg #exprTop
    | condition=exprAnd TK_INTERROGANT op1=expr TK_COLON op2=expr #exprTernary;

opAggregateExpr: TK_OP_AGG_LENGTH | TK_OP_AGG_MAX | TK_OP_AGG_MIN | ↩
    ↪ TK_OP_AGG_SUM;
```

```
exprListAgg:
    opAggregateExpr TK_LPAREN list TK_RPAREN #exprListAggregateOp
    | exprAnd #exprAnd2;

exprAnd: exprOr (TK_OP_LOGIC_AND exprOr)*;
exprOr: exprEq (TK_OP_LOGIC_OR exprEq)*;

opEquality: TK_OP_REL_EQ | TK_OP_REL_NEQ;
exprEq: exprRel (opEquality exprRel)*;

opRelational: TK_OP_REL_LT | TK_OP_REL_GT | TK_OP_REL_GE | TK_OP_REL_LE;
exprRel: exprSumDiff (opRelational exprSumDiff)*;

opSumDiff : TK_OP_ARIT_SUM | TK_OP_ARIT_DIFF;
exprSumDiff: exprMulDivMod (opSumDiff exprMulDivMod)*;

opMulDivMod: TK_OP_ARIT_MULT | TK_OP_ARIT_DIV | TK_OP_ARIT_MOD;
exprMulDivMod: exprNot (opMulDivMod exprNot)*;

exprNot: op=TK_OP_LOGIC_NOT? expr_base;

expr_base: valueBaseType | TK_LPAREN expr TK_RPAREN | varAccess;

varAccess: id=TK_IDENT varAccessObjectOrArray*;

varAccessObjectOrArray:
    TK_DOT attr=TK_IDENT
    | TK_LCLAUDATOR index=expr TK_RCLAUDATOR
    | TK_LCLAUDATOR underscore=TK_UNDERSCORE TK_RCLAUDATOR;

valueBaseType: integer=TK_INT_VALUE | boolean=TK_BOOLEAN_VALUE;


// CONSTRAINTS

constraintDefinition: ( forall | ifThenElse | constraint ) TK_SEMICOLON;

auxiliarListAssignation: TK_IDENT TK_IN list;

localConstraintDefinitionBlock: constraintDefinition*;

forall: TK_FORALL TK_LPAREN auxiliarListAssignation (TK_COMMA ↩
    ↪ auxiliarListAssignation)* TK_RPAREN TK_LBRACKET ↩
    ↪ localConstraintDefinitionBlock TK_RBRACKET;

ifThenElse:
    TK_IF TK_LPAREN expr TK_RPAREN TK_LBRACKET localConstraintDefinitionBlock ↩
    ↪ TK_RBRACKET
    (TK_ELSEIF TK_LPAREN expr TK_RPAREN TK_LBRACKET ↩
    ↪ localConstraintDefinitionBlock TK_RBRACKET)*
    (TK_ELSE TK_LBRACKET localConstraintDefinitionBlock TK_RBRACKET)?;
```

```
187
188 list: min=expr TK_RANGE_DOTS max=expr #rangList
189     | TK_LCLAUDATOR listResultExpr TK_CONSTRAINT_OR_PIPE ↩
    ↪ auxiliarListAssignation (TK_COMMA auxiliarListAssignation)* (TK_WHERE ↩
    ↪ condExpr=expr)? TK_RCLAUDATOR #comprehensionList
190     | TK_LCLAUDATOR listResultExpr (TK_COMMA listResultExpr)* TK_RCLAUDATOR #↩
    ↪ explicitList
191     | varAccess #varAccessList;
192
193
194 listResultExpr:
195     varAcc=varAccess
196     | resExpr=expr
197     | constraint_expression
198     | string;
199
200 constraint: constraint_expression | constraint_aggreggate_op;
201
202 constraint_expression: constraint_double_implication;
203
204 constraint_double_implication: constraint_implication (TK_OP_DOUBLE_IMPLIC ↩
    ↪ constraint_implication)*;
205
206
207 implication_operator: (TK_OP_IMPLIC_L | TK_OP_IMPLIC_R);
208 constraint_implication: constraint_or (implication_operator constraint_or)*;
209
210
211 constraint_or: constraint_or_2 (TK_CONSTRAINT_OR_PIPE constraint_or_2)* #↩
    ↪ cOrExpression;
212
213 constraint_or_2:
214     TK_CONSTRAINT_OR_PIPE TK_CONSTRAINT_OR_PIPE TK_LPAREN list TK_RPAREN #↩
    ↪ cOrList
215     | constraint_and #cAnd;
216
217
218 constraint_and: constraint_and_2 (TK_CONSTRAINT_AND  constraint_and_2)* #↩
    ↪ cAndExpression;
219
220 constraint_and_2:
221     TK_CONSTRAINT_AND TK_CONSTRAINT_AND TK_LPAREN list TK_RPAREN #cAndList
222     | constraint_literal #cLit;
223
224 constraint_literal: TK_CONSTRAINT_NOT? constraint_base;
225
226 constraint_base:
227     varAccess
228     | TK_BOOLEAN_VALUE
229     | TK_LPAREN constraint_expression TK_RPAREN;
230
```

```
231 aggregate_op:
232     TK_CONSTRAINT_AGG_EK
233     | TK_CONSTRAINT_AGG_EO
234     | TK_CONSTRAINT_AGG_AMK
235     | TK_CONSTRAINT_AGG_AMO
236     | TK_CONSTRAINT_AGG_ALK
237     | TK_CONSTRAINT_AGG_ALO;
238
239 constraint_aggreggate_op: aggregate_op TK_LPAREN list (TK_COMMA param=expr)? ↩
    ↪ TK_RPAREN;
240
241
242 //OUTPUT
243 string:
244     string concatString
245     | TK_LPAREN string TK_RPAREN
246     | stringTernary
247     | varAccess
248     | expr
249     | list
250     | TK_STRING;
251
252 stringTernary:
253     condition=exprAnd TK_INTERROGANT op1=string TK_COLON op2=string;
254
255 concatString:
256     TK_STRING_AGG_OP string concatString?;
```