Domain-Specific Language for Querying Particle Properties

David Christian Katheder

Master′s Thesis
at the Faculty of Physics
Ludwig–Maximilians–Universität München
Chair of Elementary Particle Physics

Supervisor:
Prof. Dr. Thomas Kuhr

Munich, 08.03.2022

Domänenspezifische Sprache zur Abfrage von Teilcheneigenschaften

David Christian Katheder

Masterarbeit
an der Fakultät für Physik
Ludwig–Maximilians–Universität München
Lehrstuhl für Experimentelle Flavorphysik

Betreuer:
Prof. Dr. Thomas Kuhr

München, 08.03.2022

Chapter 1

Introduction

These days, Big Data is on everyone's lips. Because of the infiltration of the internet into every aspect of our lives, massive amounts of data and metadata are generated every second. Tech giants like Google and Facebook gather, analyze and use this data for advertising purposes.
One Definition for Big Data is that the sheer amount makes it impossible to store, process, and analyze the records with a single computer [1]. This is not a new problem. Particle physics had to deal with Big Data before it was the talk of the town. Particle physics experiments gather large amounts of data of subatomic particle collisions to investigate the fundamental structure of our universe.
From 1998 to 2010, as part of the Belle experiment, the asymmetric electron-positron collider KEKB was operated by the Japanese High Energy Accelerator Research Organization (KEK) in Tsukuba. The facility is also called a *B-factory* since the beam energies were chosen to produce mainly B mesons in the collisions. The Belle experiment closely analyzed the properties of pairs of B and anti-B mesons and confirmed the effect of CP-violation [2].
With technological advances in handling Big Data, particle physics experiments have become more ambitious. For the successor experiment Belle II, the accelerator SuperKEKB and detector were upgraded. The goal is to collect a total of $50\,\mathrm{ab}^{-1}$ which corresponds to 60 Petabytes of raw data [3], which is 50 times more data than the previous Belle experiment. The main objective is to make precision measurements in the B-sector of the standard model and to search for leads to new physics. This huge amount of data needs to be filtered in real-time and stored for later analysis. In addition to large amounts of hardware in computing grids and storage facilities, the proper software for analyzing the stored records is critical.
The Belle II Collaboration has developed a dedicated software framework called *basf2* (**B**elle **II** **A**nalysis **S**oftware **F**ramework) for this purpose. It is used both for offline purposes (e.g., reconstruction and analysis), as well as for online duties (e.g., data acquisition, data quality monitoring, and high-level triggering) [4]. Due to the massive number of registered events, filtering and selection capabilities are essential. This work aims to extend and improve event selection using a domain-specific language.

Chapter 2

The Belle II Experiment

The Belle II Collaboration involves over 984 physicists from 115 institutions in 26 countries [5]. In the following sections, the accelerator and the detector are introduced briefly. For more detailed descriptions, the resources [6–8] can be consulted. This is followed by a description of the Belle II Analysis Software Framework.

2.1 SuperKEKB Accelerator

The electron positron accelerator SuperKEKB is located in Tsukuba, Ibaraki Prefecture, Japan. It is an upgrade of its predecessor KEKB, which was successfully in operation for more than ten years for the Belle experiment. A diagram of the SuperKEKB accelerator facilities is shown in Figure 2.1. It is an asymmetric accelerator, as the colliding electron- and positrons beams have different energies.

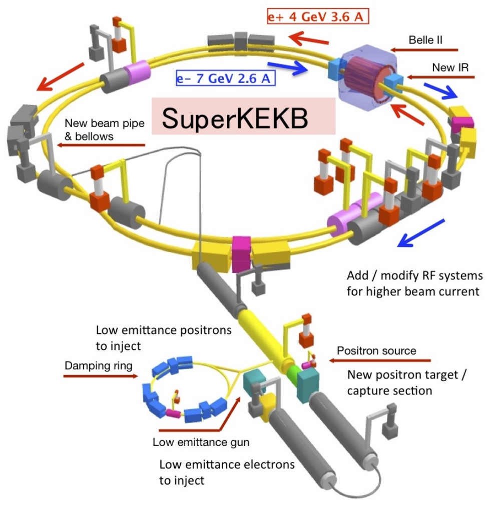

e+ 4 GeV 3.6 A
Belle II
New IR
e− 7 GeV 2.6 A
SuperKEKB
New beam pipe & bellows
Add / modify RF systems for higher beam current
Low emittance positrons to inject
Damping ring
Positron source
New positron target / capture section
Low emittance gun
Low emittance electrons to inject

Figure 2.1: Overview of the SuperKEKB accelerator facilities. Taken from [2].

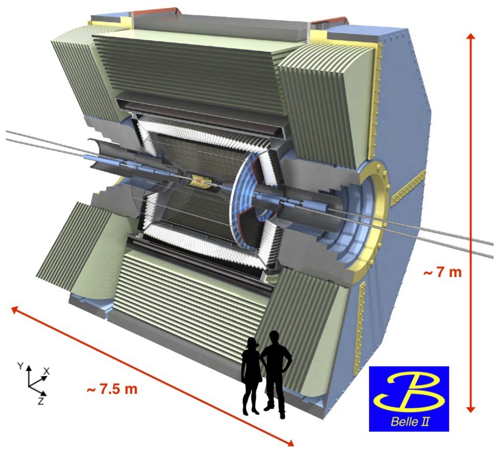
The following beam energies have been chosen:

$$E(e^+) = 4\,\text{GeV}, \qquad E(e^-) = 7\,\text{GeV} \tag{2.1}$$

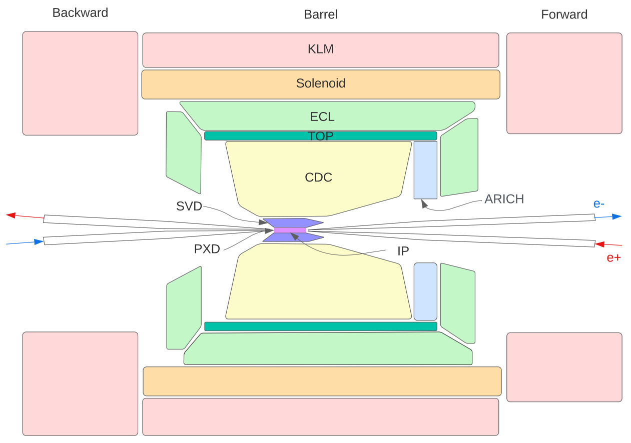
. This choice gives a center-of-mass energy of 10.58 GeV, which coincides with the mass of the Υ(4S) resonance [8].
SuperKEKB is able to deliver e^+e^--collisions with center-of-mass energies from just below the Υ(1S) (9.46 GeV) to the Υ(6S) (11.24 GeV) resonance, yet the majority of the data will be collected at the Υ(4S) resonance [7].
The interaction point where the beams collide is surrounded by the Belle II detector, which measures the result of the particle collisions. The mass of the Υ(4S) resonance is a few MeV above the mass of two charged or neutral B-mesons, allowing it to decay exclusively into B-meson pairs [8].

2.2 Detector

The Belle II detector is specifically designed to make precision measurements of B-physics around the Υ(4S) resonance. An artistic rendering of the detector is shown in Figure 2.2a. The detector is approximately 7 meters in diameter and about 7.5 meters long. A schematic visualization of the detector components is shown in Figure 2.2b. The detector is built up



(a)



(b)

Figure 2.2: (a) Artistic rendering of the Belle II detector, taken from [2]. (b) Schematic top view of the detector depicting its components.

of several layers around the *interaction point* (IP) where the beams collide.
The innermost layer is the *vertex detector* (VXD) which consists of two components. The *pixel detector* (PXD) consists of DEPFET based pixelated sensors [7]. Around it is the *silcon vertex detector* (SVD) which is composed of double-sided silicon strip sensors. They

form the VXD, which is designed to precisely measure the positions of the decay points of B-mesons and other particles.
This is followed by the *central drift chamber* (CDC), which is utilized for tracking. It is used to measure momentum, dE/dx information, and trajectories of charged particles [2].
The Belle II detector has two components for particle identification which utilize the Cherenkov effect to distinguish charged particles. The *time of propagation* (TOP) counter is used in the barrel region and consists of quartz radiator bars. In the forward end-cap, the *aerogel ring-imaging Cherenkov* detector (ARICH) is another component for particle identification. The *electromagnetic calorimeter* (ECL) is utilized for energy measurements, mainly of electrons and photons.
The KLM detector is used to identify K^0_L mesons and muons. A superconducting solenoid magnet generates a homogeneous magnetic field of 1.5 T in which all but the outermost subdetectors are located [6].

2.3 Belle II Analysis Software Framework

basf2 can be categorized into three components. It consists of code specific to the Belle II experiment, dependencies that are provided via the *externals*, and build- and configuration-related tools [9]. The *externals* provide high-energy physics-specific software like *ROOT* and *Geant4*, developed at CERN. However, it is also used to distribute about 90 supplementary Python packages and C++ libraries like boost [7, 10]. *ROOT* is used in *basf2* to read and write event data to files in their custom file format, but the library also provides statistical analysis and visualization tools [7, 11]. *Geant4* is a software package for simulating the passage of particles through matter and is used for generating Monte-Carlo data [12]. The Belle II-specific code is divided into about 40 packages covering the entire high-energy physics workflow, shown in Figure 2.3. It includes, among others, a package for each detector component, the code for track reconstruction, and the tools for post-reconstruction analysis [9].

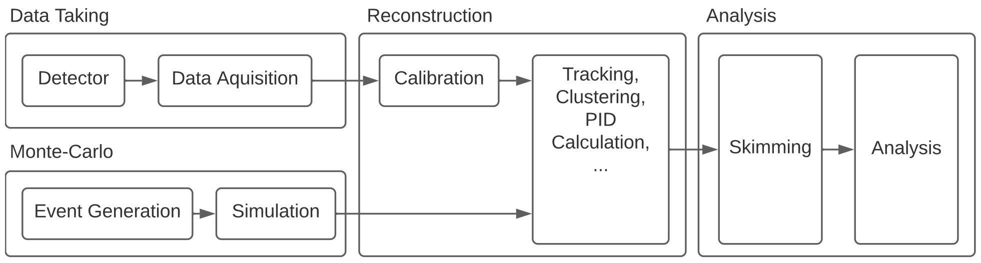


Figure 2.3: Overview of high-energy physics workflow. Adapted from [13].

2.3.1 Architecture

basf2 must be able to process vast amounts of data and operate in a flexible way. Figure 2.4 shows the data processing architecture. *basf2* executes a series of dynamically loaded modules. The modules are arranged in a Path that executes the modules subsequently. Each module encapsulates a data processing method. That can be a simple task like reading data from files but also a very complex task like a full detector simulation [4]. During execution, the modules can interact with a common *DataStore*. The *DataStore* provides access to mutable objects or arrays of objects [7].
The framework core and most of the modules are implemented in C++. *basf2* uses the *boost python* library to create a Python interface. This way, the arrangement and configuration of modules in a path can be made via simple Python scripts called "steering files". Python is a popular programming language for data science and data analysis. The full functionality of the Python language can be used in the steering files. Excellent scientific and data analysis packages like *pandas*, *NumPy* and *SciPy* can easily be integrated into data analytics tasks [14–16]. The combination of C++ and Python is very beneficial because it combines the ease of use of the Python language with the performance of compiled

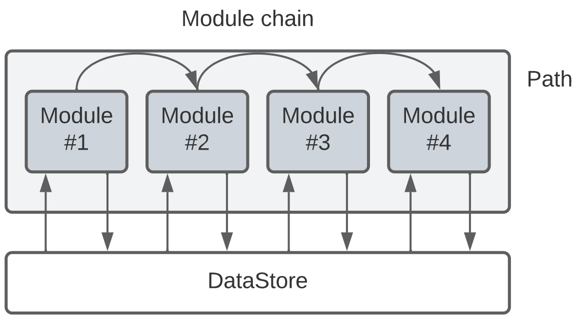


Figure 2.4: The modular architecture of *basf2*. Every processing task is implemented in a module. They are arranged in a Path and are subsequently executed. During execution, they interact with a common *DataStore*. Adapted from [4].

code for complex routines. This allows data analysis scripts to be developed faster without compromising performance too much.

2.3.2 Analysis Variablemanager

The *Manager* class from the *analysis* package handles all attributes and physical properties of *Particle* objects created from reconstructed data objects. There can be kinematic and event-based properties. These are broadly called variables and are centrally managed by the *Manager* class. The modules of the analysis package can read so-called *mdst* files containing reconstructed particles.
The *Manager* class supports different types of variables. We distinguish between standard variables, parameter- and metavariables. The class diagram of the *Manager* and the associated variable class *Var* is shown in Figure 2.5. *Var* has `name`, `group` and `description`

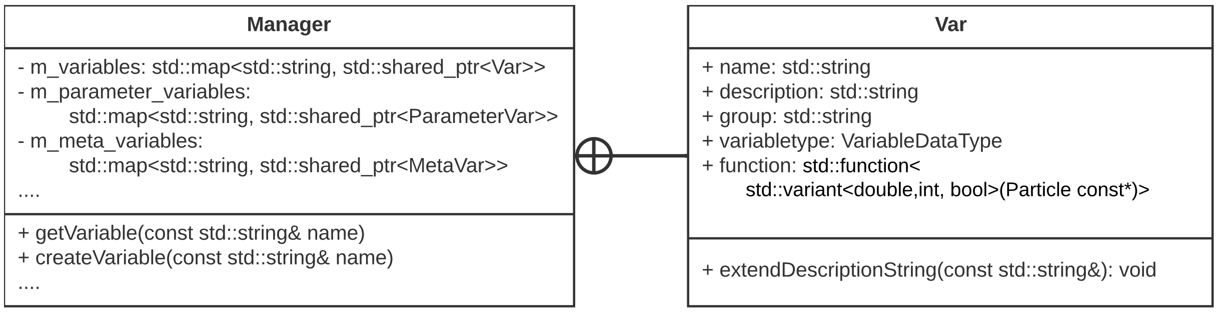

Manager
- m_variables: std::map<std::string, std::shared_ptr<Var>>
- m_parameter_variables:
std::map<std::string, std::shared_ptr<ParameterVar>>
- m_meta_variables:
std::map<std::string, std::shared_ptr<MetaVar>>
....
+ getVariable(const std::string& name)
+ createVariable(const std::string& name)
....
Var
+ name: std::string
+ description: std::string
+ group: std::string
+ variabletype: VariableDataType
+ function: std::function<
std::variant<double,int, bool>(Particle const*)>
+ extendDescriptionString(const std::string&): void

Figure 2.5: The variable manager stores shared pointers to variable objects. The *Var* class has a name and stores a function object which takes a pointer-to-const of *Particle* and returns a value of type `std::variant<double,int,bool>`.

members. The `function` member hosts a `std::function` object that can be called with a pointer-to-const of type *Particle* in order to return the value of the requested property for that object. All registered variables are stored by name in the `m_variables` map. Two additional maps are defined for parameter- and metavariables. Meta- and parameter-variables can be dynamically created and are configurable with user-defined parameters. The name is reminiscent of the concept of metaprogramming. They are frequently used in cuts where they resemble a function call. The function name corresponds to the requested metavariable, and the function arguments are the parameters used to modify the variable's construction. The `getVariable` method searches in the `m_variables` map for the requested *Var* object. If the lookup fails, the function tries to create a new variable dynamically using the `createVariable` method. The variable values can be of type `double`, `int`, and `bool`, so the return value is defined as a C++ `std::variant`. A variant is a type-safe container that can hold different data types. It also provides utility functions to check and extract the variously typed values safely.

2.3.3 Cuts in basf2

The utility class *GeneralCut* located in the *framework* package provides a binary filter functionality. In an analysis or a trigger we want to select particles based on a combination of relevant properties. Selection criteria, so-called "cuts" are provided as a string:

$$\texttt{5.2 < Mbc < 5.29}\ \text{and}\ \texttt{abs(deltaE) < 2.0} \tag{2.2}$$

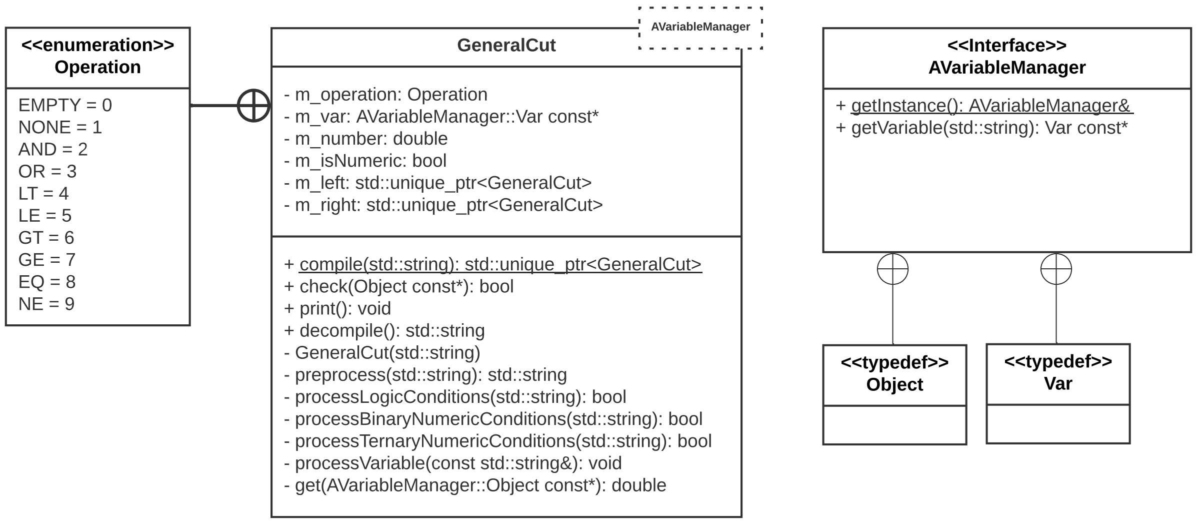
. Valid cuts can contain the following components:

1. Ternary numeric conditions, e.g `5.2 < Mbc < 5.29`
2. binary numeric conditions, e.g `abs(deltaE) < 2.0`
3. composite logic statements
4. variables, e.g `Mbc`
5. parameter- and meta-variables, e.g `abs(deltaE)`
6. numeric literals

All common comparison operators (<, <=, >, >=, ==, !=) can be used to form binary and ternary numeric conditions.
The basic components of the cut are severely limited. Each statement on one side of the comparison must be exactly a number, a variable or a metavariable/parametervariable.

2.3.4 GeneralCut implementation

Binary filter functionality is a basic feature that is needed in different modules for different types of C++ objects. C++ is a strongly typed language, yet it must be achieved that instances of different C++ classes can be filtered with cuts. This should be achieved without duplicating code and classes. Modularity is achieved with C++ template metaprogramming and separation of concerns of string parsing and variable and object management. The *GeneralCut* class template incorporates the parsing implementation and provides the check interface. *GeneralCut* has a private constructor and thus can only be constructed from a string using the static member function `compile`. The class diagram can be seen in Figure 2.6. It is responsible for processing the input into an object which can provide the filtering function for the respective objects. For that is has several string processing methods, `preprocess`, `processLogicConditions`, `processTernaryNumericConditions`, `processBinaryNumericCondtions` and `processVariable`. To derive a cut class, *Gener-*



<<enumeration>>
Operation
EMPTY = 0
NONE = 1
AND = 2
OR = 3
LT = 4
LE = 5
GT = 6
GE = 7
EQ = 8
NE = 9
GeneralCut
AVariableManager
- m_operation: Operation
- m_var: AVariableManager::Var const*
- m_number: double
- m_isNumeric: bool
- m_left: std::unique_ptr<GeneralCut>
- m_right: std::unique_ptr<GeneralCut>
+ compile(std::string): std::unique_ptr<GeneralCut>
+ check(Object const*): bool
+ print(): void
+ decompile(): std::string
- GeneralCut(std::string)
- preprocess(std::string): std::string
- processLogicConditions(std::string): bool
- processBinaryNumericConditions(std::string): bool
- processTernaryNumericConditions(std::string): bool
- processVariable(const std::string&): void
- get(AVariableManager::Object const*): double
<<Interface>>
AVariableManager
+ getInstance(): AVariableManager&
+ getVariable(std::string): Var const*
<<typedef>>
Object
<<typedef>>
Var

Figure 2.6: Definition of *GeneralCut* class template. Variable management and variable-type definitions are provided by dependency injection via template metaprogramming. The *GeneralCut* template can be specialized by injecting a class which satisfies the *AVariable-Manager* interface.

alCut accepts a variable manager type as a template argument. In C++, we can derive classes from templates at compile time by supplying types that are substituted for the template arguments. Template arguments can be integral C++ types but also classes. This allows modularity because *GeneralCut* can be specialized with different variable manager classes. They can each define their own object and variable types but must comply with the *AVariableManager* interface in Figure 2.6. There are primarily two variable managers in *basf2*. One in the *analysis* package and another in the *high-level trigger* (HLT) package. In addition, mock variable manager classes are defined for unittest purposes. The variable manager is defined as a singleton pattern. This ensures that during program execution, only one global instance of the variable manager class exists and can be requested globally [17]. This is important because all valid variables must be registered in the same variable manager object in order to pass pointers to the *GeneralCut* objects via the template argument dependent `getVariable` method. *GeneralCut* has an enumeration member called `m_operation` which designates the type of the cut object. The `check` method provides the binary evaluation interface for the object type defined in the variable manager. The return value is true if the conditions in the cut for the object are met. The `decompile` method is used to convert the cut object back to the cut statement that created it. `print` does the same but passes it to standard output instead of returning it. Cut rules can be arbitrarily nested and naturally have a tree-like structure. *GeneralCut* must map the different constructs in the language. This is achieved using the enum operation and the two pointer-members `m_left` and `m_right`. During compilation, the cut is manually split into left and right sub-statements. Depending on the detected operator, the enum value is set, and child objects are allocated.

2.3.5 Cut processing algorithm

The cut parsing is done in the *GeneralCut* constructor. The schematic structure of the algorithm is shown in Figure 2.7. It starts with preprocessing and cleaning of the text input, which is done by the `preprocess` method. In this step, the text pruning function `boost::algorithm::trim` is used to split off leading and trailing whitespace characters. In addition, global brackets are removed and the cut is trimmed again with `boost::algorithm::trim`.
If the cut is empty after this step, `m_operation` is set to `EMPTY` and the initialization is finished, otherwise processing continues. The return value of `check` is true for any object if the enum value is set to `EMPTY`.
Processing logic condtitions and their combinations with `and`/`or`, is performed by the `processLogicConditions` method. In order to split cuts into the sub-conditions at the right places, the cut is scanned from the beginning for the keywords. During the scanning process, all terms that are in brackets are ignored, which means that nested conditions will be processed later. The precedence of `and` over `or` is achieved by first scanning the cut completely for or-statements. This achieves that the weaker binding `or` operator is broken up preferentially.
In both cases, the cut is split into two substrings which are used to initialize the children `m_left` and `m_right`. The enum value is set to `AND` or `OR` accordingly. In this case, the check function evaluates the two children members recursively and combines their return value according to the enum value.
If no logic conditions are found, an attempt is made to interpret the cut as a ternary numeric comparison. This is done by scanning the string for two consecutive comparison operators, again ignoring all parenthesized subexpressions. The ternary comparison is split into two binary comparisons initializing `m_left` and `m_right`. The enum value is set to `AND` since the ternary comparison is satisfied exactly when both sides are true.
If the search for a ternary comparison fails, an attempt is made to interpret the remaining string as a binary comparison. It is scanned for comparison operators and divided the left and right sides. This initializes the child members and sets the enum value according to the found comparison operator.
If all previous processing functions have not been able to perform any action, it is a basic

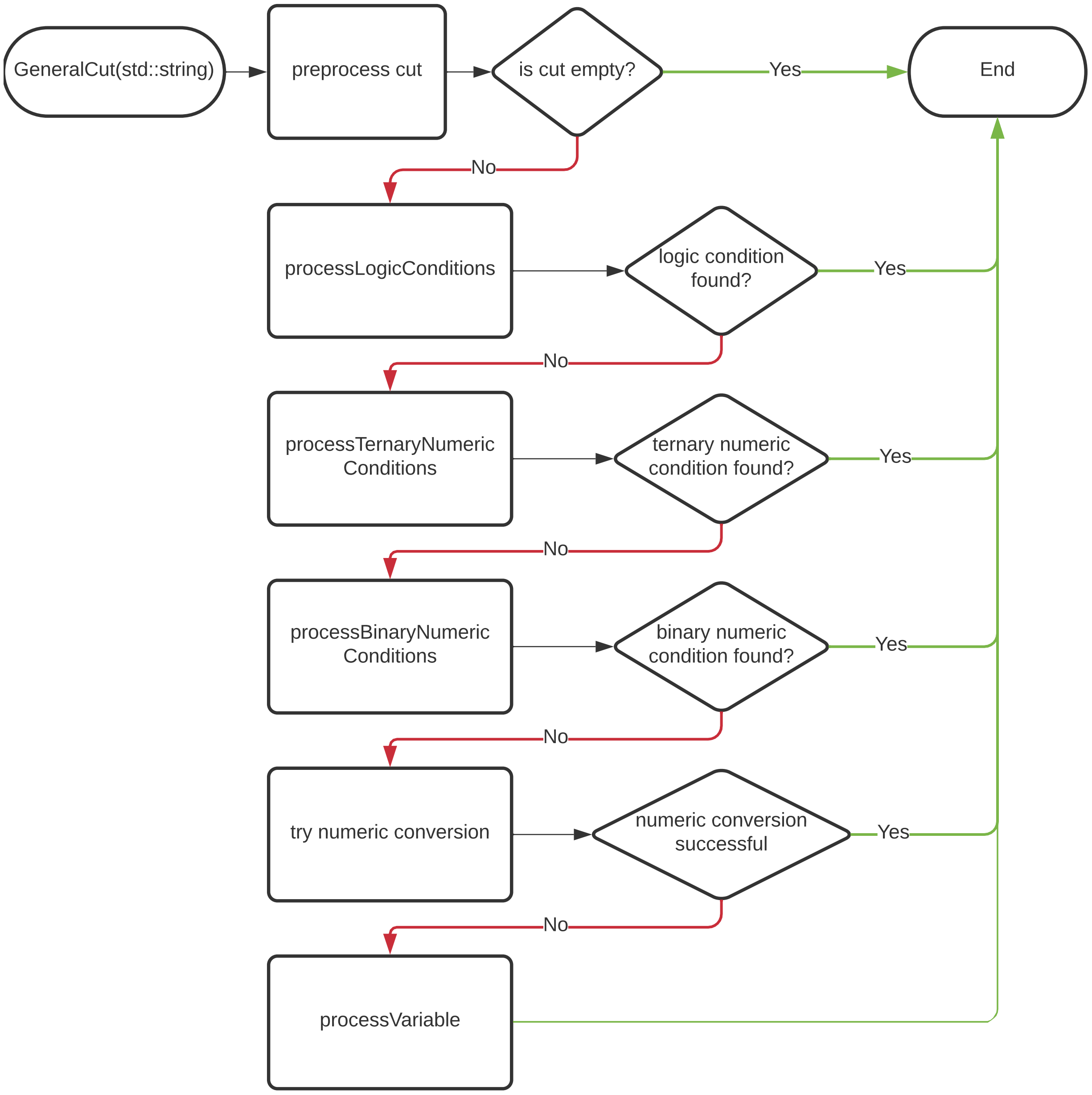


Figure 2.7: Flowchart of *GeneralCut* constructor routine. In order to parse the different cut constructs, multiple helper functions are called in sequence to process input. When the pattern is found, the input is cleaved, and child elements are initialized with the constituents. This initialization is recursive, as it is also carried out for the construction of all child elements. The routine of helper functions is applied until the cut is completely processed.

component and can be either a number, variable or metavariable/parametervariable. The enum value is set to `NONE`, and an attempt is made to convert the string to a number. If successful, the flag `m_isNumeric` is set. When evaluated with `check`, this the numeric value is returned. If the conversion fails, the exception is caught, and `processVariable` is called with the string input.
This method fetches the singleton instance of the variable manager and tries to request a pointer to the associated variable object via `getVariable`. `m_isNumeric` is set to `false`. Fetching the variable pointer may fail to result in a runtime error if the requested name does not exist. When `check` is called with the object to be tested, the `function` member of the variable object is evaluated with the object, returning the value of the object's property.

Limitations It is not possible to use formulas directly in the cut because the processing functions of *GeneralCut* are not able to parse arithmetic.
Each formula must be wrapped with the `formula` metavariable so that it can be processed. The parsing of arithmetic within the `formula` metavariable is done by another utility class, *FormulaParser*, from the *framework* package. This is a significant restriction, which is what the re-implementation is trying to improve. Through that, a much cleaner writing of selection criteria is possible.

Chapter 3

Compiler Structure

Computers are designed to decode simple instructions in binary format and execute them very quickly. Humans need a way to make the required routines understandable to the computer. Compilers serve as translators between humans and machines. They map source code written in programming languages into semantically equivalent machine languages [18]. Programming languages can be divided into several classes [18]:

1. First-generation machine languages encode instructions in binary format, which computers can decode and execute.

2. Second-generation languages are human-readable representations of instructions, e.g., assembly code.

3. Third-generation languages also called higher-level programming languages, which include general purpose languages such as C and Fortran.

4. Fourth-generation languages, also called domain-specific languages, are designed for specific applications and purposes e.g., SQL for relational database queries.

Before the advent of higher-level programming languages, programs had to be written in machine and assembly languages, which is very error-prone and cumbersome. A great achievement of computer science was the development of higher-level programming languages and associated compilers that automatically convert source code into machine code. Higher-level programming languages offer better abstractions of elementary programming elements. In addition, they are better suited to represent the concepts familiar to humans and thus facilitate programming. Furthermore, programs for different computer architectures can be compiled from the same source code [20].

3.1 Overview

The conversion process to machine code is very complex and is therefore divided into several stages and can be seen in Figure 3.1. The first three stages, lexical, syntactic, and

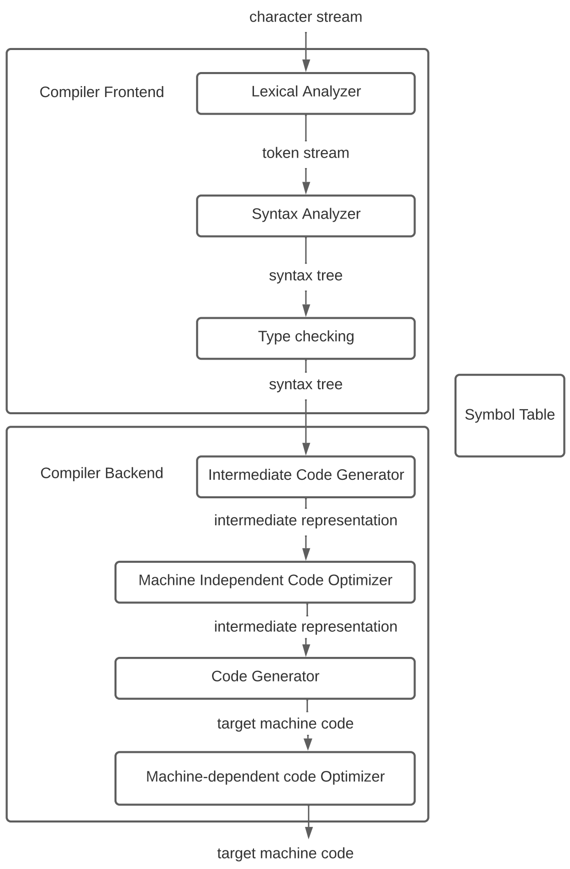


Figure 3.1: Overview of a compilation process of source code. Input processing is a multi-stage process, which can be divided into a compiler frontend and backend. The first stages are lexical analysis and grammar-based synthesis. After that, semantic analysis is applied to the syntax tree. The compiler backend turns this intermediary representation into target machine code in several conversion steps while applying optimizations. Adapted from [18].

semantic analysis, are called the compiler frontend. This brings the source code into an intermediate representation which often has the form of a syntax tree.
Modern compilers are extremely complex programs. They have not only the task to translate into machine language in a semantically correct way but are also expected to perform performance-enhancing code optimizations. The last three phases take over this task and are called compiler backend. This typically transforms the syntax tree into a machine-independent representation and applies code optimizations. This program is then further converted into assembly code via the Code Generator. In the last stage, the assembly code is converted into a machine-dependent binary representation while further optimizations can be made.
For the design of a domain-specific language, a compiler frontend is necessary to create a grammar-based parser. Established software packages exist to automatically generate the

components of a compiler frontend from formal definitions. The basic components of a compiler frontend are presented in the following.

3.2 Compiler Frontend

Semantic analysis is not necessary for our application in a domain-specific language since we do not allocate statically typed symbols and assign values. The first two stages of the compiler frontend are described below.

3.2.1 Lexical Analyzer

A *lexer*, often also called a *scanner*, is a program that reads text input incrementally and groups it into so-called tokens [18]. The character stream is converted into a string of tokens which facilitates the subsequent syntax analysis. In this phase, illegal characters can be detected and reported. In addition, the transition to the token representation eliminates irrelevant formatting characters. An example of tokenization is shown in Figure 3.2. The cut is broken up into its constituent tokens. Optionally, a token can have an attribute value if it represents a datatype or carries a name of an identifier.

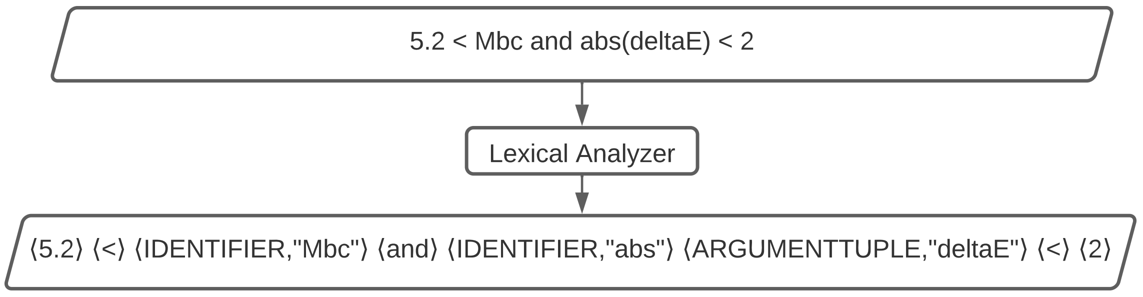

5.2 < Mbc and abs(deltaE) < 2
Lexical Analyzer
⟨5.2⟩ ⟨<⟩ ⟨IDENTIFIER,"Mbc"⟩ ⟨and⟩ ⟨IDENTIFIER,"abs"⟩ ⟨ARGUMENTTUPLE,"deltaE"⟩ ⟨<⟩ ⟨2⟩

Figure 3.2: Tokenization example for a cut. The scanner splits the input into tokens and ignores irrelevant characters such as spaces.

Lexer generators In principle, it is possible to write a scanner program by hand, but this could be very complex and difficult to maintain depending on the scope of application. For this reason, scanners are typically generated automatically by *lexer generators*. These programs can automatically generate source code or routines from a set of token definitions. Tokens are defined with regular expressions and are passed to the lexer generator as a list in a configuration file.
Regular expressions are a notation for describing strings and patterns. There are several metacharacters and constructs to represent patterns. Table 3.1 lists the essential metacharacters. If the actual character should be matched instead of the metacharacter, it has to be

Table 3.1: Explanations of essential metacharacters for regular expressions which are used to describe patterns. Adapted from [19].

Metacharacter	Description
.	...matches any character (except \n and \r).
a*	...represents zero or any number of occurrences of a.
a+	...represents one or any number of occurrences of a.
a | b	...represents the occurrence of a or b.
a?	...represents zero or one occurrence of a.
(ab)	...parentheses can be used to group subexpressions into a unit and apply operations such as +, *, or | to the whole subexpression.
[ABC]	...square brackets are used to describe a selection of characters.
[a-zA-Z]	A hyphen can define a character range when used inside of a square bracket. This example describes the ranges a-z and A-Z.
\d	...is a short-hand notation for [0-9].

escaped with a backslash. There are also shorthand notations for frequently used patterns and formatting characters.

3.2.2 Syntax Analyzer

The purpose of syntax analysis (also known as parsing) is to convert the sequence of tokens into a data structure, the so-called syntax tree, which represents the underlying structure of the input [20].

Context-free grammars

In order to check the sequence of tokens for correctness and to impose structure, fixed rules must first be defined that describe valid compositions of tokens. This set of rules is called grammar and defines the structure of the programming language. The definition of a grammar requires the following ingredients [18]:

1. A set of tokens is usually referred to as terminals. The terminals are all basic building blocks that can occur in the language.

2. A set of non-terminals must be specified. Each non-terminal defines an arrangement of terminals and non-terminals.

3. The composition of each non-terminal must be defined by one or more production rules.

4. A non-terminal must be declared as a start symbol. Typically, this is the first production rule in the grammar.

Each production rule consists of a non-terminal on the left side, also called the head, and a right side called the body [18]. The notation of the grammar is called Backus-Naur-form. In the body, the construct of the non-terminal is defined. If several derivations exist for a non-terminal, they are grouped and can be written compactly, separated with a vertical bar. If a grammar allows the derivation of two different syntax trees, it is ambiguous. One typical example is expression parsing involving operators of different precedence. Whether the ambiguity of a grammar is a problem depends on the area of application. For the domain-specific language, it is essential that the grammar is non-ambiguous. A common strategy to eliminate ambiguity is to add a non-terminal for every precedence level. Non-terminals are ordered in the grammar hierarchy according to their precedence level [18]. A simple grammar for exemplary purposes is shown in Listing 3.1.

Listing 3.1: Toy grammar for parsing expressions which include addition and multiplication of integers.

$$\begin{aligned}
\langle \text{sum} \rangle &::= \langle \text{term} \rangle && (1)\\
&\quad \mid \langle \text{sum} \rangle \texttt{ + } \langle \text{term} \rangle && (2)\\
\langle \text{term} \rangle &::= \texttt{INTEGER} && (3)\\
&\quad \mid \langle \text{term} \rangle \texttt{ * INTEGER} && (4)
\end{aligned}$$

The token set consists of `INTEGER` and the two operators for addition and multiplication. It defines the start symbol *sum*. Operator precedence of multiplication is achieved by introducing the *term* non-terminal. In bottom-up parsing, the multiplication rules are going to be applied first. A syntax tree for an example input is illustrated in Figure 3.3. The tree nodes represent the applied grammar rules, and the tree structure implicitly contains the order in which the operations must be evaluated.

Predictive parsing

The goal of predictive parsing is to construct programs that are able to generate a syntax tree automatically according to the grammar definition. A possible construction is the *LR-parser* which uses stack automata generated from the grammar rules to process the tokens [20]. *LR-parsers* are often also called shift-reduce parsers. The stack holds information about parsed symbols and already made state transitions. At each processing step, either a *shift*, *reduce* or *goto* action can be performed [20]:

- *shift* reads a token from the input and pushes it on the stack together with the state transition.
- *reduce* replaces N top elements on the stack with the corresponding non-terminal if they match a body of a production rule. After the *reduce* action, the parser is in the topmost state in the stack.

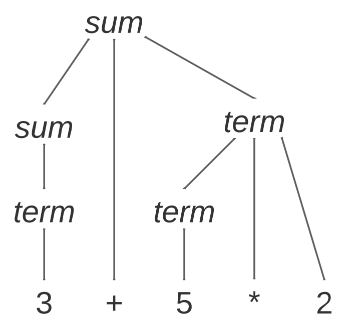


Figure 3.3: Derived syntax tree according to the example grammar. The tree nodes correspond to the production rules, and the structure implicitly contains the order in which the operations must be evaluated. The leaves of the tree correspond to the token sequence.

- *goto* is executed when encountering non-terminals on top of the stack. The parser switches to the specified state.

Production rules are applied bottom-up with the goal to arrive at the start symbol. The chosen action in each step is derived from decision tables, which can be constructed from the grammar. If the input is not legal, there will be a situation during parsing in which no legal action can be chosen. Table 3.2 shows the parser table for the grammar, which is specified in Listing 3.1. Each row represents a parser state, and each entry is a valid

Table 3.2: Parsing table for grammar specified in Listing 3.1.

State	INTEGER	PLUS	TIMES	$	sum	term
0	s3				g1	g2
1		s4		a		
2		r1	s5	r1		
3		r3	r3	r3		
4	s3					g6
5	s7					
6		r2	s5	r2		
7		r4	r4	r4		

transition from this state. The columns each represent the elements at the top of the stack. These are all terminals, non-terminals, and the end symbol $. In each parsing step, either a *shift s*, *reduce r*, or *goto g* action can be performed.
The state change for *shift* and *goto* are indicated in the table by the following numbers. The numbers in the reduce table entries indicate which grammar rule is applied and corresponds to the enumeration in Listing 3.1.

Table 3.3: Example of a shift-reduce parser routine for grammar in Listing 3.1, which executes the rules specified in parsing table 3.2. Tokens are shifted onto a stack together with the current state information (indicated in **bold**). An overview of the parser state, stack, lookahead token (LA), and the unscanned input is given for every step.

Step	Current state	Stack	LA	Unscanned	Action
0	**0**	**0**	3	3 + 5 * 2 $	s3
1	**3**	**0** 3 **3**	+	+ 5 * 2 $	r3
2	**0**	**0** term(3)	+	+ 5 * 2 $	g2
3	**2**	**0** term(3) **2**	+	+ 5 * 2 $	r1
4	**0**	**0** sum(3)	+	+ 5 * 2 $	g1
5	**1**	**0** sum(3) **1**	+	+ 5 * 2 $	s4
6	**4**	**0** sum(3) **1** + **4**	5	5 * 2 $	s3
7	**3**	**0** sum(3) **1** + **4** 5 **3**	*	* 2 $	r3
8	**4**	**0** sum(3) **1** + **4** term(5)	*	* 2 $	g6
9	**6**	**0** sum(3) **1** + **4** term(5) **2**	*	* 2 $	s5
10	**5**	**0** sum(3) **1** + **4** term(5) **2** * **5**	2	2 $	s7
11	**7**	**0** sum(3) **1** + **4** term(5) **2** * **5** 2 **7**	$	$	r4
12	**4**	**0** sum(3) **1** + **4** term(10)	$	$	g6
13	**6**	**0** sum(3) **1** + **4** term(10) **6**	$	$	r2
14	**0**	**0** sum(13)	$	$	g1
14	**1**	**0** sum(13) **1**	$	$	a

Furthermore, there is the *accept* action *a*, which signals that the input parsing was successful. An example of a shift-reduce parser is given in Table 3.3. It shows all parser actions and state transitions for an exemplary input. Parser generator frameworks allow binding of auxiliary functions, which are executed when reductions are applied. This can be used to recursively build a syntax tree data structure.

Chapter 4

Domain-specific language

4.1 Choice of lexer and parser generators

One of the most popular lexer generator is *lex*, implemented in 1975 at Bell Laboratories [21]. *lex* and its newer implementation *flex* are widely distributed and available on most UNIX-based systems [21]. It is an executable that translates a configuration file into C source code. The source code can be compiled into a lexer program with a C-compiler. It is commonly used with the parser generator *GNU Bison* to create compiler frontend source code in C/C++. They have their language for writing configuration files which are processed by the executables, generating source code. This procedure would need to be integrated into the build process of *basf2*. Today, Python packages implement the same algorithms but are easier to use. In terms of comprehensibility, choosing a well-documented Python library that implements the same algorithms is preferred. A Python-based scanner and parser generator framework are relatively easy to integrate because *basf2* already supports external Python dependencies via the *externals*. In addition, *basf2* already uses the *boost python* library to achieve a hybrid C++/Python architecture. It can also be used to embed the Python interpreter in C++ modules to run a compiler frontend implemented in Python.
The *SLY* package was chosen to implement the compiler frontend [22]. It is a lightweight package without dependencies. It provides two classes, `sly.Lexer` and `sly.Parser`, which can be derived from to create own lexer and parser implementations.

4.2 Scanner

Token definition The allowed tokens have to be defined for the domain-specific language. Table 4.1 shows the complete list of tokens. The constructs needed for this are the following:

1. Structure tokens. Square brackets group logical statements. We define `LBRACK` and `RBRACK` for opening and closing square brackets. In addition, round brackets are used

Table 4.1: Token specification for the domain-specific language.

Token	Regular expressions
LBRACK	\\[
RBRACK	\\]
LPAREN	\\(
RPAREN	\\)
EQUALEQUAL	==
GREATER	>
LESS	<
GREATEREQUAL	>=
LESSEQUAL	<=
NOTEQUAL	!=
PLUS	\\+
MINUS	-
TIMES	*
DIVIDE	/
POWER	(**|^)
IDENTIFIER	[a-zA-Z_][a-zA-Z_0-9]*
INTEGER	(0x[0-9a-fA-F]+|\\d+)
DOUBLE	(?i)((\\d+\\.\\d*|\\d*\\.\\d+)(e(-|\\+)?\\d+)?\\d+(e(-|\\+)?\\d+))
BOOLEAN	(true|True|false|False)
ARGUMENTTUPLE	

in formulas to group subexpressions. For this, the tokens LPAREN and RPAREN are defined.

2. Comparison operators. We want to allow all standard comparisons in cuts and add regular expressions for all operators.

3. Boolean Operators. It should be possible to combine logical expressions using the boolean operators and & or. It should also be possible to negate logical expressions using the unary operator not.

4. Arithmetic operators. All standard arithmetic operators must be tokenized in order to parse formulas correctly. The POWER token stands for the exponentiation operator, and both the Python notation and the circumflex character are possible.

5. Data types. Literals of different data types should be possible in cuts. DOUBLE tokenizes floating-point numbers in all possible notations, including scientific notations The INTEGER token allows hexadecimal representations, besides the normal notation for integers.

6. Identifiers and reserved keywords. The IDENTIFIER token is used to match all names in a cut. Many reserved keywords overlap with the IDENTIFIER token definition.

Their token attributes and types can be remapped to the correct values in the `IDENTIFIER` scanning function.

The *basf2* framework allows the user to create new objects with self-selected names in many places, such as new particle lists. Some metavariables accept these names as arguments and return properties related to them. This makes it impossible to tokenize the metavariable arguments since the self-selected names do not necessarily match the specified token definitions. This is the reason why the token `ARGUMENTTUPLE` is not defined via a regular expression. An escape mechanism is provided in the scanning procedure to allow arbitrary arguments. This works similarly to the way multi-line comments work in C++. There is a well-defined start and end of the sequence of characters that are supposed to be matched. Metavariables in cuts always start with a valid `IDENTIFIER` token. If the immediately following token is an opening parenthesis, it finds the matching closing parenthesis in the string, and everything in between is captured as an `ARGUMENTTUPLE` token. This substring is then split on the comma delimiter to produce a list of the individual arguments used for variable creation.

***SLY* implementation** Part of the *lexer* class implementation is shown in Listing 1 in the appendix. The whole set of tokens needs to be declared in the class body. For each token, a regular expression can be set via an identically named class attribute or a method. The regular expressions are bound to the scanning methods with a decorator. The *SLY Lexer* class has a `tokenize` method that receives a string as an argument and returns a Python generator from which the tokens can be yielded.

4.3 Parser

In the following, the grammar of the domain-specific language is explained. It also briefly described how the parser is created from the grammar using the *SLY* package.

4.3.1 Grammar Definition

The complete grammar definition for the domain-specific language is shown in Listing 4.1. Tokens are denoted in capital letters. Non-terminals are written lowercase wrapped in angular brackets. Literal characters and keywords are enclosed in single quotation marks. The grammar rules are numbered and can be divided into two logically separated units.

```
               ⟨cut⟩  ::=  ϵ                                                          (1)
                         | ⟨boolean_expression⟩                                       (2)
⟨boolean_expression⟩  ::=  ⟨disjunction⟩                                              (3)
       ⟨disjunction⟩  ::=  ⟨conjunction⟩                                              (4)
                         | ⟨disjunction⟩ 'or' ⟨conjunction⟩                           (5)
       ⟨conjunction⟩  ::=  ⟨negation⟩                                                 (6)
                         | ⟨conjunction⟩ 'and' ⟨negation⟩                             (7)
          ⟨negation⟩  ::=  ⟨bracket_expression⟩                                       (8)
                         | 'not' ⟨negation⟩                                           (9)
⟨bracket_expression⟩  ::=  ⟨relational_expression⟩                                   (10)
                         | '[' ⟨boolean_expression⟩ ']'                              (11)
⟨relational_expression⟩ ::= ⟨expression⟩                                             (12)
                         | ⟨expression⟩ ⟨comparison_operator⟩ ⟨expression⟩           (13)
                         | ⟨expression⟩ ⟨comparison_operator⟩ ⟨expression⟩           (14)
                           ⟨comparison_operator⟩ ⟨expression⟩                        (15)
⟨comparison_operator⟩ ::=  '==' | '>' | '<' | '>=' | '<=' | '!='                     (16)
```

```
⟨expression⟩ ::= ⟨sum⟩                 (17)     ⟨power⟩ ::= ⟨primary⟩               (27)
       ⟨sum⟩ ::= ⟨term⟩                (18)               | ⟨primary⟩ POWER
               | ⟨sum⟩'+'⟨term⟩        (19)                 ⟨factor⟩                (28)
               | ⟨sum⟩'-'⟨term⟩        (20)   ⟨primary⟩ ::= '('⟨expression⟩')'      (29)
      ⟨term⟩ ::= ⟨factor⟩              (21)               | ⟨function⟩              (30)
               | ⟨term⟩'*'⟨factor⟩     (22)               | IDENTIFIER              (31)
               | ⟨term⟩'/'⟨factor⟩     (23)               | INTEGER                 (32)
    ⟨factor⟩ ::= ⟨power⟩               (24)               | BOOLEAN                 (33)
               | '+'⟨factor⟩           (25)               | DOUBLE                  (34)
               | '-'⟨factor⟩           (26)  ⟨function⟩ ::= IDENTIFIER
                                                            ARGUMENTTUPLE           (35)
```

Listing 4.1: Complete grammar specification of the domain-specific language for cuts.

The upper part of the grammar (1-16) defines the composition and combination of logic conditions. The non-terminal *cut* is the start symbol. We want to allow empty strings as cuts, which is why rule (1) defines an empty production (denoted with the ϵ-character) for the start symbol.
Otherwise, a cut is composed of something which is reduced to a *boolean_expression*.

The base for logic conditions are defined in the non-terminal *relational_expression* (12-15). They can either consist of a singular expression or binary and ternary comparisons. Any token can be substituted for the comparison operators, which is specified in rule (16). The comparisons yield boolean results, and singular expressions are cast to boolean values.
Logic conditions should be able to be combined with boolean operators. This is represented by the non-terminal *conjunction* and *disjunction*. The negation for logic conditions is represented by rule (9).
Logic conditions can be grouped using square brackets to override operator precedences. This is achieved by rule (12). There, a *boolean_expression* enclosed in square brackets is reduced to the non-terminal *bracket_expression*. This is a deliberate wraparound in grammar. *boolean_expression* can be a reduced form of *conjunctions*, *disjunctions*, *negations*, and *bracket_expressions*. This allows arbitrary nested combinations of conditions in square brackets.
Rules (17-35) form the formula part of the grammar and are reduced to the non-terminal *expression* used in the comparisons.
The basic building blocks that can occur in formulas as a unit are defined by the non-terminal *primary* (29-35). They can consist of the tokens `IDENTIFIER`, `INTEGER`, `BOOLEAN`, `DOUBLE`, and of the compound non-terminal *function*. In addition, a wraparound is used again to integrate expressions enclosed in parentheses.
The remaining grammar rules allow the parsing of arithmetic. Again, the grammar hierarchy corresponds to the operators' precedence starting with lowest to highest.

***SLY* implementation** Listing 2 in the appendix shows a part of the parser implementation using the *SLY* package. The token set must be specified. Operator associativity and precedence are set via the `precedence` class attribute. The grammar rules are bound to the parsing functions with a decorator. The parsing functions create a syntax tree as a nested Python tuple. The `sly.Parser` class exposes the method `parse`. This method accepts the Generator object produced by the scanner's `tokenize` function and returns the syntax tree representation.

4.3.2 Syntax tree

The grammar-based parser replaces the cut processing routine described in section 2.3.5. The structures in the language can be represented by a set of node classes which will be introduced in the following section. In the syntax tree representation, arithmetic operators, node types, and comparison operators are encoded as numbers. Each tuple represents a node, where the first argument encodes the node type. In Figure 4.1, the encoding schemes for the node types and operators are shown. Moreover, Equation 4.1 gives an example of the conversion to the syntax tree.

$$\text{"E} > 2.3\text{"} \rightarrow (0, (3, (8, \text{"E"}), (9, 2.3), 3), \text{False}, \text{False}) \tag{4.1}$$

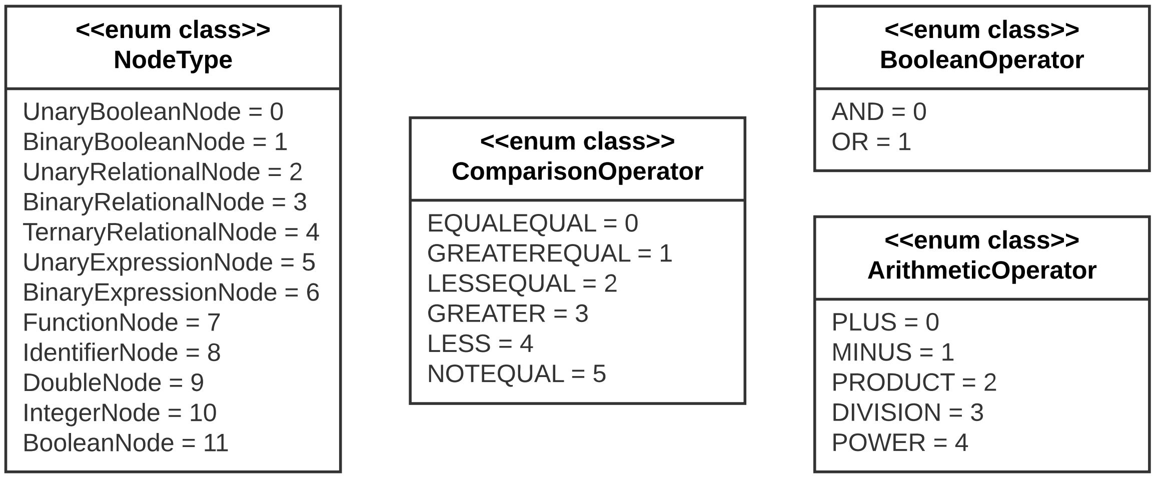


Figure 4.1: In the syntax tree, operators and node types are encoded as numbers.

4.4 CutNodes

In the previous implementation, all nodes are represented by the *GeneralCut* template class in which the enum `m_operation` controls the node type. The enum would need more values to represent the nodes for capturing formulas. This would also lead to more case handling in the member functions of *GeneralCut* and bloat the implementation.
In order to achieve better encapsulation, new node classes are introduced. Every class only captures one language construct. Thus, the `print`, `decompile` and `check` implementations become simple since they are defined for each class separately.
The various language elements can be divided into a few concepts and functionalities. They are classified into *boolean* and *expression* nodes. Boolean nodes take on the task of mapping logic conditions and are used for structures that combine statements with operators. Expression nodes are there to map the constituents of expressions. They contain nodes for formulas, constants, and variables.
The outsourcing of functionality to the node classes simplifies the implementation of *GeneralCut*. The class has only one pointer member to the root of the tree. Therefore, the call to `check`, `print`, and `decompile` is passed on recursively to the tree's root.
All classes are templates analogous to *GeneralCut* and accept a variable manager as a template argument. This is necessary because the functions `check` and `evaluate` implemented by the nodes must access variable manager dependent type definitions. Abstract base classes *AbstractBooleanNode* and *AbstractExpressionNode* are defined to achieve polymorphism. We can define pointers to abstract classes, which can accept pointers to any derived class.

4.4.1 Boolean nodes

In the following, the boolean nodes are explained. Figure 4.2 shows the class diagrams.

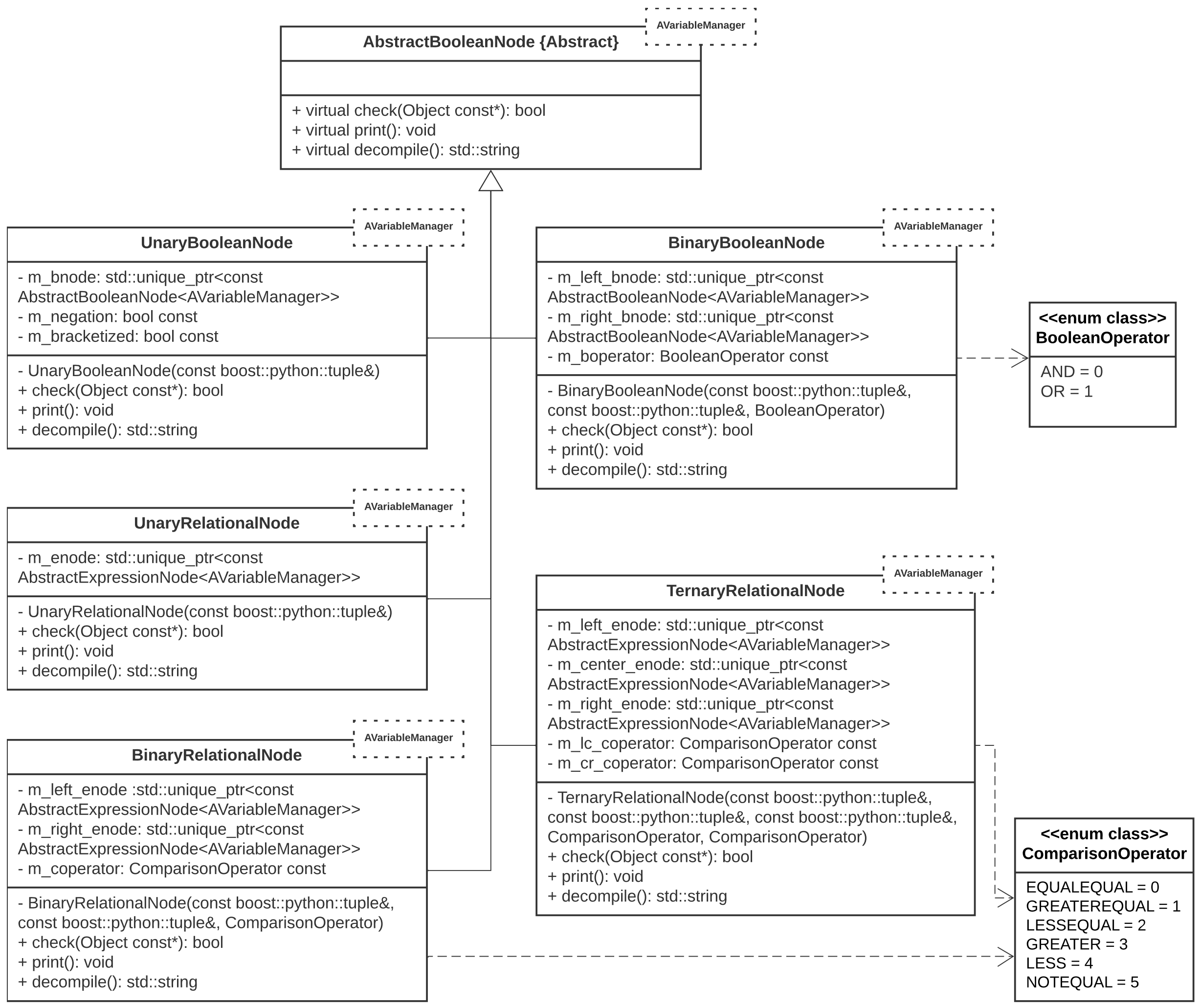

AVariableManager
AbstractBooleanNode {Abstract}
+ virtual check(Object const*): bool
+ virtual print(): void
+ virtual decompile(): std::string
AVariableManager
UnaryBooleanNode
- m_bnode: std::unique_ptr<const AbstractBooleanNode<AVariableManager>>
- m_negation: bool const
- m_bracketized: bool const
- UnaryBooleanNode(const boost::python::tuple&)
+ check(Object const*): bool
+ print(): void
+ decompile(): std::string
AVariableManager
BinaryBooleanNode
- m_left_bnode: std::unique_ptr<const AbstractBooleanNode<AVariableManager>>
- m_right_bnode: std::unique_ptr<const AbstractBooleanNode<AVariableManager>>
- m_boperator: BooleanOperator const
- BinaryBooleanNode(const boost::python::tuple&, const boost::python::tuple&, BooleanOperator)
+ check(Object const*): bool
+ print(): void
+ decompile(): std::string
<<enum class>> BooleanOperator
AND = 0
OR = 1
AVariableManager
UnaryRelationalNode
- m_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- UnaryRelationalNode(const boost::python::tuple&)
+ check(Object const*): bool
+ print(): void
+ decompile(): std::string
AVariableManager
TernaryRelationalNode
- m_left_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_center_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_right_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_lc_coperator: ComparisonOperator const
- m_cr_coperator: ComparisonOperator const
- TernaryRelationalNode(const boost::python::tuple&, const boost::python::tuple&, const boost::python::tuple&, ComparisonOperator, ComparisonOperator)
+ check(Object const*): bool
+ print(): void
+ decompile(): std::string
AVariableManager
BinaryRelationalNode
- m_left_enode :std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_right_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_coperator: ComparisonOperator const
- BinaryRelationalNode(const boost::python::tuple&, const boost::python::tuple&, ComparisonOperator)
+ check(Object const*): bool
+ print(): void
+ decompile(): std::string
<<enum class>> ComparisonOperator
EQUALEQUAL = 0
GREATEREQUAL = 1
LESSEQUAL = 2
GREATER = 3
LESS = 4
NOTEQUAL = 5

Figure 4.2: Node classes which inherit from *AbstractBooleanNode*. Each derived class implements the member functions `check`, `print`, and `decompile`.

1. **UnaryBooleanNode** has a unique pointer of type *AbstractBooleanNode* and thus can hold a pointer to a nested logic condition. Conditions can be negated and enclosed in square brackets. *UnaryBooleanNode* implements this functionality and represents the grammar rules (9) and (11). We need to save the positioning of square brackets because decompile must output an exact reproduction of the original input. The flag `m_bracketized` controls whether square brackets are added during decompile and print. If `m_negation` is true, the recursive evaluation using `check` negates the return value.

2. **BinaryBooleanNode** has two pointers of type *AbstractBooleanNode* and an enum member referring to a boolean operator. This class evaluates two conditions and combines their return values with the respective boolean operator. This corresponds to grammar rules (5) and (7).

3. **UnaryRelationalNode**, **BinaryRelationalNode** and **TernaryRelationalNode** implement grammar rule (12-14). They have unique pointer members to *AbstractExpressionNode* and a matching number of comparison operator enum members. Their respective `check` functions call the `evaluate` function of the expression nodes and perform the comparisons.

4.4.2 Expression nodes

The class diagram of all expression nodes is shown in Figure 4.3. The following explains the purpose of each class:

1. **UnaryExpressionNode** has one unique pointer to an *AbstractExpressionNode* and two boolean flags for remembering parenthesis and information concerning a unary minus. In the `evaluate` function, the sign of the result is flipped if `m_unary_minus` is set to true. We need to be able to reproduce parenthesis in cuts identically with `decompile` and `print`. Parenthesized expressions are wrapped into a *UnaryExpressionNode* with `m_parenthesized` set to true. The member `m_enode` points to the expression contained inside of the parentheses.

2. **BinaryExpressionNode** takes on the task of evaluating arithmetic. It has two pointers to *AbstractExpressionNode* and an *ArithmeticOperator* enum value. In the `evaluate` function, the operation is performed with the evaluation results.

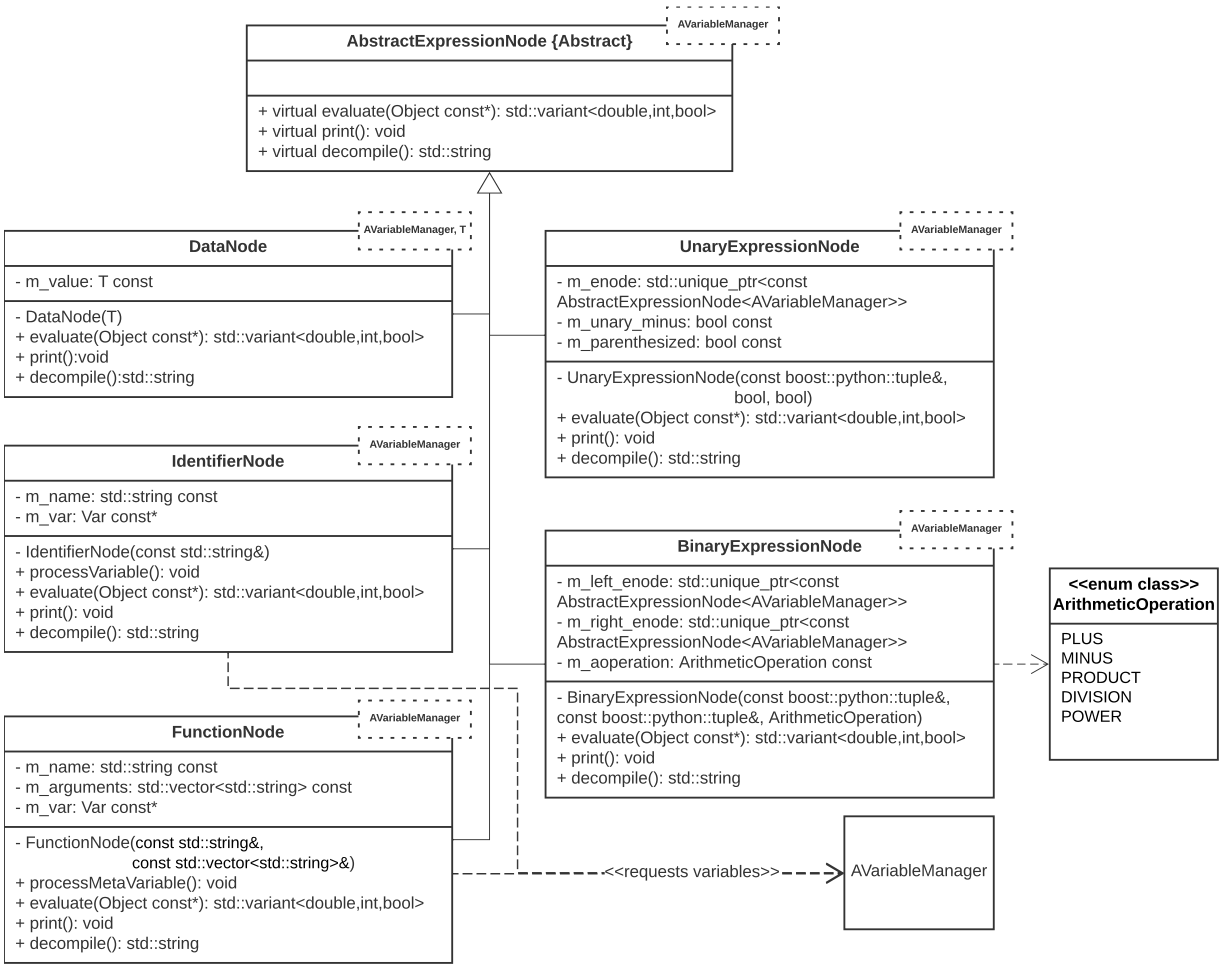
3. **IdentifierNode** is used to retrieve, store and evaluate normal variables.

4. **FunctionNode** is used to retrieve, store and evaluate metavariables/parametervariables.

5. Nodes for integer, double and boolean constants are needed. Instead of defining three separate classes, we define **DataNode**, which has an additional template parameter. This allows the compiler to automatically generate these three specializations. Their `evaluate` method returns the constant values.

4.5 NodeFactory class

The *NodeFactory* class provides a convenient way to convert the parser result into cut nodes. In this step, the `boost::python::tuple` must be processed iteratively and recursively to build the required tree structure. Figure 4.4 shows the process of parser invocation and processing of the syntax tree. The numeric constants, operators, brace



AbstractExpressionNode {Abstract}
AVariableManager
+ virtual evaluate(Object const*): std::variant<double,int,bool>
+ virtual print(): void
+ virtual decompile(): std::string
DataNode
AVariableManager, T
- m_value: T const
- DataNode(T)
+ evaluate(Object const*): std::variant<double,int,bool>
+ print():void
+ decompile():std::string
UnaryExpressionNode
AVariableManager
- m_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_unary_minus: bool const
- m_parenthesized: bool const
- UnaryExpressionNode(const boost::python::tuple&, bool, bool)
+ evaluate(Object const*): std::variant<double,int,bool>
+ print(): void
+ decompile(): std::string
IdentifierNode
AVariableManager
- m_name: std::string const
- m_var: Var const*
- IdentifierNode(const std::string&)
+ processVariable(): void
+ evaluate(Object const*): std::variant<double,int,bool>
+ print(): void
+ decompile(): std::string
BinaryExpressionNode
AVariableManager
- m_left_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_right_enode: std::unique_ptr<const AbstractExpressionNode<AVariableManager>>
- m_aoperation: ArithmeticOperation const
- BinaryExpressionNode(const boost::python::tuple&, const boost::python::tuple&, ArithmeticOperation)
+ evaluate(Object const*): std::variant<double,int,bool>
+ print(): void
+ decompile(): std::string
<<enum class>>
ArithmeticOperation
PLUS
MINUS
PRODUCT
DIVISION
POWER
FunctionNode
AVariableManager
- m_name: std::string const
- m_arguments: std::vector<std::string> const
- m_var: Var const*
- FunctionNode(const std::string&, const std::vector<std::string>&)
+ processMetaVariable(): void
+ evaluate(Object const*): std::variant<double,int,bool>
+ print(): void
+ decompile(): std::string
<<requests variables>>
AVariableManager

Figure 4.3: Node classes which are derived from *AbstractExpressionNode*. They override the `evaluate` method and perform arithmetic operations, variable lookups, and host constants.

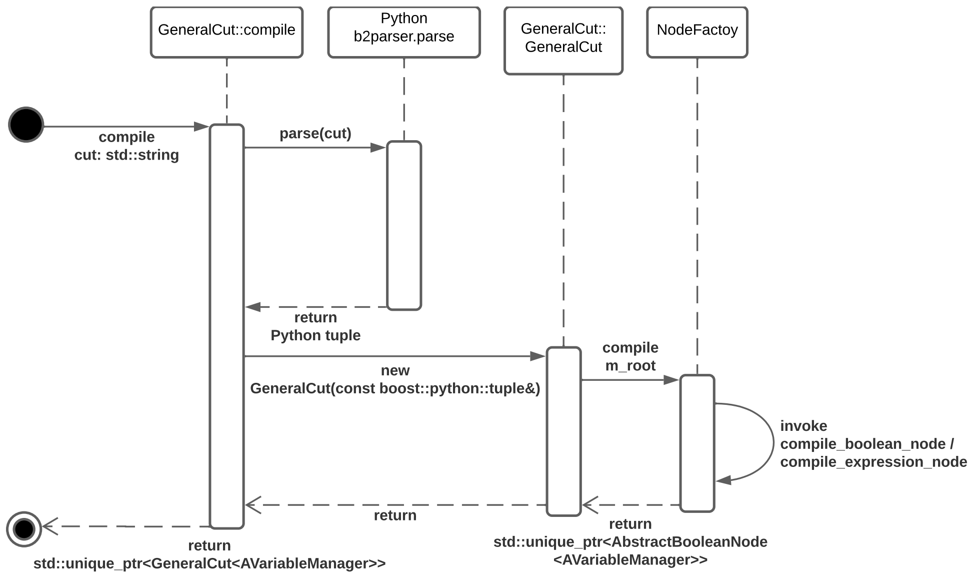

GeneralCut::compile
Python b2parser.parse
GeneralCut:: GeneralCut
NodeFactoy
compile cut: std::string
parse(cut)
return Python tuple
new GeneralCut(const boost::python::tuple&)
compile m_root
invoke compile_boolean_node / compile_expression_node
return
return std::unique_ptr<AbstractBooleanNode <AVariableManager>>
return std::unique_ptr<GeneralCut<AVariableManager>>

Figure 4.4: Sequence diagram of cut object generation. `b2parser.parse` is invoked by `compile` to generate the syntax tree. Then, the `m_root` member is compiled from the tuple using the *NodeFactory* functions.

information, and variable names must be extracted from the individual tuple entries and cast into C++ types. The *NodeFactory* class provides two templated static functions, `compile_boolean_node` and `compile_expression_node`, for creating boolean and expression nodes. Both accept a `boost::python::tuple` and return a `std::unique_ptr` of the created node.
Each node in the syntax tree is encoded as a Python tuple, where the first entry is guaranteed to be an integer and indicates the node type. Therefore this value can be extracted and decoded safely. Depending on the decoded node type, the number and types of arguments are known. The functions extract all arguments and construct the associated class. Since the cut node constructors are private to only allow creation via `GeneralCut::compile`, *NodeFactory* is defined as a friend class to all node classes in order to access their constructors.

Chapter 5

Summary and Conclusion

A domain-specific language for cuts was presented in this work. Backward compatibility, which could be achieved, was an essential criterion in the design. The grammar of the domain-specific language includes the existing syntax and extends the functionality by allowing arithmetic directly in the cut.
The *SLY* Python package is used to generate a parser directly from the grammar. The Python parser was integrated into the *GeneralCut* C++ utility class using the *boost python* library, replacing the manually written processing functions.
New node classes were introduced instead of packing the entire functionality into the *GeneralCut* class. These take over one responsibility each and are created from the Python parser output using the *NodeFactory*.
It is a sensible advancement, as only the grammar must be adapted for extensions of the cut syntax. The general architecture would not be affected. Adapting the individual processing functions is much more difficult.
If compatibility issues with the *SLY* package arise in the future, e.g., due to Python versions, the definitions of the domain-specific language can be used to generate the parser with any other *LALR(1)* parser generator framework.

Appendix

```python
from sly import Lexer
class B2Lexer(Lexer):
    """
    Class responsible for scanning the cut
    and generating a stream of tokens.
    """
    #: Declaration of all lexer tokens as a set
    tokens = {
        RBRACK, LBRACK, LPAREN, RPAREN, AND, OR, NOT,
        EQUALEQUAL, GREATEREQUAL, LESSEQUAL, GREATER, LESS,
        NOTEQUAL, DOUBLE, INTEGER, IDENTIFIER, BOOLEAN,
        POWER, TIMES, DIVIDE, PLUS, MINUS
    }
    # Scanning Rules
    ignore = " \t\n"  # ignore spaces tabs and newlines
    literals = {r","}  # literal definitions
    # Regular expressions as token definitions
    EQUALEQUAL = r"=="
    ...
    POWER = r"\*\*|\^"

    @_(r"(0(x|X)[0-9A-Fa-f]+)|\d+")
    def INTEGER(self, t):
        """Scanning function for integer values"""
        try:
            t.value = int(t.value)
        except ValueError:
            # cast hex value
            t.value = int(t.value, base=16)
        return t
```

Listing 1: Lexer class implementation using the *SLY* package. The regular expressions can be assigned to tokens either via class attributes or via function definitions named after the token. A decorator is used to bind the regular expression to the function.

```
from sly import Parser
...
class B2Parser(Parser):
    """
    Class responsible for producing the syntax tree
    from the token sequence.
    """
    # Token declaration
    tokens = B2Lexer.tokens
    # precedence definition
    precedence = (("left", "OR"), ("left", "AND"), ("nonassoc", "NOT"),
        ("left", "EQUALEQUAL", "GREATER", ...,  "NOTEQUAL"),
        ("left", "PLUS", "MINUS"),
        ("left", "TIMES", "DIVIDE"),
        ("right", "POWER"),
    )
    @_(r"", r"boolean_expression",)
    def cut(self, p):
        """
        Parsing function for <cut> nonterminal
        """
        try:
            # pass on the hitherto constructed syntax tree tuple
            return p.boolean_expression
        except AttributeError:
            # Return a new tuple
            return (
                self.get_node_type("UnaryRelationalNode"),
                (self.get_node_type("BooleanNode"),True)
            )
    ...
```

Listing 2: Parser class definition with the *SLY* package. The token set must be specified. Operator associativity and precedence are set via the `precedence` class attribute. The grammar rules are bound to the parsing functions with the @_ decorator. The parsing functions create a syntax tree as a nested Python tuple.

Bibliography

1. Earnshaw, R., Dill, J. & Kasik, D. *Data Science and Visual Computing* ISBN: 9783030243678 (Springer International Publishing, 2019).
2. *Super KEKB and Belle II* `https://www.belle2.org/project/super_kekb_and_belle_ii/`. Accessed: 26.02.2022.
3. Barrett, M. *et al.* The Belle II Online–Offline Data Operations System. *Computing and Software for Big Science* **5,** 1–12 (2021).
4. Moll, A. *The software framework of the Belle II experiment* in *Journal of Physics: Conference Series* **331** (2011), 032024.
5. *The Collaboration — Belle II Experiment* `https://belle2.jp/the-collaboration/`. Accessed: 03.03.2022.
6. Abe, T. *et al.* Belle II technical design report. *arXiv preprint arXiv:1011.0352* (2010).
7. Collaboration, B. I. *et al.* The Belle II physics book. *Progress of Theoretical and Experimental Physics* **2019,** ARTN–123C01 (2019).
8. Moll, A. *Comprehensive study of the background for the Pixel Vertex Detector at Belle II* PhD thesis (lmu, 2015).
9. Kuhr, T., Pulvermacher, C, Ritter, M, Hauth, T & Braun, N. The Belle II core software. *Computing and Software for Big Science* **3,** 1–12 (2019).
10. *Boost C++ libraries* `https://www.boost.org/`. Accessed: 26.02.2022.
11. Antcheva, I. *et al.* ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications* **180.** 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures, 2499–2512. ISSN: 0010-4655. `https://www.sciencedirect.com/science/article/pii/S0010465509002550` (2009).
12. Agostinelli, S. *et al.* Geant4—a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **506,** 250–303. ISSN: 0168-9002. `https://www.sciencedirect.com/science/article/pii/S0168900203013688` (2003).
13. Tamponi, U., Ritter, M., Hartbrich, O., Eliachevitch, M. & Cunliffe, S. *Fundamentals — basf2 release-06-00-03 documentation* `https://software.belle2.org/sphinx/release-06-00-03/online_book/fundamentals.html`. Accessed: 07.03.2022.

14. *pandas - Python Data Analysis Library* `https://pandas.pydata.org`. Accessed: 07.03.2022.
15. *NumPy* `https://numpy.org`. Accessed: 07.03.2022.
16. *SciPy* `https://scipy.org`. Accessed: 07.03.2022.
17. Schneeweiß, R. *Moderne C++ Programmierung: Klassen, Templates, Design Patterns* ISBN: 9783642214295 (Springer Berlin Heidelberg, 2012).
18. Aho, A., Lam, M., Sethi, R. & Ullman, J. *Compilers: Principles, Techniques, & Tools* ISBN: 9780321486813 (Pearson/Addison Wesley, 2007).
19. Wagenknecht, C. & Hielscher, M. *Formale Sprachen, abstrakte Automaten und Compiler: Lehr- und Arbeitsbuch für Grundstudium und Fortbildung* ISBN: 9783658026929 (Springer Fachmedien Wiesbaden, 2015).
20. Mogensen, T. *Introduction to Compiler Design* ISBN: 9783319669663 (Springer International Publishing, 2017).
21. Lesk, M. E. & Schmidt, E. *Lex: A lexical analyzer generator* (Bell Laboratories Murray Hill, NJ, 1975).
22. Beazley, D. *SLY (Sly Lex Yacc) — sly 0.0 documentation* en. `https://sly.readthedocs.io/en/latest/sly.html`. Accessed: 26.02.2022.

Danksagung

Ich bedanke mich herzlich bei Prof. Dr. Thomas Kuhr für die Betreuung und für die Übernahme des Referats.
Ein großes Dankeschön geht an die Arbeitsgruppe für die Unterstützung bei technischen Problemen und den hilfreichen Anmerkungen in den Gruppensitzungen.

Erklärung/Declaration

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt zu haben.

I hereby declare that this thesis is my own work, and that I have not used any sources and aids other than those stated in the thesis.

[illegible]

München, 08.03.2022
David Christian Katheder