CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering
Department of Cybernetics

BACHELOR'S THESIS

Julia Language Support for Kaitai Struct Binary Format Compiler

Dias Rystin

Supervisor: Ing. Michal Sojka, Ph.D.

Study program: Open Informatics

Specialisation: Artificial Intelligence and Computer Science

May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Rystin Dias**	Personal ID number:	**507288**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Julia Language Support for Kaitai Struct Binary Format Compiler

Bachelor’s thesis title in Czech:

Podpora jazyka Julia v kompilátoru binárních formátů Kaitai Struct

Guidelines:

1. Make yourself familiar with open source project Kaitai Struct and with the Julia langage.
2. Extend Kaitai Struct compiler to generate Julia modules from binary format descriptions in the .ksy format.
3. Extend Kaitai Struct testing infrastructure and write tests for validation of generated Julia code.
4. Setup up continuous integration system for the developed functionality. Ensure that the code is tested automatically with several supported Julia releases, including at least the latest Julia release or better a pre-release.
5. Document the results.

Bibliography / sources:

[1] https://doc.kaitai.io/
[2] https://docs.julialang.org/
[3] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. Modern Compiler Design. Springer, New York, NY, 2nd edition, 2000.

Name and workplace of bachelor’s thesis supervisor:

Ing. Michal Sojka, Ph.D. Embedded Systems CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **08.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Michal Sojka, Ph.D.	prof. Dr. Ing. Jan Kybic	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Citation of this thesis: Rystin Dias. *Julia Language Support for Kaitai Struct Binary Format Compiler*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2024.

Contents

Acknowledgments v

Declaration vi

Abstract vii

List of abbreviations viii

1 Introduction 1
- 1.1 Thesis structure . . . 2

2 Background 3
- 2.1 Binary formats . . . 3
- 2.2 Parsing binary data . . . 3
  - 2.2.1 Endianness . . . 4
  - 2.2.2 Data Types . . . 4
- 2.3 Julia Programming Language . . . 5
  - 2.3.1 Key Features and Strengths . . . 5
- 2.4 Kaitai Struct . . . 6
  - 2.4.1 KSY format gallery . . . 7
  - 2.4.2 Kaitai Struct Compiler . . . 8
  - 2.4.3 Runtime Libraries . . . 8
  - 2.4.4 Kaitai CI . . . 8
- 2.5 Similar works . . . 9
  - 2.5.1 EverParse . . . 10
  - 2.5.2 Spicy . . . 10
  - 2.5.3 BinData . . . 10
  - 2.5.4 FileIO.jl . . . 11

3 Methodology and design 12
- 3.1 Generated parsers design . . . 12
- 3.2 The Julia Runtime library . . . 14
- 3.3 Extending KSY compiler to support Julia . . . 15

4 Implementation 17
- 4.1 KSY primitive types . . . 17
- 4.2 Relational, bitwise and logical operators . . . 18

4.3 Arithmetic operators 18
4.4 Conversion methods 19
4.5 String Methods 20
4.6 Array methods 20
4.7 KSY features 20
4.7.1 Variable-length structures 21
4.7.2 Delimited structures 22
4.7.3 Substructures (subtypes) 23
4.7.4 Conditionals 26
4.7.5 Repetitions 27
4.7.5.1 Repeat for a specified number of times 27
4.7.5.2 Repeat until end of stream 28
4.7.5.3 Repeat until a condition is met 29
4.7.6 Switching types on an expression 29
4.7.7 Instances: data beyond the sequence 31
4.7.8 Enums (named integer constants) 32
4.7.9 Checking for “magic” signatures 33
4.7.10 Validating attribute values 35

5 Testing **37**
5.1 Unit testing in Julia 37
5.2 KST Translator 38
5.2.1 Testing the Julia Runtime library 39
5.3 Evaluation 40

6 Conclusion **41**

A Source code **42**

I would like to thank my supervisor Ing. Michal Sojka, Ph.D. and Kaitai Struct contributors for their assistance and support. I would also like to thank my family and friends for supporting me during my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague on May 20, 2024

Abstract

Kaitai Struct (KS) is a powerful tool designed for working with binary formats. It offers a declarative domain-specific language Kaitai Struct YAML (.ksy), enabling the description of complex structures within binary data. With KS, users can generate parsing modules in 11 target programming languages based on provided specifications. The goal of this thesis is to add Julia as a 12th target language to Kaitai Struct. This involves extending the Kaitai Struct Compiler, implementing the Julia runtime library, and integrating the solution into Kaitai Struct CI system. All these steps were successfully completed. This allows users of the increasingly popular Julia language to use KS in their projects.

Keywords Kaitai Struct, Julia, parsing, parser generator, binary format

Abstrakt

Kaitai Struct (KS) je mocný nástroj navržený pro práci s binárními formáty. Nabízí deklarativní doménově specifický jazyk Kaitai Struct YAML (.ksy), který umožňuje popis složitých struktur v rámci binárních dat. KS umožňuje uživatelům generovat moduly pro parsování v 11 cílových programovacích jazycích na základě poskytnutých specifikací. Cílem této práce je přidat Julii jako 12. cílový jazyk do Kaitai Struct. To zahrnuje rozšíření kompilátoru Kaitai Struct, implementaci runtime knihovny v Julii a integraci řešení do systému Kaitai Struct CI. Všechny tyto kroky byly úspěšně dokončeny. To umožňuje uživatelům stále populárnějšího jazyka Julia používat KS ve svých projektech.

Klíčová slova Kaitai Struct, Julia, parsování, generátor parserů, binární formát

List of abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CI	Continuous Integration
GPLv3	GNU General Public License, version 3
KS	Kaitai Struct
KSC	Kaitai Struct Compiler
KST	Kaitai Struct Test
KSY	Kaitai Struct YAML

Chapter 1
Introduction

Binary files are versatile and widely used across various applications and industries. They are used for storing executable programs, multimedia content like images and videos, structured data in databases, networking protocols, system configuration files, and data interchange between different software applications and systems. They play a fundamental role in computer systems, providing an efficient way of storing and processing data in a format that computers can directly understand and manipulate.

Binary formats define the structure and organization of data within binary files, specifying how different data types are encoded and stored.

Parsing is the process of extracting structured data from binary sources, and it is crucial for analyzing and interpreting digital information. However, several factors can make implementing parsers for binary formats challenging. Firstly, binary formats can vary widely in complexity, requiring parsers to handle different data types, encoding schemes, etc. Additionally, parsing binary data requires meticulous error handling to manage unexpected input and edge cases, further complicating the implementation. Furthermore, endianness and alignment add even more complexity to parser development. One approach to make a parser implementation easier is to utilize tools that generate parsers from high-level descriptions. These tools abstract away low-level implementation details, simplifying the parser development and reducing the risk of errors.

Kaitai Struct (KS) is a free and open-source project designed for working with binary formats. KS offers a declarative domain-specific language to describe the structure of binary data, allowing users to work with it in different programming languages. In Kaitai Struct, the format specification is separate from the choice of programming language. A specification can be automatically compiled into any of the 11 supported languages.

This thesis aims to improve KS by adding support for the Julia programming language. This means extending the KS compiler, implementing a Julia runtime library, and integrating the solution into Kaitai Struct CI system.

1.1 Thesis structure

This thesis is structured as follows:

Chapter 2 provides an overview of binary formats and binary data parsing, along with an introduction to the Julia programming language and the Kaitai Struct project. Additionally, it explores several tools that address similar challenges as Kaitai Struct.

Chapter 3 shows what KSC-generated parsing modules in Julia look like. It also presents the Julia runtime library in the process.

In Chapter 4 we analyze how features of the Kaitai Struct YAML (KSY) language can be translated and implemented in Julia.

Chapter 5 demonstrates the testing and briefly discusses the evaluation of the implemented solution.

Chapter 2
Background

This Chapter provides an overview of binary formats and binary data parsing, along with an introduction to the Julia programming language and the Kaitai Struct project. Additionally, it explores several tools that address similar challenges as Kaitai Struct.

2.1 Binary formats

A binary file is any file that contains at least some data that consists of sequences of bits that do not represent plain text [1]. Plain text formats use a byte of computer memory to interpret it as a character. A binary format is a more complex storage solution than a plain text format. Still, it will typically provide faster and more flexible access to the data and use up less memory.

A file with a binary format is simply a block of computer memory, just like a file with a plain text format. The difference lies in how the bytes of computer memory are used [2].

The characteristic feature of a binary format is that there is no simple rule for determining how many bits or how many bytes constitute a basic unit of information. Given a series of, say, four bytes, we cannot assume that these correspond to four characters, a single four-byte integer, or half of an eight-byte floating-point value. A description of the rules for the format that states what information is stored and how many bits or bytes are used for each piece of information is necessary to extract the information from a binary file. [3].

2.2 Parsing binary data

Binary parsing is the process of extracting structured data from binary files or streams. Unlike text-based formats such as JSON or XML, binary formats store data in a compact, binary representation that is not human-readable.

Therefore, parsing binary data requires an understanding of the underlying binary format and how to interpret it.

Parsing binary data involves several techniques depending on the complexity of the format:

- Fixed-Length Fields: Fields have fixed lengths, making parsing straightforward by reading a fixed number of bytes for each field.
- Variable-Length Fields: Some formats use markers or delimiters to indicate the start and end of variable-length fields, requiring more sophisticated parsing techniques to identify and extract these fields.
- Offset Pointers: Some formats may use pointers or offsets to refer to other parts of the binary data, requiring additional steps to resolve these references and navigate through the data.

2.2.1 Endianness

Endianness describes how multi-byte data is represented by a computer system and is dictated by the CPU architecture of the system. There are two common byte orders: big-endian and little-endian. In big-endian representation, the most significant byte comes first, while in little-endian representation, the least significant byte comes first [4].

The term comes from Swift's "Gulliver's Travels" via the famous paper "On Holy Wars and a Plea for Peace" by Danny Cohen [5].

2.2.2 Data Types

Binary formats define specific data types and their representations within the binary data. Types supported by most programming languages include integers (signed and unsigned), floating-point numbers, strings, and structures [6].

- Integers can be represented using various bit widths (e.g., 8-bit, 16-bit, 32-bit, 64-bit) and can be signed or unsigned.
- Floating-point numbers represent real numbers with fractional parts. They are typically encoded using IEEE 754 standard formats.
- Strings are sequences of characters [7] encoded using specific character encodings such as ASCII, UTF-8, or UTF-16.
- Structures define the layout of composite data types composed of multiple fields. Each field may have a different data type and size.

2.3 Julia Programming Language

Julia is a high-level, high-performance programming language specifically designed for technical and scientific computing. Developed to address the shortcomings of existing languages in the domain of numerical and data-intensive computing, Julia aims to provide a perfect balance between simplicity and speed.

The creators of Julia experienced in languages like Matlab, Lisp, Python, Ruby, and others, wanted to create a programming language that combines the strengths of various languages while minimizing their limitations. The result is an open-source language that combines the speed of C with the dynamism of Ruby, the macros of Lisp, and the usability of Python. It is mainly used in scientific computing, machine learning, data mining, and large-scale linear algebra, offering the power of C with the simplicity of other languages. Julia aims to be interactive yet compiled, providing the speed of C without the complexity [8].

2.3.1 Key Features and Strengths

Julia features optional typing, multiple dispatch, and good performance that is achieved using type inference and just-in-time (JIT) compilation (and optional ahead-of-time compilation), implemented using LLVM. It is a multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the lineage of mathematical programming languages but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby. The most significant departures of Julia from typical dynamic languages are [9]:

- The core language imposes very little; Julia Base and the standard library are written in Julia itself, including primitive operations like integer arithmetic.
- A rich language of types for constructing and describing objects, that can also optionally be used to make type declarations.
- The ability to define function behavior across many combinations of argument types via multiple dispatch.
- Automatic generation of efficient, specialized code for different argument types.
- Good performance, approaching that of statically compiled languages like C.

Although one sometimes speaks of dynamic languages as being typeless, they are not: every object, whether primitive or user-defined, has a type. The lack of type declarations in most dynamic languages, however, means that one cannot instruct the compiler about the types of values, and often cannot explicitly talk about types at all. In static languages, on the other hand, while one can and usually must annotate types for the compiler, types exist only at compile time and cannot be manipulated or expressed at run time. In Julia, types are themselves run-time objects, and can also be used to convey information to the compiler [9].

Apart from the breakout in runtime performances from traditional high-level dynamic languages, the fact that Julia was created from scratch means it uses the best, most modern technologies, without concerns over maintaining compatibility with existing code or internal architectures [10]. With substantial assistance from Julia developers, particularly those involved in the development of the JuMP (Julia for Mathematical Programming) package, Julia is an ideal tool for students and professionals engaged in operations research and its related domains, including industrial engineering, management science, transportation engineering, economics, and regional science [11].

2.4 Kaitai Struct

Kaitai Struct is a free and open-source project that has been developed since 2016. It provides a declarative domain-specific language Kaitai Struct YAML (KSY). Some of the provided features include a powerful expression language, primitive built-in data types, and the ability to define new types. KSY offers a structured approach to define data formats in a human-readable format, making it easier to understand and maintain complex data structures. Format specifications written in that language have `.ksy` extension. Kaitai Struct Compiler then can compile the KSY format specifications into parsing modules in 11 target languages. Those specifications also can be translated to GraphViz diagrams.

Let us see what the KSY language looks like. As an example, we can use a fixed-size structure. Below is an illustration of a database entry represented in the Julia programming language [12]:

```julia
struct AnimalRecord
    name::Vector{UInt8};        /* Name of the animal */
    birth_year::UInt16          /* Year of birth */
    weight::Float64;            /* Current weight in kg */
end
```

And here is what its `.ksy` specification would look like [12]:

```
meta:
  id: animal_record
  endian: be
seq:
  - id: name
    type: str
    size: 24
    encoding: UTF-8
  - id: birth_year
    type: u2
  - id: weight
    type: f8
```

The Kaitai Struct Language is a YAML-based language. Each `.ksy` file serves as a description of a type (format). Descriptions usually start with a `meta` section. Top-level info on the whole described structure is specified in this section. `seq` element with an ordered sequence of elements describes which attributes this structure consists of. Every attribute includes several keys, namely [12]:

- **id** is used to give the attribute a name
- **type** designates the attribute type:
  - no type means that data are represented as a raw byte array; **size** designates the number of bytes in the array
  - **s1**, **s2**, **s4**, **u1**, **u2**, **u4**, etc for integers
    - "s" means signed, "u" means unsigned
    - number is the number of bytes
    - non-default endianness can be used by appending **be** or **le** - i.e. **s4be**, **u8le**, etc
  - **f4** and **f8** for IEEE 754 floating point numbers; **4** and **8**, again, designate the number of bytes (single or double precision)
    - non-default endianness can be enforced by appending **be** or **le** - i.e. **f4be**, **f8le**, etc
  - **str** is used for strings; that is almost the same as "no type", but a string has a concept of encoding, which must be specified using **encoding**

2.4.1 KSY format gallery

Kaitai Struct provides a Format Gallery that shows a variety of binary file formats along with their respective Kaitai Struct specifications. Exploring the Format Gallery can help to understand the main features of the Kaitai Struct Language. The gallery includes formats from different fields such as multimedia, cryptography, compression, and networking.

2.4.2 Kaitai Struct Compiler

In its most general form, a compiler is a program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text. This process is called translation, as it would be if the texts were in natural languages [13].

The Kaitai Struct Compiler[1] translates `.ksy` specifications into parsers in 11 programming languages. It is written in Scala language. The compiler is released under the GPLv3 license.

2.4.3 Runtime Libraries

To generate a parsing module in a target language Kaitai Struct Compiler requires a utility language-specific runtime library. The library must follow the Kaitai Stream API. Generated modules use that library to parse data types provided by KSY, position in byte stream, and process byte arrays. Not only do runtime libraries make work with the byte streams easier but also they improve the readability of the generated modules. The Kaitai Stream API also standardizes some functions that may differ in different languages and provides processing operations to aid the conversion of byte arrays into their unpacked forms.

Runtime libraries provide throwable validation errors. These errors help validate files during the parsing process of the file formats that have security measures to ensure that parsed files are in the required format.

2.4.4 Kaitai CI

Ensuring the stability and reliability of Kaitai Struct across multiple languages, platforms, and compilers is a difficult task. With 11 programming languages and various platforms involved, manually testing everything becomes impractical. To address this, Kaitai Struct has set up an automated testing system that is integrated into its Continuous Integration (CI) setup.

The system puts together the testing process across 233 test KSY specifications. Each test consist of three parts: the KSY specification, a binary file, and the expected parsed values.

The testing process begins with the compilation of KSY specifications into parsers across all supported languages, using the latest version of the Kaitai Struct compiler. After that, the binary file associated with each test is parsed with the generated parser. The parsing output is then compared against the expected results.

Kaitai Struct uses the Kaitai Struct Test (KST) language to make the testing process consistent. This language defines the expected output for each test case. The KST translator uses these specifications to automatically generate

[1] `https://github.com/kaitai-io/kaitai_struct_compiler`

unit tests. These unit tests contain assertions that validate the equality of expected and actual parsed values.

Let us take a look at what KST specifications look like. Consider this KSY specification [14]:

```
meta:
  id: term_strz
  endian: le
seq:
  - id: s1
    type: str
    encoding: UTF-8
    terminator: 0x7c
  - id: s2
    type: str
    encoding: UTF-8
    terminator: 0x7c
    consume: false
  - id: s3
    type: str
    encoding: UTF-8
    terminator: 0x40
    include: true
```

Once a parser is generated based on this format, it can be tested with arbitrary binary data. However, we select a specific binary input and create a corresponding KST specification with test assertions for expected values. Here's what the corresponding KST specification might look like [14]:

```
id: term_strz
data: term_strz.bin
asserts:
  - actual: s1
    expected: '"foo"'
  - actual: s2
    expected: '"bar"'
  - actual: s3
    expected: '"|baz@"'
```

2.5 Similar works

There are lots of tools and frameworks for parsing binary data, each offering unique approaches and capabilities. This section briefly overviews several tools, that solve similar problems as Kaitai Struct does.

2.5.1 EverParse

EverParse is a framework for generating parsers and serializers from tag-length-value binary message format descriptions. The resulting code is verified to be safe (no overflow, no use after free), correct (parsing is the inverse of serialization), and non-malleable (each message has a unique binary representation). These guarantees underpin the security of cryptographic message authentication and enable testing to focus on interoperability and performance issues [15].

EverParse consists of two parts: LowParse, a library of parser combinators and their formal properties written in F*; and QuackyDucky, a compiler from a domain-specific language of RFC message formats down to low-level F* code that calls LowParse. While LowParse is fully verified, we do not formalize the semantics of the input language and keep QuackyDucky outside our trusted computing base. Instead, it also outputs a formal message specification, and F* automatically verifies our implementation against this specification. EverParse yields efficient zero-copy implementations, usable both in F* and in C [15].

2.5.2 Spicy

Spicy is a parser generator that makes it easy to create robust C++ parsers for network protocols, file formats, and more [16]. Spicy offers a domain-specific scripting language to define the syntax and semantics of input formats.

The Spicy toolchain turns such grammars into efficient C++ parsing code that exposes an API to host applications for instantiating parsers, feeding them input, and retrieving their results. At runtime, parsing proceeds fully incrementally—and potentially highly concurrently—on input streams of arbitrary size. Compilation of Spicy parsers takes place either just-in-time at startup (through a C++ compiler), or ahead-of-time either by creating pre-compiled shared libraries or by giving you generated C++ code that you can link into your application [16].

2.5.3 BinData

BinData provides a declarative way to read and write structured binary data. The programmer specifies the format of the binary data and BinData works out how to read and write data in this format. It is an alternative to Ruby's `#pack` and `#unpack` methods [17].

The BinData documentation provides the following comparison of parsing data using Ruby's built-in functionality and using the library. Here is what the first option may look like:

```
io = File.open(...)
len = io.read(2).unpack("v")[0]
name = io.read(len)
width, height = io.read(8).unpack("VV")
puts "Rectangle #{name} is #{width} x #{height}"
```

Here is what parsing the same structure using BinData looks like:

```
class Rectangle < BinData::Record
  endian :little
  uint16 :len
  string :name, read_length: :len
  uint32 :width
  uint32 :height
end
io = File.open(...)
r = Rectangle.read(io)
puts "Rectangle #{r.name} is #{r.width} x #{r.height}"
```

It supports all the common datatypes that are found in structured binary data. Support for dependent and variable length fields is built in [17].

2.5.4 FileIO.jl

FileIO aims to provide a common framework for detecting file formats and dispatching to appropriate readers/writers. The two core functions in this package are called `load` and `save`, and offer high-level support for formatted files (in contrast with julia's low-level `read` and `write`) [18].

If a format is supported by FileIO, the `load` function can be used to read data from a formatted file. FileIO will attempt to find an installed package capable of reading filename; if no such package is found, it will suggest an appropriate package for the user to add [18].

Chapter 3

Methodology and design

Parsers are pervasive software basic blocks: as soon as a program needs to communicate with another program or to read a file, a parser is involved. However, writing robust parsers can be difficult, as is revealed by the amount of bugs and vulnerabilities related to programming errors in parsers. One way to solve the problem is to rely on tools to generate the actual parsers from high-level descriptions [19]. This chapter shows how one can use one of such tools the Kaitai Struct to generate parsing modules in the Julia programming language.

3.1 Generated parsers design

Let us consider the following example of a binary format specification in KSY:

```yaml
meta:
  id: example
  bit-endian: be
seq:
  - id: a
    type: b1
  - id: b
    type: b32
  - id: c
    type: b7
```

This specification describes a simple format named “example”, consisting of three fields of different sizes. The `meta` section provides additional information for the “example” format. In some cases, the `meta` section might be used in intermediate types as well, for example, to switch default endianness or encoding [20]. The fields inside `seq` are parsed sequentially, one after another.

The `id` and `type` specify the name and type of the field. The identifier `b` represents bits, with the number indicating the number of bits.

We can define a `mutable struct` and override its constructor to create a parser for the format above in Julia. Fields inside **`seq`** can naturally serve as attributes of the `struct`. Since these fields are parsed sequentially, we can populate them during parsing using Julia's incomplete initialization. The parsing and assigning can be extracted to a function to make the code more readable. Each parser generated by the Kaitai Struct Compiler must provide a `from_file` method for ease of use. A user can call the `from_file` function with the path to the binary file to parse it. The generated parser looks like this:

```julia
mutable struct Example
  a::Bool
  b::UInt32
  c::UInt8
  _io::KaitaiStruct.KaitaiStream
  _root::Union{Example, Nothing}
  _parent::Any
  function Example(_io, _parent = nothing, _root = nothing)
    this = new()
    this._io = _io
    this._parent = _parent
    this._root = _root === nothing ? this : _root
    _read(this)
    this
  end
end

function from_file(filename::String)::Example
  Example(KaitaiStruct.KaitaiStream(open(filename, "r")))
end

function _read(this::Example)
  this.a = KaitaiStruct.read_bits_int_be(this._io, 1) != 0
  this.b = KaitaiStruct.read_bits_int_be(this._io, 32)
  this.c = KaitaiStruct.read_bits_int_be(this._io, 7)
  nothing
end
```

Each generated `struct` includes special pseudo-attributes: `_io`, `_parent`, and `_root`. The `_parent` attribute can be used to access the parent structure in each generated type. In some cases, writing several `_parent` attributes in a row may be impractical or impossible, especially if describing a type that might be used on several different levels, requiring a different number of `_parent` attributes. In such cases, the special pseudo-attribute `_root` can be used to navigate from the top-level type.

Notice how the generated parser follows the corresponding `.ksy` specification. This alignment is possible because the parsing of common data structures is extracted to a dedicated utility library called the runtime library, which provides methods to read commonly used data types from a byte stream.

3.2 The Julia Runtime library

The Julia Runtime library is a Julia package that follows the Kaitai Stream API. It is published as a Julia package under KaitaiStruct.jl name, making it accessible to the broader Julia community. The package is published under KaitaiStruct.jl name. Kaitai Stream API includes the following methods, that can be classified into the following categories:

- stream positioning functions include: checking for the end of the stream `iseof`, moving the stream position `seek`, retrieving the current position `pos`, and determining the size of the stream `size`.
- Integer number operations include reading signed `read_s1` and unsigned `read_u1` values. Both big-endian and little-endian formats are supported for signed (e.g., `read_s2be`, `read_s4le`) and unsigned (e.g., `read_u2be`, `read_u4le`) integers.
- Floating-point number operations provide reading for both big-endian and little-endian (`read_f4be`, `read_f8be`, `read_f4le`, `read_f8le`) formats.
- Unaligned bit values can be processed using functions like `align_to_byte`, `read_bits_int`, and `read_bits_array`.
- Byte arrays can be read with functions `read_bytes`, `read_bytes_full`, and `read_bytes_term`. Those methods can be used to work with variable-length structures or delimited structures.
- Byte array processing methods include XOR operations `process_xor`, rotation `process_rotate_left`, and zlib compression `process_zlib`.
- Miscellaneous runtime operations include a static modulo function static `mod`. It is important to ensure that such operations work in the expected way with for example negative numbers.

These methods operate on the `KaitaiStream` structure defined within the module:

```
mutable struct KaitaiStream
  io::IO
  bits_left
  bits
  KaitaiStream(io) = new(io, 0, 0)
end
```

The code above declares a `KaitaiStream` structure that has `io`, `bits_left`, and `bits` attributes. The attributes `bits_left`, and `bits` allow to work with bits. Julia built-in functions can read only bytes from byte streams, so to overcome that `KaitaiStruct.read_bits` reads bytes from the stream and saves unused bits in the `KaitaiStream.bits` field. It is important to note that while Julia's built-in `Base.eof` function will return true when there are unread bits left, `KaitaiStruct.iseof` should be used when working with bits to ensure correct behavior.

The Kaitai API also defines several error types that can be thrown during parsing. Let's take a closer look at each of them:

- `ValidationFailedError`: This serves as an abstract type for validation errors. It encompasses the following specific validation errors. All of them help to safely handle scenarios when KSY's **valid** key is used:
  - `ValidationNotEqualError`: Indicates that a value is not equal to the expected value.
  - `ValidationLessThanError`: Occurs when a value is less than the expected value.
  - `ValidationGreaterThanError`: Occurs when a value is greater than the expected value.
  - `ValidationNotAnyOfError`: Indicates that a value does not match any of the expected values.
  - `ValidationExprError`: Indicates that a value does not match the expected expression.
- `UndecidedEndiannessError`: This error occurs when the endianness of the data cannot be determined.

3.3 Extending KSY compiler to support Julia

KS compiler is written in Scala language. Adding support for Julia language to KS compiler consists of implementing a language-specific compiler that can generate code in Julia. Typically a language-specific compiler consists of two basic parts. In our case they are called `JuliaCompiler` and `JuliaTranslator`. But we also need the third part called `JuliaClassCompiler`.

The `JuliaCompiler` maps KS concepts to corresponding concepts in Julia and generates the parsing modules in Julia. For example, it defines how KS's repeated parsing of a field is mapped to a for cycle in Julia. `JuliaCompiler` uses `JuliaTranslator` to translate basic data types, operators, and expressions from KSY to Julia syntax. For instance, an integer in KS would be translated to an integer in Julia by the `JuliaTranslator`. `JuliaClassCompiler` focuses on complex types. It translates KS types into Julia structs, ensuring that data structures are appropriately represented in the generated Julia code.

The next Chapter provides the detailed information about how KS concepts are mapped to Julia code.

Chapter 4

Implementation

Kaitai Struct offers a declarative domain-specific language Kaitai Struct YAML (KSY). This chapter shows how KSY's primitive types, methods, and its other concepts are translated to Julia.

4.1 KSY primitive types

KSY byte arrays are defined by omitting the type attribute.The size of a byte array is thus determined using `size`, `size-eos`, or `terminator` fields, one of which is mandatory in this case [12].

```
seq:
- id: byte_array
  size: 16
```

In Julia, they can be represented by using `Base.Vector{UInt8}` type. Fixed-sized byte arrays can be represented using a byte-array string literal: `b"..."`. KSY booleans can be specified as boolean literal `true` and `false` values or can be derived by using `type: b1`. This type specifies that a single bit from a stream is parsed and represented as a boolean value: 0 becomes false, and 1 becomes true [12]. However, assigning 0 or 1 directly to a field with type `Base.Bool` in Julia is not allowed. To handle this, the parsed value can be compared to 0, as shown in the following code snippet:

```
KaitaiStruct.read_bits_int_be(io, 1) != 0
```

It is important to be aware of incomplete initialization in Julia. To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized [21]. In Julia, an uninitialized `Base.Bool` defaults to `false`, not `nothing`. To simulate three-valued logic `Union{Bool, Nothing}` type can be used.

Three-valued logic may be needed for example when endianness was additionally specified in the KSY meta section. Consider the following example:

```yaml
meta:
  endian:
    switch-on: indicator
    cases:
      '[0x49, 0x49]': le
      '[0x4d, 0x4d]': be
```

Kaitai Struct handles such cases by adding `_is_le` attribute to the type [22]. In case `indicator` matches neither of the cases, the special undecided endianness error must be thrown. However, if `Base.Bool` is used for `_is_le`, it will be initialized to `false`, potentially leading to parsing data with the wrong endianness. Adding the third `nothing` option helps avoid this issue.

In KSY, strings are typically specified using the `type: str` or `type: strz` which implicitly adds **`terminator: 0`**. Literal strings can be specified using double quotes or single quotes. Single-quoted strings are interpreted literally, meaning backslashes `\`, double quotes `"`, and other special symbols carry no special meaning; they are treated as part of the string [12]. In Julia strings are denoted by double quotation marks `"text"` [23]. Single quotation marks are used for characters in Julia `'c'`. Certain characters, such as $, must be escaped during the translation. This is necessary because Julia allows interpolation into string literals using $.

4.2 Relational, bitwise and logical operators

Literals in KSY can be combined using operators to create meaningful expressions. Operators vary depending on the types involved: for instance, the + operator applied to two integers represents arithmetic addition, while the same operator applied to two strings signifies string concatenation [12]. Relational operators can be directly translated to Julia without any alterations. In KSY, bitwise XOR is denoted by `^`, whereas in Julia the same operator is presented by ⊻ [24]. KSY logical operators require the following translation `not` becomes `!`, `and` becomes `&&`, and `or` becomes `||`.

4.3 Arithmetic operators

KSY arithmetic operators are almost identical to the ones in Julia. However, some adjustments must be made during translation. In KSY, if both operands are integers, the result of a division operation is an integer [12]; otherwise, it is a floating-point number. For example, `7 / 2` equals `3` in KSY, whereas `7 / 2.0` equals `3.5`. In Julia, `7 / 2` produces `3.5`, so for integer division, KSC uses the `Base.fld` function instead of `/`. It's worth noting that the `//` operator in Julia yields the special type `Rational`, and the comparison `7//2 == 3.5` evaluates to `true` [24].

In KSY, `%` denotes modulo, whereas in Julia, this symbol represents the remainder operator [24]. The Kaitai Stream API standardizes the computation of modulo, and the Julia Runtime library provides its implementation [25].

The addition operator can be used with two strings in KSY, resulting in their concatenation [12]. Julia, on the other hand, uses the multiplication operator for string concatenation. It's important to be aware of potentially dangerous situations such as the concatenation of invalid UTF-8 strings. The resulting string may contain different characters than the input strings, and its number of characters may be lower than the sum of the number of characters of the concatenated strings [23].

4.4 Conversion methods

The KSY expression language provides several built-in conversion methods for various data types.

Floating-point numbers in KSY can be converted to integers using the `to_i` method, which truncates the floating-point value [12]. In Julia, the same operation is performed using the `Base.trunc()` function.

KSY provides the `to_s(encoding)` method to convert byte arrays into strings [12]. The similar functionality is available in the `StringEncodings`[1] package in Julia. The package offers support for decoding and encoding texts across various character encodings.

Converting a string to an integer can be done in KSY by the `to_i` method, with an optional `radix` argument to specify the base [12]. Julia has a built-in `Base.parse()` function that serves the same purpose. It allows base specification as well.

For booleans and enums, KSY provides the `to_i` method to convert them to their corresponding integer representations [12]. In Julia, this functionality can be achieved using the `Base.Int()` function [9].

For integers, the `to_s` method in KSY allows conversion to strings [12]. In Julia, the same result can be achieved by using the `Base.string()` function. KSY allows using prefixes indicating base we have to be careful to translate the literal with the wanted base. It is possible to use _ as a visual separator in KSY integer literals. Julia's built-in `Base.string()` has the functionality to choose the output base and handles _ as well [23].

[1] `https://github.com/JuliaStrings/StringEncodings.jl`

4.5 String Methods

KSY offers several methods for string manipulation. The `length` method returns the number of characters in a string, while `reverse` provides the reversed version of the string. Additionally, the `substring(from, to)` method extracts a portion of the string between the characters at the specified offsets (`from` and `to - 1`), inclusive of `from` and exclusive of `to` [12].

To translate these methods to Julia, we can use `Base.length()` for obtaining the string length, `Base.reverse()` for reversing the string, and range indexing for substring extraction [23]. It's important to note that Julia's indexing starts from one, so when using range indexing, we need to adjust the `from` value by incrementing it by one. However, the original `to` value can be directly used in range indexing as it is inclusive in the indexing scheme of Julia [9].

4.6 Array methods

KSY provides the following methods to work with arrays [12]:

- **first**: Retrieves the first element of an array.
- **last**: Retrieves the last element of an array.
- **size**: Returns the number of elements in an array.
- **min**: Retrieves the minimum element of an array.
- **max**: Retrieves the maximum element of an array.

In Julia, the keywords `begin` and `end` can be used in indexing to access the first and last elements of an array respectively. For obtaining the minimum and maximum elements of an array, Julia provides the functions `Base.minimum()` and `Base.maximum()`. To determine the number of elements in an array, `Base.size()` can be used [9].

It's important to note that in KSY, the term "number of elements in an array" for a 2-dimensional array refers to the number of "rows". In Julia, however, `Base.length()` returns the total number of elements across all dimensions of an array, while `Base.size()` returns a tuple containing the number of elements for each dimension. To replicate the behavior of KSY's **size** method, we can call Julia's `Base.size(_, 1)`. The second argument 1 means that size over the first dimension is calculated.

4.7 KSY features

Let us explore specific features of the KSY language. Notice that code snippets in Julia do not always show the exact way of what generated parsers look like. They are meant to show how the corresponding feature in KSY can be implemented in Julia.

4.7.1 Variable-length structures

Variable-length structures are common in many protocols and file formats, especially for strings where conserving bytes is crucial. For instance, using a fixed buffer size of 512 bytes for a string that typically ranges from 3 to 5 bytes in length would be inefficient. To address this, Kaitai Struct offers support for variable-length structures [12].

In Kaitai Struct, handling variable-length structures is straightforward. For example, consider parsing a string preceded by an integer that designates its length:

```yaml
seq:
  - id: my_len
    type: u4
  - id: my_str
    type: str
    size: my_len
    encoding: UTF-8
```

To achieve this functionality in Julia we can just read `my_len` and pass it as an argument to `KaitaiStruct.read_bytes` to specify the number of bytes to parse. After that, we can decode the read bytes using the specified encoding:

```julia
my_len = KaitaiStruct.readU4(io)
my_str = decode(KaitaiStruct.read_bytes(io, my_len), "UTF-8")
```

Kaitai Struct also allows specifying a size that spans automatically to the end of the stream. This can be achieved using a slightly different syntax [12]:

```yaml
seq:
  - id: string_spanning_to_the_end_of_file
    type: str
    encoding: UTF-8
    size-eos: true
```

In this case, we can not use `KaitaiStruct.read_bytes` as the number of needed bytes is unknown. Instead, the Julia Runtime library provides a function to read bytes until the end of a stream `KaitaiStruct.read_bytes_full`:

```julia
string_spanning_to_the_end_of_file
    = decode(KaitaiStruct.read_bytes_full(io), "UTF-8")
```

4.7.2 Delimited structures

Delimited structures offer a flexible way to handle data without requiring fixed-size buffers, and Kaitai Struct provides support for defining and parsing such structures. Let's explore this concept further with examples and options available in KSY.

Consider a common scenario of parsing a null-terminated string [12]:

```yaml
seq:
  - id: my_string
    type: str
    terminator: 0
    encoding: UTF-8
```

In this example, the string is terminated by the null byte (0). By default, the terminator is consumed and not included in the read data. However, KSY provides options to customize this behavior [12]:

- `consume: false`: Specifies not to consume the terminator.
- `include: true`: Includes the terminator in the read data.

KSY also has the option to silence an end-of-stream error and use parsed data if no terminator is met `eos-error: false`. It is set to `true` by default.

We can use `KaitaiStruct.read_bytes_term` from the Julia Runtime library to read such structures in Julia. This function reads bytes from the stream until the terminator byte is encountered or the end of the stream is reached. It offers flexibility through arguments such as including or consuming the terminator byte and handling end-of-stream errors. The arguments are:

- `stream::KaitaiStream`: The input stream from which bytes will be read.
- `term::UInt8`: The terminator byte.
- `include_term::Bool`: A flag indicating whether to include the terminator byte in the output.
- `consume_term::Bool`: A flag indicating whether to consume the terminator byte.
- `eos_error::Bool`: A flag indicating whether to raise an error if the end of the stream is reached before encountering the terminator byte.

The null-terminated string can be parsed in the following way:

```julia
bytes = KaitaiStruct
  .read_bytes_term(io, 0x00, false, true, true)
my_string = decode(bytes, "UTF-8")
```

Reading “until the terminator byte is encountered” could be dangerous. What if we never encounter that byte? A common way to avoid that danger is to have both a fixed-sized buffer and a terminator [12].

It‘s possible to model that kind of behavior in Kaitai Struct just by combining size and terminator [12]:

```
seq:
  - id: name
    type: str
    size: 16
    terminator: 0
    encoding: UTF-8
```

This works in 2 steps:

- `size`: Ensures that exactly 16 bytes are read from the stream.
- `terminator`: Given that `size` is present, only works inside these 16 bytes, cutting the string short early with the first terminator byte encountered, saving the application from getting all that trailing garbage.

This functionality can be reached by using `KaitaiStruct.read_bytes` to read a specified number of bytes and calling `KaitaiStruct.bytes_terminate` to trim out unnecessary data. The function allows choosing whether to include the terminator byte.

```
raw_bytes = KaitaiStruct.read_bytes(io, 16)
cut_bytes = KaitaiStruct
  .bytes_terminate(raw_bytes, 0x00, false)
name = decode(cut_bytes, "UTF-8")
```

4.7.3 Substructures (subtypes)

In KSY, you can define additional types within the same `.ksy` file, making it easier to manage repetitive data structures. Let’s take a closer look at how this works and how you we can achieve similar functionality in Julia.

```yaml
seq:
  - id: track_title
    type: str_with_len
  - id: album_title
    type: str_with_len
  - id: artist_name
    type: str_with_len
types:
  str_with_len:
    seq:
      - id: len
        type: u4
      - id: value
        type: str
        encoding: UTF-8
        size: len
```

Here, a type named `str_with_len` is defined, allowing for its reuse in the `track_title`, `album_title`, and `artist_name` fields. The types section encapsulates the definition of `str_with_len`, where its structure is specified using the **seq** designation.

Notice that there is no need for `meta:/id:` in the `types:` section, as the type name is derived from the type key name here [12].

To achieve the same functionality in Julia, we can create a `struct` and customize its constructor to parse the necessary data from the stream. Here's an example implementation

```julia
mutable struct StrWithLen
  len::UInt32
  value::String
  ...
  function StrWithLen(...)
    this = new()
    this.len = KaitaiStruct.readU1(this._io)
    this.value = decode(
      KaitaiStruct.read_bytes(this._io, this.len), "UTF-8")
    this
  end
end
```

With this setup, parsing becomes straightforward:

```julia
track_title = StrWithLen(...)
album_title = StrWithLen(...)
artist_name = StrWithLen(...)
```

The format description may contain several `types:` fields at different levels. A type name must be unique only within the scope of the `types:` field in which it is declared. This may lead to naming conflicts for languages that do not support classes. Consider this example [12]:

```
seq:
  - id: main_data
    type: main
  - id: dummy
    type: dummy_obj
types:
  main:
    seq:
      - id: foo
        type: foo_obj
    types:
      foo_obj:
        seq:
          ...
  dummy_obj:
    seq:
      - id: foo
        type: foo_obj
    types:
      foo_obj:
        seq:
          ...
```

Two types with the same name, `foo_obj`, are declared at different levels. Class-based languages can implement a parser for such formats using inner classes, as an inner class is associated with an instance of its enclosing class [26]. Julia, however, is not a class-based language, so in my implementation, the name of a type includes the name of the parent type as a prefix:

```
mutable struct DummyObj
  ...
mutable struct DummyObj_FooObj
  ...
```

KSY's expression language also allows users to refer to attributes of other types. Here's an example of its usage [12]:

```
seq:
  - id: header
    type: main_header
  - id: body
    size: header.body_len
types:
  main_header:
    seq:
      - id: body_len
        type: u4
```

If the `body_len` attribute were in the same type as the body, we could simply use `size: body_len`. However, in this case, we have decided to split the main header into a separate subtype, so we must access it using the dot operator — i.e., `size: header.body_len`. In Julia, we can access attributes of other objects using the dot operator, similar to how it's done in KSY.

4.7.4 Conditionals

Certain fields may be optional and exist only under specific conditions in some protocols and file formats. For instance, a byte may designate whether another field exists `1` or not `0`. In Kaitai Struct, we can handle such scenarios using the `if` key [12]:

```
seq:
  - id: has_crc32
    type: u1
  - id: crc32
    type: u4
    if: has_crc32 != 0
```

In this example, a boolean expression is specified in the `if` key using expression language. If the expression evaluates to true, the field is parsed and the result is assigned. If the expression evaluates to false, the field is skipped, and accessing it will return `nothing` (or its closest equivalent in our target programming language).

In Julia, this can be done the following way:

```
has_crc32 = KaitaiStruct.readU1(io)
if has_crc32 != 0
  crc32 = KaitaiStruct.readU4be(io)
end
```

Note that by default `has_crc32` can be initialized with a random value [21]. This is because `UInt32` is a plain data type and the initial contents of plain data types are undefined. To avoid undefined behavior the type of `has_crc32` is a union of `UInt32` and `Nothing`.

Julia also supports short-circuit evaluation behavior. This behavior is frequently used in Julia to form an alternative to very short if statements [10]. Instead of if `<cond> <statement> end`, one can write `<cond> && <statement>` which could be read as: `<cond> and then <statement>`. Similarly, instead of if `!<cond> <statement> end`, one can write `<cond> || <statement>`. So the code above can be rewritten as:

```
has_crc32 = KaitaiStruct.readU1(io)
has_crc32 != 0 && crc32 = KaitaiStruct.readU4be(io)
```

But for consistency with other languages supported by Kaitai Struct, I have decided not to use short circuit evaluation behavior.

4.7.5 Repetitions

Many file formats consist of repeated patterns rather than single elements. These repetitions can take different forms:

- Elements repeated until the end of the stream
- Elements repeated while a certain condition is not satisfied (or until a condition becomes true)
- Elements repeated a predefined number of times

Kaitai Struct supports all these types of repetitions. In each case, it creates a vector or its nearest equivalent available in the target language and populates it with elements [12].

In Julia, there are two main constructs for repeated evaluation of expressions: the `while` loop and the `for` loop. The `while` loop is suitable for the first two types of repetition because the number of repetitions is unknown. The `for` loop is appropriate for the third type since we know the number of repetitions [9].

In the next three subsections, we'll explore each type of repetition in more detail.

4.7.5.1 Repeat for a specified number of times

Sometimes, an element needs to be repeated a certain number of times. This can be achieved by referencing an attribute to determine the array's length [12]:

```yaml
seq:
  - id: num_floats
    type: u4
  - id: floats
    type: f8
    repeat: expr
    repeat-expr: num_floats
```

In Julia, this structure can be parsed like so:

```julia
num_floats = KaitaiStruct.readU4le(_io)
floats = Vector{Float64}()
for i in 1:num_floats
    push!(floats, KaitaiStruct.readF8(_io))
end
```

The code above creates a vector of doubles and populates it with parsed data. Preallocation may seem like a good idea since we know the number of floats in advance, which can potentially speed up the code execution. However, in reality, it may not always be the best approach. This is because if there's a parsing error or data corruption, a large amount of extra memory may be allocated unnecessarily. This can lead to memory wastage and performance issues. Therefore, it's often better to dynamically allocate memory as needed, especially in scenarios where the integrity of input data is uncertain. Notice that expression language can be used to determine the number of repetitions.

4.7.5.2 Repeat until end of stream

This is the simplest type of repetition, done by specifying `repeat: eos`. For example:

```yaml
seq:
  - id: numbers
    type: u4
    repeat: eos
```

This yields an array of unsigned integers, each 4 bytes long, which spans till the end of the stream. Notice that if we have some amount of bytes left in the stream that‘s not divisible by 4, we‘ll end up reading as much as possible, and then the parsing procedure will throw an end-of-stream exception [12].

A generated parser of such format in the Julia language could look like this:

```
numbers = Vector{UInt32}()
while !KaitaiStruct.iseof(_io)
    push!(numbers, KaitaiStruct.readU4le(_io))
end
```

4.7.5.3 Repeat until a condition is met

In some file formats, the number of elements in an array is not specified. Instead, a special element acts as a terminator to signify the end of the data. Kaitai Struct supports this using the repeat-until syntax. For example:

```
seq:
  - id: numbers
    type: s4
    repeat: until
    repeat-until: _ == -1
```

The description above reads 4-byte signed integer numbers until encountering -1. On encountering -1, the loop will stop and further sequence elements (if any) will be processed. Note that -1 would still be added to the array [12].

The underscore (_) is used as a special variable name that refers to the element that was just parsed. In Julia, this can be implemented as follows:

```
numbers = Vector{Float32}()
while true
    _it = KaitaiStruct.readS4le(_io)
    push!(numbers, _it)
    if _it == -1
        break
    end
end
```

This code continuously reads 4-byte signed integers from the stream until encountering -1. Once -1 is encountered, the loop terminates, and further processing continues.

4.7.6 Switching types on an expression

In Kaitai Struct, you can utilize a switch-type operation to handle different types based on an expression's value. This is particularly useful when parsing formats where the type of data varies depending on a specific code or indicator. Here's an example [12]:

```
seq:
  - id: code
    type: u1
  - id: body
    type:
      switch-on: code
      cases:
        1: u1
        2: u2
        4: u4
        8: u8
```

Notice that `size` is specified on the attribute level, thus it applies to all possible type values, setting us a good hard limit. If a type description is missing the match, as long as it has `size` specified, `body` would still be parsed with the given size, but instead of interpreting it as some user type, it would be treated as having no `type`, thus yielding a raw byte array. This allows a user to work on TLV-like formats step-by-step, starting by supporting only 1 or 2 types of records, and gradually adding more and more types [12].

The generated parser in Julia uses an `if - elseif - end` structure to achieve the same functionality:

```
code = KaitaiStruct.readU1(io)
_on = code
if _on == 1
    body = KaitaiStruct.readU1(io)
elseif _on == 2
    body = KaitaiStruct.readU2le(io)
elseif _on == 4
    body = KaitaiStruct.readU4le(io)
elseif _on == 8
    body = KaitaiStruct.readU8le(io)
end
```

It's important to ensure that the type used in `switch-on` and the types used in `cases` are either identical or at least comparable [12].

Additionally, KSY provides a special keyword "_" for the default (else) case which will match every value that was not listed explicitly. For example:

```
type:
  switch-on: code
  cases:
    1: u1
    2: u2
    _: u4
```

In Julia, the same functionality can be achieved by using `else` for the `_` case:

```
code = KaitaiStruct.readU1(io)
_on = code
if _on == 1
    body = KaitaiStruct.readU1(io)
elseif _on == 2
    body = KaitaiStruct.readU2be(io)
else
    body = KaitaiStruct.readU4be(io)
end
```

This setup ensures that if the `code` value does not match any of the specified cases, `body` will be parsed as `u4`.

4.7.7 Instances: data beyond the sequence

Up to this point, all data specifications were defined within a `seq`, meaning they would be parsed immediately from the beginning of the stream, one by one, in strict sequence. But what if the desired data is located elsewhere in the file, or arrives out of sequence [12]?

Kaitai Struct's "Instances" feature provides support for such scenarios. They are specified within a `instances` block at the same level as `seq`. Consider the following example:

```
meta:
  id: instance
  endian: le
instances:
  header:
    pos: 2
    type: str
    size: 5
    encoding: ASCII
```

Within the `instances` block, a map is created where the keys represent attribute names, and the values specify attributes in a manner similar to how it's done in `seq`. However, an important additional feature is introduced: using `pos:...`, one can specify the position from which to start parsing that attribute (in bytes from the beginning of the stream). Expressions and references to other attributes can also be used in `pos`, similar to `size`.

Another significant difference between the `seq` attribute and the `instances` attribute is that instances are lazy by default. It means that unless someone would call that `body` getter method programmatically, no actual parsing of `body` would be done [12].

Let us first understand how data can be parsed from a specified position to achieve similar functionality in Julia. We need first to store the stream's current position to return to it after parsing the needed data. Then, we seek the stream to the specified position, read the data, and seek back to the stored position. It's important to ensure that the same instance is not parsed more than once. The following code snippet demonstrates how this can be implemented:

```julia
function get_header(this::InstanceStd)
  if this.header !== nothing
    return this.header
  end

  _pos = KaitaiStruct.pos(this._io)
  KaitaiStruct.seek(this._io, 2)
  this.header
      = decode(KaitaiStruct.read_bytes(this._io, 5), "ASCII")
  KaitaiStruct.seek(this._io, _pos)
  return this.header
end
```

Notice that from the programming point of view (from the target programming languages and internal Kaitai Struct‘s expression language), `seq` attributes and `instances` are the same. So users can access them the same way. To achieve it in Julia we can override `Base.getproperty()` method in the following way:

```julia
function Base.getproperty(obj::InstanceStd, sym::Symbol)
  if sym === :header
    return get_header(obj)
  else
    return getfield(obj, sym)
  end
end
```

4.7.8 Enums (named integer constants)

The nature of binary format encoding dictates that we‘ll often be using some integer constants to encode certain entities. For example, an IP packet uses a 1-byte integer to encode the protocol type for the payload: 6 would mean “TCP” (which gives us TCP/IP), 17 would mean “UDP” (which yields UDP/IP), and 1 means “ICMP” [12].

It is possible to live with just raw integers, but most programming languages provide a way to program using meaningful string names instead. This approach is usually dubbed “enums” and it‘s possible to generate an enum in the Kaitai Struct [12]:

```yaml
seq:
  - id: protocol
    type: u1
    enum: ip_protocol
enums:
  ip_protocol:
    1: icmp
    6: tcp
    17: udp
```

Corresponding Julia code would look like this:

```julia
@enum IP_Protocol::Int8 begin
  icmp = 1
  tcp = 7
  udp = 12
end

# protocol has type Union{IP_Protocol, Integer}
protocol = KaitaiStruct
  .resolve_enum(IP_Protocol, KaitaiStruct.readU1(_io))
```

In Julia, attempting to assign an `Integer` to a field declared as type `Enum` would throw an exception [9]. However, KSY allows the assignment of an invalid enum, treating it as an integer. We declare such fields as a union of integer and enum `Union{Integer, Enum}` to handle invalid enum values. Note that `resolve_enum()` from Kaitai Julia Runtime library will return `Enum` if there is a corresponding “key” in the given `Enum` found (it will try to cast the given `Integer`), otherwise `Integer` is returned.

4.7.9 Checking for “magic” signatures

Many file formats use some safeguard measure against a completely different file type instead of the required one. The simple way to do so is to include some “magic” bytes (AKA “file signature”): for example, checking that the first bytes of the file are equal to their intended values provides at least some degree of protection against such blunders.

To specify “magic” bytes (i.e. fixed content) in structures, Kaitai Struct includes a special `contents` key. For example, this is the beginning of a `seq` for Java `.class` files:

```
seq:
  - id: magic
    contents: [0xca, 0xfe, 0xba, 0xbe]
```

The description above can be translated into the Julia language like this:

```
magic = KaitaiStruct.read_bytes(_io, 4)
if !(magic == [0xca, 0xfe, 0xba, 0xbe])
  throw(KaitaiStruct.ValidationNotEqualError(...))
end
```

This code reads the first 4 bytes and compares them to `CA FE BA BE`. If there is any mismatch (or less than 4 bytes are read), it throws an error and stops parsing at an early stage, before any damage (pointless allocation of huge structures, waste of CPU cycles) is done [12].

There is no need to specify type or size for fixed content data. Contents are very flexible and you can specify:

- A UTF-8 string — bytes from such a string would be checked against
- An array with:
  - bytes in decimal representation
  - bytes in hexadecimal representation, starting with 0x
  - UTF-8 strings

All elements' byte representations would be concatenated and expected in sequence when using an array. Some examples [12]:

```
  - id: magic1
    contents: JFIF
    # expects bytes: 4A 46 49 46
  - id: magic2
    # we can use YAML block-style arrays as well
    contents:
      - 0xca
      - 0xfe
      - 0xba
      - 0xbe
    # expects bytes: CA FE BA BE
  - id: magic3
    contents: [CAFE, 0, BABE]
    # expects bytes: 43 41 46 45 00 42 41 42 45
```

4.7.10 Validating attribute values

The Kaitai Struct provides a mechanism for validating attribute values using the `valid` key to ensure attributes in data structures adhere to expected formats and ranges. This key allows you to define constraints for values, enhancing the robustness of your specifications [12].

- To ensure the attribute value exactly matches the given value `eq` (or directly `valid: value`) can be used [12]:

  ```
  # Equality constraint: the only valid value is 0x42
    valid: # can be shortened to valid: 0x42
      eq: 0x42
  ```

  Which will be translated into the Julia language like this:

  ```
  if !(exact_value == 0x42)
      throw(KaitaiStruct.ValidationNotEqualError(...))
  end
  ```

- `min` and `max`: specify the minimum and maximum valid value for the attribute [12]:

  ```
  # Value must be at least 100 and at most 200
    valid:
      min: 100
      max: 200
  ```

  And corresponding Julia code:

  ```
  if !(bounded_value >= 100)
      throw(KaitaiStruct.ValidationLessThanError(...))
  end
  if !(bounded_value <= 200)
      throw(KaitaiStruct.ValidationGreaterThanError(...))
  end
  ```

- `any-of`: defines a list of acceptable values, one of which the attribute must match [12]:

  ```
  # Value must be one of 3, 5, or 7
    valid:
      any-of: [3, 5, 7]
  ```

```
if !((enum_constraint_value == 3)
    || (enum_constraint_value == 5)
    || (enum_constraint_value == 7))
    throw(KaitaiStruct.ValidationNotAnyOfError(...))
end
```

- `expr`: an expression that evaluates to true for the attribute to be considered valid [12]:

```
# Value must be even
  valid:
    expr: _ % 2 == 0
```

```
if !(expr_constraint_value % 2 == 0)
    throw(KaitaiStruct.ValidationExprError(...))
end
```

When a value does not meet the specified criteria, the Kaitai Struct throws a validation error, halting further parsing. This preemptive measure ensures the data being parsed is within the expected domain, providing a first layer of error handling [12].

Chapter 5
Testing

The primary goal of testing Julia as a target language in the Kaitai CI system is to make sure that the generated Julia parsers correctly parse binary data according to the specified KSY specifications. Kaitai Struct standardizes testing using 233 tests. Each test consist of three parts: the KSY specification, a binary file, and the expected parsed values. The expected parsed values are stored in KST files.

The testing process starts with the generating parsers from KSY specifications. After that, the binary file associated with each test is parsed with the generated parser. The parsing output is then compared against the expected results.

Julia Runtime library is additionally tested on every push through CI system in KaitaiStruct.jl repository.

5.1 Unit testing in Julia

Simple unit testing in Julia can be performed using `Test` package. It provides the `@test` and `@test_throws` macros:

-
  ```
  @test ex
  ```

  Tests that the expression ex evaluates to `true`. It allows to call functions in a slightly more readable approach. The `@test f(args...) key=val...` form is equivalent to writing `@test f(args..., key=val...)` which can be useful when the expression is a call using infix syntax such as approximate comparisons.

-
  ```
  @test_throws exception expr
  ```

Tests that the expression expr throws an exception. The exception may specify either a type, a string, a regular expression, or a list of strings occurring in the displayed error message, a matching function, or a value.

Typically, many tests are used to ensure functions work correctly over a range of inputs. If a test fails, the default behavior is to throw an exception immediately. However, it is normally preferable to run the rest of the tests first to get a better picture of how many errors there are in the code being tested.

The `@testset` macro can group tests into sets. All the tests in a test set will be run, and a summary will be printed at the end of the test set. If any of the tests failed, or could not be evaluated due to an error, the test set will throw a `TestSetException` [27].

5.2 KST Translator

In the testing process, various KST specifications are used to define the expected behavior of Julia parsers for different scenarios. This Section shows three examples of these specifications and corresponding tests in Julia.

Some of Kaitai Struct CI's KST specifications are designed to test that a parser throws an exception when parsing wrong data. Consider this KSY specification:

```yaml
meta:
  id: valid_fail_range_float
seq:
  - id: foo
    type: f4le
    valid:
      min: 0.2
      max: 0.4
```

It checks that the read `Float32` is less than 2 and greater than 4. If the check fails, `ValidationGreaterThanError` should be thrown. The corresponding KST specification feeds the binary file with a value greater than 0.4 and asserts that the exception is thrown.

```yaml
id: valid_fail_range_float
data: floating_points.bin
exception: ValidationGreaterThanError<f4>
```

KST translator translates `.kst` specifications to unit tests in the specified target language. Here is how the specification above might look in Julia:

```julia
using Test
using TestReports
using ValidFailRangeFloat

@testset "ValidFailRangeFloat test" begin
    @test_throws KaitaiStruct.ValidationGreaterThanError
        ValidFailRangeFloat.from_file("floating_points.bin")
end
```

Most asserts are directly translated using the `==` operator. For example, consider these assertions:

```yaml
asserts:
  - actual: s1
    expected: '"foo"'
  - actual: s2
    expected: '"bar"'
  - actual: s3
    expected: '"|baz@"'
```

The resulting Julia code will look like this:

```julia
@test r.s1 == "foo"
@test r.s2 == "bar"
@test r.s3 == "|baz@"
```

Notice that comparisons with `nothing` should use `===`, and comparisons with floating-point values should use the ≈ operator to account for floating-point imprecision. The KST Translator has the functionality to handle these comparisons correctly, ensuring accurate validation in Julia tests.

5.2.1 Testing the Julia Runtime library

The Julia Runtime library uses GitHub Actions for automated testing on multiple platforms. The CI configuration `ci.yml`[1] is designed to run tests on supported versions of Julia, on multiple operating systems and architectures.

The CI setup tests the library with Julia 1.8, the latest stable Julia 1.x release, and the nightly build. These tests run on Ubuntu, macOS, and Windows operating systems, each with a 64-bit architecture. The workflow is triggered on pushes to the `main` branch, pull requests to `main`, and any new tags. This ensures that changes are tested in various scenarios.

This setup provides flexibility to test with additional Julia versions or operating systems as needed:

[1]`https://github.com/rystidia/KaitaiStruct.jl`

```yaml
matrix:
  version:
    - '1.8'
    - '1'
    - 'nightly'
  os:
    - ubuntu-latest
    - macOS-latest
    - windows-latest
  arch:
    - x64
```

To manage versions and releases, the package uses a tagging system. The CI workflow includes a TagBot configuration to automate the tagging process. TagBot is a GitHub application used in the Julia ecosystem to automate the process of tagging new releases of Julia packages and updating their registry entries. This tool creates a new tag in the Git repository and ensures that the Julia package registry is updated accordingly.

5.3 Evaluation

The KST Translator was extended to generate unit tests in Julia. Out of 233 tests in the Kaitai Struct CI test suite, 232 pass successfully. The exception is related to circular imports, which are atypical for Julia code. This presents only a minor complication as it can be resolved by making small changes to the `.ksy` format specification. The generated tests in Julia, along with scripts for running them (run-julia) and extracting the test results into the Kaitai Struct CI system (ci-julia), are available in my fork of the KS testing repository. A pull request[2] was submitted for these changes.

Kaitai Struct CI uses Docker images to test generated parsers. A script for building a Docker image with the required Julia version and all testing dependencies was implemented and merged into the KS repository for testing images[3].

The KS compiler was extended to generate Julia modules from the `.ksy` descriptions. The pull request for this change was submitted to the KS compiler repository[4].

The Julia runtime library was tested with different Julia versions, and a continuous integration system was set up for it.

Overall, the successful execution of the majority of tests, combined with automated testing across multiple Julia versions, validates the robustness and reliability of the generated parsing modules in Julia.

[2]https://github.com/kaitai-io/kaitai_struct_tests/pull/126
[3]https://github.com/kaitai-io/kaitai_struct_docker_images/pull/1
[4]https://github.com/kaitai-io/kaitai_struct_compiler/pull/305

Chapter 6
Conclusion

This project successfully added the Julia programming language to the Kaitai Struct as a target language. The biggest challenge was creating a parsing module design in Julia that would be consistent with other supported languages and respect Julia's unique style guide. It was achieved and Kaitai Struct compiler was extended to generate Julia modules from binary format descriptions in the `.ksy` format. The pull request for this change was submitted to the KS compiler repository[1] and is now under review. The project administrator provided positive feedback.

Implementing the language-specific runtime library was an important part of the work. The Julia runtime library was implemented and published under KaitaiStruct.jl name. A continuous integration system was set up for the Julia runtime library, ensuring that it is tested automatically with several supported Julia releases.

The solution was integrated into the Kaitai Struct CI pipeline. Out of 233 tests from the Kaitai Struct CI test suite, 232 passed, demonstrating the robustness and reliability of the generated parsing modules in Julia. The one exception is related to circular imports, which are atypical for Julia code. Kaitai Struct CI uses Docker images to test generated parsers. A script for building a Docker image with the required Julia version and all testing dependencies was implemented and merged into the KS repository for testing images[2]. The KST Translator was extended to generate unit tests in Julia. Scripts for running tests in Julia and extracting the test results into the Kaitai Struct CI system were created. A pull request[3] was submitted to the KS testing repository for these changes.

Overall, the project accomplished its goal of adding Julia to Kaitai Struct as a target language, establishing a solid foundation for future support.

[1]`https://github.com/kaitai-io/kaitai_struct_compiler/pull/305`
[2]`https://github.com/kaitai-io/kaitai_struct_docker_images/pull/1`
[3]`https://github.com/kaitai-io/kaitai_struct_tests/pull/126`

Appendix A
Source code

All code is available in public repositories on GitHub.

- Kaitai Struct runtime library for Julia:

  `https://github.com/rystidia/KaitaiStruct.jl/commit/fe9e99677 04d91b6ee3144eaf31368e4bc191060`

- Kaitai Struct compiler with Julia support:

  `https://github.com/kaitai-io/kaitai_struct_compiler/commit/23 54168f88c9c50ead7bb1f68cb38cbb59c5232f`

- Kaitai Struct tests:

  `https://github.com/kaitai-io/kaitai_struct_tests/commit/6879a 2f846833835a99133910fdd65dbcfe4c5f0`

- KS repository for testing images:

  `https://github.com/kaitai-io/kaitai_struct_docker_images/comm it/cc990ebc99e7e197856ec2511b9c986071bc4798`

As a backup, in case the GitHub repositories listed above are unavailable for any reason, an archive with clones of these GitHub repositories is submitted along with the thesis.

The `clone-github.sh` shell script records the commands that were used to create the `compiler/`, `julia_runtime/`, `tests/`, and `docker_images/` folders. The commit hashes checked out in each repository are the same as those used in `https://github.com`.

Project-related commits can be identified by inspecting the commit graph to see which commits were authored by me. I use the name “Dias Rystin” for all commits made using the local Git client, or “rystidia” for commits made via the GitHub web interface.

Project-related commits, excluding those for Kaitai Struct runtime library, also can be viewed in the following pull requests:

- KS repository for testing images:

  `https://github.com/kaitai-io/kaitai_struct_docker_images/pull/1`

- Kaitai Struct tests:

  `https://github.com/kaitai-io/kaitai_struct_tests/pull/126`

- Kaitai Struct compiler with Julia support:

  `https://github.com/kaitai-io/kaitai_struct_compiler/pull/305`

Bibliography

1. LINFO. *Binary File Definition (The Linux Information Project)* [online]. 2006. Available also from: `https://www.linfo.org/binary_file.html`. Accessed: 2023/12/25.
2. MURRELL, Paul. *Paul Murrell – Binary formats* [online]. 2007. Available also from: `https://www.stat.auckland.ac.nz/~paul/ItDT/HTML/node39.html`. Accessed: 2023/12/24.
3. MURRELL, Paul. *Paul Murrell – Binary files* [online]. 2002. Available also from: `https://statmath.wu.ac.at/courses/data-analysis/itdtHTML/node58.html`. Accessed: 2023/12/24.
4. BLANC, Bertrand; MAARAOUI, Bob. *White Paper: Endianness or Where is Byte 0?* [Online]. 2005. Available also from: `http://3bc.bertrand-blanc.com/endianness05.pdf`. Accessed: January 16, 2024.
5. FOLDOC CONTRIBUTORS. *FOLDOC (Free On-line Dictionary of Computing) – Endian* [online]. 2007. Available also from: `https://foldoc.org/endian`. Accessed: January 16, 2024.
6. FOLDOC CONTRIBUTORS. *FOLDOC (Free On-line Dictionary of Computing) – Type* [online]. 2003. Available also from: `https://foldoc.org/type`. Accessed: January 16, 2024.
7. FOLDOC CONTRIBUTORS. *FOLDOC (Free On-line Dictionary of Computing) – String* [online]. 2015. Available also from: `https://foldoc.org/string`. Accessed: January 16, 2024.
8. BEZANSON, Jeff; KARPINSKI, Stefan; SHAH, Viral B.; EDELMAN, Alan. *Why We Created Julia* [online]. 2012. Available also from: `https://julialang.org/blog/2012/02/why-we-created-julia/`. Accessed: 2023/12/24.
9. JULIALANG.ORG CONTRIBUTORS. *Julia Documentation* [online]. 2023. Available also from: `https://docs.julialang.org/en/v1/`. Accessed: November 18, 2023.

10. LOBIANCO, Antonello. *Julia Quick Syntax Reference: A Pocket Guide for Data Science Programming.* Apress, 2019. ISBN 978-1-4842-5189-8.

11. KWON, Changhyun. *Julia Programming for Operations Research.* 2019. ISBN 1798205475. Available also from: `https://juliabook.chkwon.net/book/introduction`. Accessed: November 18, 2023.

12. KAITAI PROJECT. *Kaitai Struct User Guide* [online]. 2024. Available also from: `https://doc.kaitai.io/user_guide.html`. Accessed: May 1, 2024.

13. GRUNE, Dick; REEUWIJK, Kees van; BAL, Henri E.; JACOBS, Ceriel J.H.; LANGENDOEN, Koen. *Modern Compiler Design.* 2nd. New York, NY: Springer, 2000. ISBN 978-1-4614-4699-6.

14. KAITAI PROJECT. *Kaitai Struct testing repository* [online]. 2024. Available also from: `https://github.com/kaitai-io/kaitai_struct_tests`. Accessed: May 1, 2024.

15. RAMANANANDRO, Tahina; DELIGNAT-LAVAUD, Antoine; FOURNET, Cédric; SWAMY, Nikhil; CHAJED, Tej; KOBEISSI, Nadim; PROTZENKO, Jonathan. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In: *USENIX Security.* 2019. Available also from: `https://www.microsoft.com/en-us/research/publication/everparse/`.

16. ZEEK PROJECT. *Spicy — Generating Robust Parsers for Protocols and File Formats* [online]. 2024. Available also from: `https://docs.zeek.org/projects/spicy`. Accessed: May 1, 2024.

17. MENDEL, Dion. *BinData - Reading and Writing Binary Data in Ruby* [online]. 2023. Available also from: `https://github.com/dmendel/bindata/wiki`. Accessed: January 16, 2024.

18. FILEIO.JL CONTRIBUTORS. *FileIO.jl documentation* [online]. 2022. Available also from: `https://juliaio.github.io/FileIO.jl/v1.16/`. Accessed: May 19, 2024.

19. LEVILLAIN, Olivier; NAUD, Sébastien; RASOAMANANA, Aina Toky. Work-in-Progress: towards a platform to compare binary parser generators. In: *IEEE Security and Privacy Workshops (SPW) (LangSec).* San Jose, United States, 2021. Available also from: `https://hal.archives-ouvertes.fr/hal-04001619`.

20. KAITAI PROJECT. *KSY Style Guide* [online]. 2023. Available also from: `https://doc.kaitai.io/ksy_style_guide.html`. Accessed: January 12, 2024.

21. JULIALANG.ORG CONTRIBUTORS. *Julia Documentation – Constructors* [online]. 2023. Available also from: `https://docs.julialang.org/en/v1/manual/constructors/`. Accessed: January 12, 2024.

22. KAITAI PROJECT. *Kaitai Struct Compiler* [online]. 2024. Available also from: `https://github.com/kaitai-io/kaitai_struct_compiler`. Accessed: May 1, 2024.
23. JULIALANG.ORG CONTRIBUTORS. *Julia Documentation – Strings* [online]. 2023. Available also from: `https://docs.julialang.org/en/v1/manual/strings/#man-concatenation`. Accessed: January 16, 2024.
24. JULIALANG.ORG CONTRIBUTORS. *Julia Documentation – Mathematical Operations and Elementary Functions* [online]. 2023. Available also from: `https://docs.julialang.org/en/v1/manual/mathematical-operations/`. Accessed: January 16, 2024.
25. KAITAI PROJECT. *Kaitai Stream API* [online]. 2020. Available also from: `https://doc.kaitai.io/stream_api.html`. Accessed: January 12, 2024.
26. ORACLE. *Java Documentation* [online]. 2014. Available also from: `https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`. Accessed: November 18, 2023.
27. JULIALANG.ORG CONTRIBUTORS. *Julia Documentation – Unit Testing* [online]. 2023. Available also from: `https://docs.julialang.org/en/v1/stdlib/Test/#Basic-Unit-Tests`. Accessed: January 16, 2024.