README.md

FUML

FUML (acronym for **Fu**nctional **M**inimal **L**anguage) is a data serialization language inspired from functional programming languages like F# and OCaml. It also borrows some ideas from TOML and YAML.

Data serialization language is a language which can be used to represent data and then can be translated into multiple programming languages. Think of FUML as combination of Protocol Buffers and YAML. It prescribes how the data would look like and how to describe the data using type theory.

Problems that I want to solve

- I like type theory and started looking for data serialization formats that allowed its use. I did not find any, so came up with a new language.
- I don't like YAML colon separator between property name and its value, and the use of hyphen for lists. I don't find TOML's `[table]` syntax that appealing. F# has a nice syntax for records and lists so adopted it in FUML.
- I wanted a format which gave the user a choice between using whitespace to cleanly represent data and a compact alternative format.

- In many places, I found lack of config files to explain and validate the properties extremely annoying. I wanted a language which forces schema design from the get go and not just an after thought.

Goals

- FUML should be easily readable by humans
- FUML should be easy to type.
- FUML must be backed by a schema
- FUML should support common data types used in functional programming languages
- FUML should be portable between different types of programming languages.

Specs

Comments

- You can add single-line comment like so:

```
// the answer
42
```

- You can also add multi-line comments like so:

```
(*
    42 is the answer to the “ultimate question of life,
    the universe, and everything”
*)
42
```

- You can also add both types of comments in the FUML schema:

```
<schema>

// returns
(*
    the answer to question of life, the universe
    and everything
*)
data: int
```

Types

- FUML documents can be thought to be an instance of a type.
- Following types are allowed in FUML:
  - Boolean
  - Integer
  - Float
  - String
  - List
  - Map
  - Tuple
  - Record
  - Sum Type
  - Option
  - Result
  - DateTime
  - Type alias

Boolean

- Boolean values are either `true` or `false` (all lowercase). Example:

```
true
```

  Corresponding schema:

```
<schema>

data: bool
```

Integer

- In FUML, integers are numbers without any fraction part.
  - For example, `-12` is an integer while as `12.40` is not.
- FUML has following data types:
  - `i8`
    - Range = -128 to 127

    - `i16`
        - Range = -32,768 to 32,767
    - `i32`
        - Range = -2,147,483,648 to 2,147,483,647
    - `i64`
        - Range = -2^63 to (2^63 - 1)
    - `i128`
        - Range = -2^127 to (2^127 - 1)
    - `u8`
        - Range = 0 to 255
    - `u16`
        - Range = 0 to 65,536
    - `u32`
        - Range = 0 to 4,294,967,296
    - `u64`
        - Range = 0 to 2^64
    - `u128`
        - Range = 0 to 2^128

- Example:

```
2022
```

  Corresponding schema:

```
<schema>

data: i32
```

- There's also an `int` data type, which is a type alias for `i32`:

```
2_147_483_647
```

  Corresponding schema:

```
<schema>

data: int
```

- Its optional to add `+` sign before positive integers. Thus, following two FUML documents

  ```
  +25
  ```

  and

  ```
  25
  ```

  are valid.

- Negative integers are prefixed with `−`. Example:

  ```
  −25
  ```

- Undescores can be added to enhance readability:

  ```
  25_000
  ```

Float

- Floats should be implemented as IEEE 754 binary64 values. Floats can consist of integer value followed by a fractional part. Example:

  ```
  3.14
  ```

  Here, integer value is `3` and fractional part is `.14`

  Corresponding schema

  ```
  <schema>
  data: float
  ```

- Float can also be represented by integer value followed by exponent:

  ```
  4e12
  ```

Here, integer value is `4` and exponent is `e12`.

- Integer + fractional part can also precede exponent:

```
4.78e12
```

- Float must not be just an integer, or if a decimal point is present, must include both integer and fractional part. Examples of invalid float values:
  - `50`
  - `50.`
  - `.5`

Strings

- Strings comprises of unicode characters, and are enclosed in double quotes. Example:

```
"Hello world!"
```

Corresponding schema:

```
<schema>

data: string
```

- Escape double quotes and other special characters using `\`:

```
"Hello, \"Peter\""
```

- Multi-line strings must be enclosed within `"""` (triple double quotes):

```
"""
This is first sentence.
This is second sentence.
"""
```

  - Triple quotes `"""` must be in seperate lines. So, following are invalid:

```
""" This is first sentence.
This is second sentence. """
```

  - Triple double quotes are not allowed inside multi-line strings.

- If you have a long sentence that you want to split, you can use `"""` followed by `&"<join character>"`. Example:

```
"""&"+"
For the binary formats, the representation is made
unique by choosing the smallest representable exponent
allowing the value to be represented exactly.
"""
```

  This will be equivalent to the following single-line string:

```
"For the binary formats, the representation is made+unique by
choosing the smallest representable exponent+allowing the value to be
represented exactly."
```

  - If `join character` is a double quote `"`, escape it via `\`.

List

- Lists are a collection of values having the same data type. Example:

```
[
    1
    2
    3
]
```

  Corresponding schema:

```
<schema>

data: int list
```

  - List types are written in postfix notation. `int list` is equivalent to `List<Integer>` in Java.

- List elements can also be written in compact form as below:

```
[1, 2, 3]
```

  - Space between comma is not required, but recommended.
  - Values in list, if on the same line, are separated by comma.

- You can mix-and-match separating values via comma and writing them in new lines:

```
[
    1, 2
    3, 4
    5,6
]
```

Tuple

- Tuples are data types which can store multiple types of data in a specific order. Example:

```
(2, ["apple", "banana"], "fruits")
```

  Corresponding schema:

```
<schema>

data: int * (string list) * string
```

- Tuple values are seperated by a comma `,` and are enclosed within a pair of round brackets `(` and `)`.
- Tuples must be written in a single line.
  - If the tuple size goes big, then you should consider modeling the data as a record which allows for a more readable format.
  - Record also allows you to name the values, something which is not possible in a tuple.

Map

- Maps are a list of key-value pair. Key and value may have different datatypes, but must be the same across all pairs in the map. Example:

```
{
    "Thousand" ⇒ 1_000
    "Million" ⇒ 1_000_000
}
```

Corresponding schema:

```
<schema>

data: (string * int) map
```

- Map pairs are represented as `<key> ⇒ <value>`
- Pairs are modeled as tuples, hence the data type above for each pair is `(string * int)`

- Only integer, string and enum data types are allowed for map keys. Using any other data type should throw a compilation error.
- Difference between a map of key-value pairs and a similar list of pairs is that in a map, duplicate keys are not allowed. Thus, the following should throw an error during deserialization:

```
"Thousand" ⇒ 1_000
"Thousand" ⇒ 1_000_000
```

while as this is allowed in a list:

```
[
    ("Thousand", 1_000)
    ("Thousand", 1_000_000)
]
```

- Maps can also be written in compact form as below:

```
{"Thousand"⇒1_000,"Million"⇒1_000_000}
```

Record

- Records. Example:

```
name = "Sumeet Das"
username = "sumeetdas"
```

Corresponding schema:

```
<schema>

type GithubUser =
    name: string
    username: string

data: GithubUser
```

- Records can also be written in compact form:

```
{name="Sumeet Das",username="sumeetdas"}
```

- FUML recommendations for naming record types:
  - It should follow CamelCase convention
  - The first letter should be uppercase
- Some property names aren't just names; they are sentences. You can use round brackets `(` and `)` to use sentences as property names:

```
username: "sumeetdas"
(has the user completed the course?): true
```

  - Such field names can also include special characters like `?`
  - You can use round brackets `(` and `)` here via escaping them using `\`
- You can directly assign nested property values:

```
fruits.apple.(weight in grams) = 85
```

  - If the nested property does not exist, it should result in an error during deserialization.
- Nested records. Example:

```
name = "Sumeet Das"
username = "sumeetdas"
stats =
    (number of projects) = 10
    (number of followers) = 20
    stars = 30
```

Corresponding schema:

```
<schema>

type GithubStats =
    (number of projects): int
    (number of followers): int
    stars: int

type GithubUser =
    name: string
    username: string
    stats: GithubStats

data: GithubUser
```

- List of records:

```
[
    name = "Cat"
    sound = "meow"
    ,
    name = "Dog"
    sound = "woof"
]
```

Compact form:

```
[{name="Cat",sound="meow"},{name="Dog",sound="woof"}]
```

Corresponding schema:

```
<schema>

type Animal =
    name: string
```

```
    sound: string

data: Animal list
```

- Nested list of records:

```
animals = [
    name = "Cat"
    sound = "meow"
    ,
    name = "Dog"
    sound = "woof"
]
```

  You can also use compact form of records as below:

```
animals = [
    {name = "Cat", sound = "meow"}
    {name = "Dog", sound="woof"}
]
```

  - If one record is written in compact form, others too must follow the same pattern.
  - Compact form records don't need comma `,` in between if they are written in separate lines.

- Map to a record:

```
"Cat" ⇒
    family = "Felidae"
    sound = "meow"
"Dog" ⇒
    family = "Canidae"
    sound = "woof"
```

  Compact form:

```
{"Cat" ⇒ {family="Felidae",sound="meow"},"Dog"⇒{family="Canidae" ,
sound = "woof"}}
```

Generic records

- If a record type uses generic types, then such a type is called generic record type. For example, if you want to create a type `Pair` for storing two different types of value, you can define it as:

```
<schema>

type ('x, 'y) Pair =
    valueA: 'x
    valueB: 'y
```

To use `Pair` type, you need to provide types for `'x` and `'y` generic types:

```
<schema>

type CustomType =
    propertyA: (int * string) Pair
    propertyB: (float * int) Pair

data: CustomType
```

One example of FUML document for above schema would be:

```
propertyA =
    valueA = 12
    valueB = "some string"
propertyB =
    valueA = 4.5
    valueB = 100
```

Sum Type

- Sum types are data structures that can take on several different, but fixed, types.
- To understand it, lets consider the following example - Suppose you want to create a type called `Shape` which can accept instances of different types of shapes like `Circle`, `Rectangle`, `Polygon`. Or if you don't have any shape to store, the type will accept a `NoShape` instance. To implement it, you can define `Shape` as a sum type:

```
<schema>

type Sides =
    numberOfSides: int
    sideLengths: int list
```

```
type Shape =
    | Circle of int
    // Length * Breadth
    | Rectangle of int * int
    | Polygon of Sides
    | NoShape

data: Shape
```

This schema can accept the following FUML documents, each of which represents an instance of a shape type:

```
Circle 5
```

```
Rectangle (5, 3)
```

```
Polygon
    numberOfSides = 5
    sideLengths = [4, 4, 4, 4, 4]
```

```
NoShape
```

- FUML recommendations for naming sum types:
    - It should follow CamelCase convention
    - The first letter should be uppercase
- You can also use one of the sum types:

```
5
```

Corresponding schema:

```
<schema>

data: Shape.Circle
```

    - For types like these, you only need to provide the parameter values. For example, here you only need to specify `5` as the `int` value an instance of `Shape.Circle`

type expects.

- Individual types in sum types (e.g. `Circle` and `Rectangle` in `Shape` data type), if used as a data type for a property, requires fully qualified name. For example, you cannot use `Circle` type as follows:

```
<schema>

type Shape =
    | Circle of int

data: Circle
```

- Instead, the correct way to use `Circle` type is to use its fully qualified name `Shape.Circle`:

```
<schema>

//..

data: Shape.Circle
```

- A schema must not define any property whose type is a sum type with no parameters. For example, the following is an invalid schema:

```
<schema>

type Shape =
    | NoShape

data: Shape.NoShape
```

as `NoShape` type expects no parameters and hence makes no sense to use it as a property's data type.

Generic Sum Type

- Its also possible to create generic sum type. In fact, the following two types - `Option` and `Result` - are generic sum type.
- To understand generic sum type, let's look at the type definition of `Option` type:

```
type 't Option =
    | Some of 't
    | None
```

- `'t` is a generic type which must be provided when a property is defined to be an `Option` type.
- The generic sum type `Option` can be used as follows:

```
<schema>

type CustomType =
    propertyA: int option
```

- The above schema defines a property named `propertyA` which is an integer `Option`. This means the property can accept either `Some <integer-value>` or `None`.

Enum

- Enums are a variant of sum types which do not accept any parameters.
- They are similar to enums in C and can be treated as constants.
- For example, suppose you want to define a type which would accept only a select few colors. You can define enum type called `Color` as follows:

```
<schema>

enum Color =
    | Red
    | Green
    | Blue

data: Color
```

- Enums are defined using keyword `enum`. Rest of the syntax is similar to sum type.
- The above type `Color` only accepts `Red`, `Green` and `Blue`. So, the following FUML document is allowed:

```
Red
```

while as any other type would result in runtime error:

```
Yellow
```

- You can also define a map from enum to another data type. For example, to map colors to hex values, you can define a type as follows:

```
<schema>

enum Color =
    | Red
    | Green
    | Blue

data: (enum * string) map
```

which will accept a map as follows:

```
Red ⇒ '#FF0000'
Green ⇒ '#00FF00'
Blue ⇒ '#0000FF'
```

Option

- Option is a sum type which is defined as follows:

```
type 't Option =
    | Some of 't
    | None
```

where `'t` could be any type.

- This data type is useful when you want to model a nullable data. In other words, a property which may or may not have a value.
- Example:

```
None
```

```
Some 2
```

Corresponding schema:

```
<schema>

data: int Option
```

  - If the value is present, use `Some <value>`
  - If the value is not present, use `None`

- This type has `None` as default. If a property is not included in the FUML document, then its value is assumed to be `None`.
- You can also drop `Some` and directly write the value. Thus, the following FUML document:

```
2
```

  would be valid for above schema.

Result

- Result is a sum type which is defined as follows:

```
type 'x, 'y Result =
    | Ok of 'x
    | Error of 'y
```

- Result type can be used in cases when you want to return result of an operation if its successful, or an error response in case of failure.
- Example:

```
Ok 5
```

```
Error "error happened"
```

  Corresponding schema:

```
<schema>

data: (int * string) Result
```

DateTime

- DateTime is a sum type used to represent multiple formats of date and time. DateTime type is defined as:

```
type DateTime =
    // example: 1985-04-12T23:20:50.123456Z (T can be omitted)
    | UtcDateTime of string

    // example: 1996-12-19T16:39:57-08:00 (T can be omitted)
    | OffsetDateTime of string

    // example: 1996-12-19T16:39:57.123456-08:00 (T can be omitted)
    | OffsetWithFractionDateTime of string

    // example: 1996-12-19
    | YearMonthDate of string

    // example: 07:32:00
    | LocalTime of string

    // example: 00:32:00.123456
    | LocalTimeWithFraction of string
```

- Using incorrect date or time format with a given Format type must throw an error during deserialization. For example, the following will result in an error:

```
UtcDateTime "1996-12-19"
```

  Corresponding schema:

```
<schema>

data: DateTime
```

- Oftentimes, we don't accept multiple date-time formats. To specify which format to accept, you can use fully qualified name of Format type as the data type. For example, if you want to use OffsetDateTime as the format, you can do so as follows:

```
"1996-12-19T16:39:57-08:00"
```

  Corresponding schema:

```
<schema>

data: DateTime.OffsetDateTime
```

  - Using Format data type would allow you to directly use the string containing date and/or time.

- Date and time formats follow the RFC 3339 specs.

Type alias

- You can use type alias to rename a type.
- Example:

```
(50, "Jay")
```

  Corresponding schema:

```
<schema>

// type alias
type User = int * string

data: User
```

- FUML recommendations for naming type aliases:
  - It should follow CamelCase convention
  - The first letter should be uppercase
- Type alias definition must be in a single line. The following is invalid:

```
<schema>

// invalid; should throw a compilation error
type User =
    int * string
```

- You cannot name a type alias as any one of the lowercase pre-defined types:
  - any of the integer types like `int` and `i64`

    - `float`
    - `string`
    - `map`
    - `list`

  So, the following would result in a compilation error:

```
<schema>

type list = i32
```

- If there are two type alias definitions, the latest definition would be considered. Example:

```
<schema>

type WholeNumber = i8

type WholeNumber = i32
```

  Here, `WholeNumber` would be a type alias for `i32`.

    - One corollary of this rule is that you can define a type alias named `DateTime`. This would effectively replace the existing `DateTime` sum type with the new type alias. For example:

```
<schema>

type DateTime = string
```

  would make `DateTime` an alias of type `string`.

Files and Namespaces

FUML Files

- Any type of FUML content, be it FUML document or FUML schema, must be stored in files with extension `.fuml`.
- Schema file names must start with either an underscore (_) or an uppercase letter, followed by any number of uppercase letters, lowercase letters, underscores and digits. Examples:

```
_Schema.fuml

Schema_123.fuml
```

- Schema files must have the tag `<schema>` at the beginning of the file. Example:

```
<schema>

type User =
    name: string
    username: string

// ...
```

- Every schema file must end with `data: <type-name>` .

FUML Namespaces

- FUML schema can be split in multiple files. However, all FUML files must be stored under one directory.
- Typical folder tree involving large number of FUML schemas might look the following:

```
<base-directory>
|
|--- base.fuml (optional)
|--- SchemaA.fuml
|--- SchemaB.fuml
|--- NamespaceA
    |--- SchemaC.fuml
    |--- SchemaD.fuml
|--- NamespaceB
    |--- SchemaE.fuml
    |--- SchemaF.fuml
```

- Directories inside `<base-directory>` are called **Namespaces**.
  - They are used to group similar schema files together.
  - They also allow using schemas with same name by storing them under different namespaces.

For example, consider you want to store Twitter and Github user data, and would want to create schema named `User.fuml` to model it. Since you cannot store two files named `User.fuml` in a single directory, you create two namespaces `Twitter` and `Github` and create `User.fuml` schema in each namespace.

  - `<base-directory>` is called as **root namespace**.

- `base.fuml` is a file which contains information about the order in which FUML files need to be compiled.

  - If the file is not present, the default order of compilation is recursively compile namespaces in alphabetical order. In each namespace, schema files would be compiled in alphabetical order.

  - To define compile order, use `compile` keyword.

  - For example, if you want to compile `NamespaceA` schemas, then `NamespaceB` schemas, followed by `SchemaB.fuml` and `SchemaA.fuml` in root namespace, the `base.fuml` contents would look like:

```
compile "NamespaceB"
compile "NamespaceA"
compile "SchemaB.fuml"
compile "SchemaA.fuml"
```

  - Schemas in `NamespaceB` and `NamespaceA` in the above example would be compiled in alphabetical order. If you want to change the order of compilation for `NamespaceB` , you need to explicitly specify the order for all files in the namespace:

```
compile "NamespaceB.SchemaD.fuml"
compile "NamespaceB.SchemaC.fuml"
```

Import schemas

- Schemas are imported automatically, provided they have been compiled before. For example, if `SchemaB` is a schema stored in `SchemaB.fuml` file, and `SchemaC` is another schema stored in `SchemaC.fuml` file under `NamespaceA` directory, then you can make use of these two schemas directly as follows:

```
<schema>
type ComplexType =
    someProperty: SchemeB
    anotherProperty: NamespaceA.SchemaC
```

```
data: ComplexType
```

provided the `base.fuml` file compiles `SchemaB` and `SchemaC` before `ComplexType`:

```
compile "NamespaceA"
compile "SchemaB"
compile "ComplexType"
```

- If the schemas are not compiled before importing them, then it would result in a compilation error.
- Namespace schemas will still be referenced using their fully qualified names (e.g. `NamespaceA.SchemaC`) as opposed to using just their names (e.g. `SchemaC`) when used as a property's data type.

Property metadata

- By default, all properties are required (meaning they need to have valid values), except for optional type which has `None` as default.
  But what if you want to use some default value when a property is missing? To allow that, you can make use of property metadata syntax.
- In FUML schemas, you can provide additional metadata about the property via the following syntax:

```
<schema>

data: i32
    metadata1 = 2
    metadata2 = <some value>
    // ...
```

- Metadata is allowed only for properties having integer, float or string types, or having type as a type alias mapping to one of these three types.
- Using metadata syntax, you can specify the default value as follows:

```
<schema>

type Fruit =
    name: string
    producer: string
```

```
            default = "Fruit company"
        (price per kg): float
            default = 4.0

data: Fruit
```

In this schema, `name` property is required as there's no default value defined for it, while as `producer` and `(price per kg)` properties are optional. If `producer` is missing, its value would be `"Fruit company"`, while as if `(price per kg)` is missing then its value would be `4.0`.

License

Releases

No releases published

Packages

No packages published