COMS W4115 Project Proposal
Super Serial
A Data Serialization Definition Language

D. Kaminsky

Fall 2014

1 Overview

The `Super Serial` language is a domain-specific programming language intended to specialize in concisely and unambiguously defining semantics around the serialization, deserialization and transformation of data.
The features of this language will be designed around providing for the following three basic capabilities:

1. The ability to model a well-structured data domain (e.g. a JSON object, XML, etc.)
2. The ability to define various encode and decode operations on each data domain using a common vocabulary
3. The ability to transform the data in place within a defined data domain

Sample canonical use cases for the `Super Serial` programming language:

- Decode, transform, then re-encode data using a single encoding (e.g. update values in a JSON object database).
- Translate from one encoding of a particular data domain to another encoding of the same data domain (e.g. translate from JSON text to BSON)
- Translate from one encoding of a particular data domain to an encoding of an alternate data domain (e.g. translate from JSON text to an XML structure)

Additionally, having the ability to accurately and clearly describe a method for encoding and decoding data has several real-world applications. For example:

- Translating enterprise message bus events or remote procedure calls into native, locally consumable objects
- Bridging the gap between *legacy* platforms and *next generation* applications within a complex technical infrastructure.
- Dealing with external vendor platforms that send data in a proprietary format or expect to receive it in a proprietary format.
- Maintaining wire-compatible language-specific implementations of an open source serialization framework library.

2 Language Properties

The `Super Serial` language is a *strongly typed*, *imperative* programming language with some powerful built-in functional language features which replace standard imperative control flow manipulation semantics.
It is a high level language, with first class constructs to represent the primary elements of its domain:

type Describes one aspect of the the data domain of the program

predicate Represents a boolean function that can be applied as a constraint to an existing datatype (e.g. require that an input string does not contain a particular character) or an allowable exception to a parsing rule (e.g. allow whitespace, ignore comments).

encoder, decoder Reads or writes elements of the data domain.

3 Target Execution Platforms

Primary execution platform will be a *command line interpreter* that will implement the core language specification. The architecture of the compiler will be left open to extension so that future additions can allow generation of source or bytecode for various language targets. **Note:** This is not a "time-permitting" goal. This is an "if this project turns out interesting and valuable, this would enable reasonable next steps to be taken outside of the scope of COMS W4115" goal.

4 Design Principles

The `Super Serial` language is designed with the following principles in mind:

- Focus on usability
  - Clear, human-readable syntax
  - Descriptive naming of language constructs
  - Discourage use of comments in favor of more fluent code
- Open to extension
  - Building block architecture allows composition of complex structures from simple ones
  - Expose core language semantics to enable customization
  - Allow pluggable execution target for potential future expansion

5 Project Roles

Manager	Douglas Kaminsky
Language Guru	Douglas Kaminsky
System Architect	Douglas Kaminsky
Verification and Validation	Douglas Kaminsky

6 Sample Program

This section shows a sample "interesting" program definition using the `Super Serial` language.

```
namespace Core.String

predicate StringPredicate
{
    can apply -> string
}

predicate StrictRegExPredicate is StringPredicate
{
    "expr" -> string
    to apply -> input matches "expr"
}
```

```
namespace JSON
using Core.String

predicate ValidString is StrictRegexPredicate([^\u005C\u0022])

type UndefinedType is literal 'undefined'

type NullType is literal 'null'

type BooleanType is option of (literal 'true', literal 'false')

type StringType is string with escape '\u005C',
                               constraint ValidString

type ObjectPair
{
  "key" -> StringType
  "value" -> any JSON
}

type Object
{
  "members" -> array of ObjectPair with unique "key"

  can get -> string
  to get -> first (find "members" where "key" = input)
}

type Array
{
  "members" -> array of any JSON

  can get -> int32
  to get -> nth input "members"
}

type SciNotationNumber
{
  "significand" -> Number
  "exponent" -> Number
}

type Number is option of (numeric literal 'Infinity',
                          literal 'NaN',
                          float64,
                          SciNotationNumber)
```

```
encoder Text
{
  to write Object -> write '{'
                     write each "members" with separator ','
                     write '}'

  to write ObjectPair -> write "key"
                         write ':'
                         write "value"

  to write Array -> write '['
                    write each "members" with separator ','
                    write ']'

  to write StringType -> write '"' input "'"

  to write SciNotationNumber -> write "significand"
                                write 'e'
                                write "exponent"

  to write any -> write input
}
```

```
decoder Text
{
  "whitespace" <- StrictRegexPredicate([\t\r\n\f ])

  to read Object -> expect '{' with allow "whitespace"
                    read each "members" with
                      allow "whitespace", separator ','
                    expect '}' with allow "whitespace"

  to read ObjectPair -> read "key"
                        expect ':' with allow "whitespace"
                        read "value"

  to read Array -> expect '[' with allow "whitespace"
                   read each "members" with
                     allow "whitespace", separator ','
                   expect ']' with allow "whitespace"

  to read StringType -> expect '"' with allow "whitespace"
                        read input
                        expect '"' with allow "whitespace"
                        expect ':' with allow "whitespace"

  to read SciNotationNumber -> expect "significand"
                               expect 'e' or 'E'
                               expect "exponent"

  to read any -> read input
}
```

```
namespace Main
  "jsonInput" <- JSON.Text decode |in.txt|

  to transform "rawJson" ->
    using "rawJson" as
      Object : transform each "members"
      Array : with get each "members"
                using input as
                  SciNotationNumber :
                    input <-
                      Number("significand" * 10 exp "exponent")
                  any : void
      ObjectPair : using "value" as
                     SciNotationNumber :
                       input <-
                         ObjectPair("key",
                           Number("significand" * 10 exp "exponent"))
                     any : void
      SciNotationNumber :
          input <- Number("significand" * 10 exp "exponent")
      any : void

  JSON.Text encode "jsonInput" |out.txt|
```

This program reads in a JSON text file, expands all scientific notation numbers to their full numeric form, then (assuming no overflow error has occurred) writes out the transformed data to a new file.