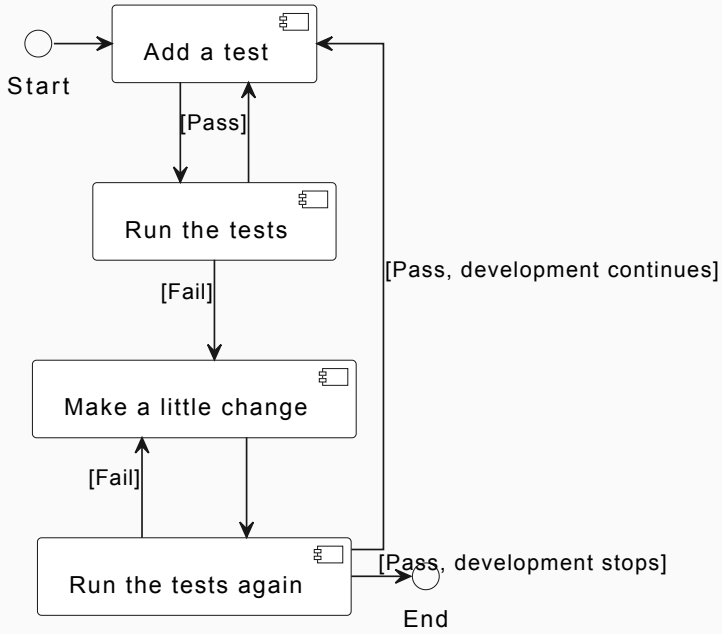
Type-Extensible Object Notation: JSON with Syntax for Types

Test-Driven Development

Testing has implications not only for the approach to implementation, but also the structure of the code itself and the overall practices surrounding projects. As each feature needs to be developed on a foundation of demonstrable necessity, the process must be divided into small enough steps for testing to be possible. The testing of a smaller component in a large codebase is referred to as `unit tests`, as each component becomes a testable `unit`.

The testing of a unit either aims to demonstrate a flaw in the current system, or demonstrate the functional implementation of a feature that meets the requirements for acceptance.

A test should ideally be executed in a short amount of time, resulting in fast testing. Tests should be executable in isolation, resulting in unordered testing. Tests should use production data when applicable, and tests should represent a feature or component of an overall project and implementation.



Test cycle: Guernsey, M. (2013). Test-driven Database Development: Unlocking Agility. Addison-Wesley.

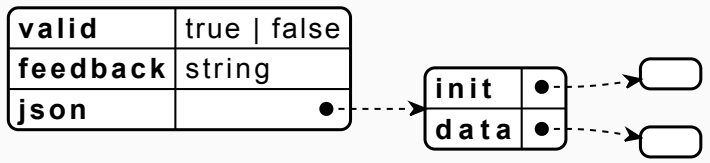
Unit and Acceptance Testing

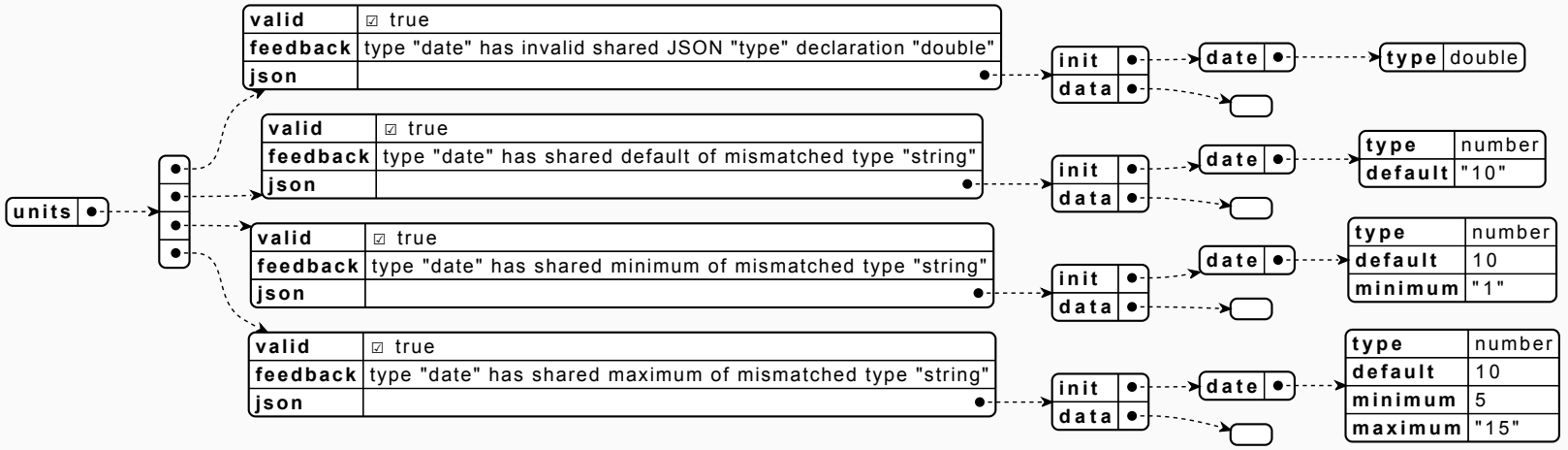
In this project the units are exclusively validation checks that return an error based on nonconformance to requirements or types in a data structure. A unit is a JSON data structure consisting of a sample nonconforming JSON object, its expected validity or invalidity, and an optional expected feedback string describing the source of nonconformance.

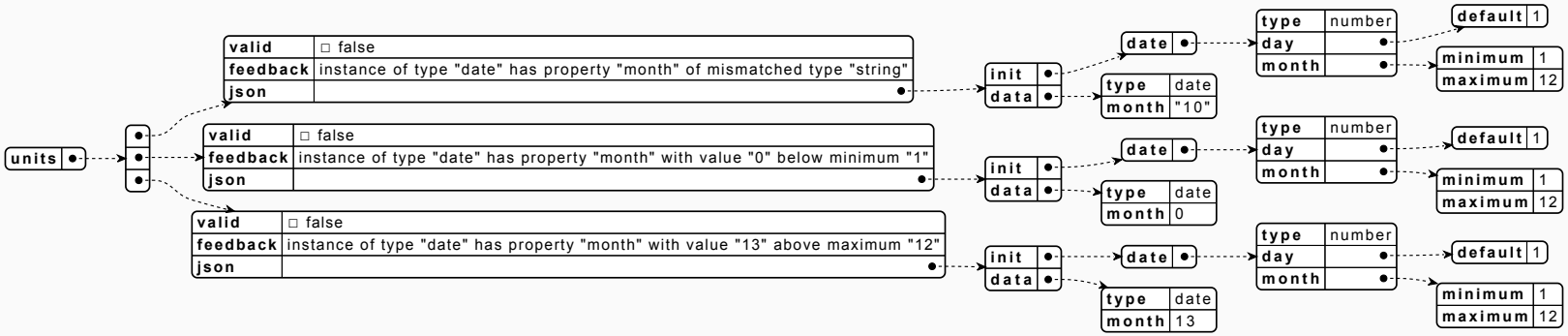
These units act as acceptance tests that describe the requirements for successfully implementing a feature through validation. This also results in implementing type semantics, as a feature can be utilised once its correct application can be verified.

These units do not act as tests demonstrating the necessity for an implementation or evolution of the JSON specification, which I identify as a flaw of the experiment documented in this project.

```
{
  "valid": "true | false",
  "feedback": "string",
  "json": {
    "init": {},
    "data": {}
  }
}
```







Unit tests for type declarations (top) and type instances (bottom)

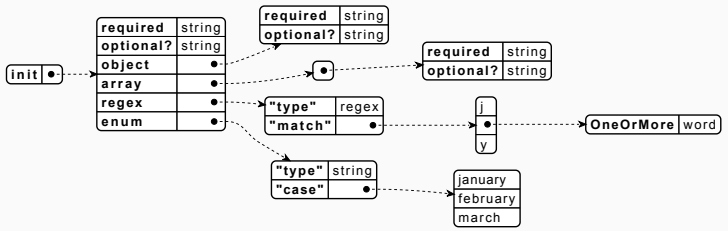
Type Semantics

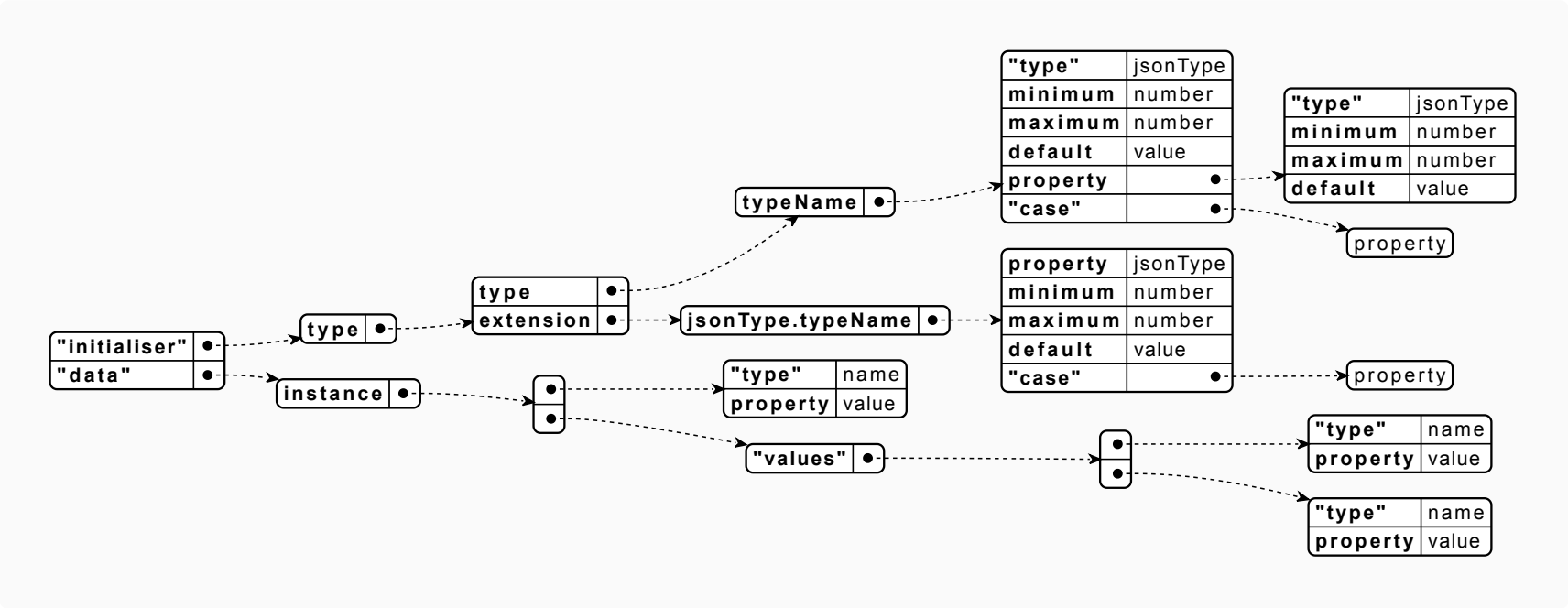
The extensible implementation of types through this project meant that nodes in a data structure were only validated once they were explicitly typed. This choice ensures that untyped data structures remain valid JSON data, but has implications for designing the semantics of types.

As a semantic feature was added to the implementation, the possible combinations increased exponentially. As such the validation had to test for all possible cases and combinations of semantics, which in turn informed the design of type semantics.

In reflecting on this implementation it became evident that it is missing some features, and that there are better grammatical alternatives to the type semantics derived from testing. This includes semantics for `single` `optional` `arrayrised` `regex` and `enumerated` values.

```
{
  "init": {
    "required": "string", "optional?": "string",
    "object": { "required": "string", "optional?": "string" },
    "array": [ { "required": "string", "optional?": "string" } ],
    "regex": { "type": "regex", "match": [ "j", { "OneOrMore": "word" }, "y" ] },
    "enum": { "type": "string", "case": [ "january", "february", "march" ] }
  }
}
```





Current implementation of type semantics from proposal

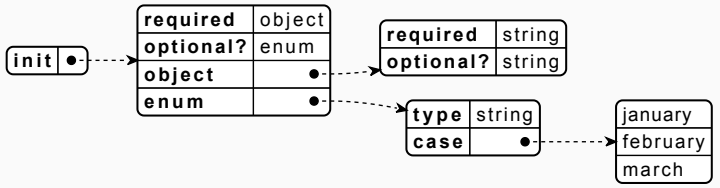
Relational Type

As a typed object typically inherits other types in object-oriented programming, there is a necessity for declaring relational types as properties of a typed object. This is evident in the implementation through TypeScript, but is missing from my implementation.

As a result types cannot reference each other when declared, but can be nested inside each other when instantiated. This is a suboptimal implementation because it results in the intermediary data structure not reflecting the data structure from which it is derived.

As an alternative it should be possible to reference not only types from the JSON typeset, but also types that have been declared. Types should be unordered, meaning a type can reference another type before the other type has been declared.

```
{
  "init": {
    "required": "object",
    "optional?": "enum",
    "object": {
      "required": "string", "optional?": "string"
    },
    "enum": {
      "type": "string", "case": [ "january", "february", "march" ]
    }
  }
}
```



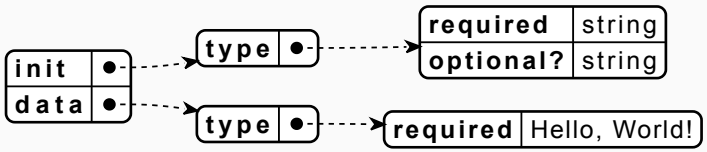
Optional Type

A typed data structure is verified based on its type conformance, as it is crucial to ensure that the data is correctly formatted and contains the necessary contents to be used in an application. However there can be data that is explicitly not necessary, or data that can have no value.

These nullable types can be considered optional, as opposed to required types. The current implementation facilitates nullable types by providing default values, which is a typical practice in object-oriented programming. This is however not semantically ideal for an intermediary data structure, which would not know or necessitate defaults.

As an alternative the nullable types could use the `null data type` from the JSON specification, but this would result in a loss of intended type. Instead I propose using the question mark syntax (?) for optional types.

```
{
  "init": {
    "type": {
      "required": "string", "optional?": "string"
    }
  },
  "data": {
    "type": {
      "required": "Hello, World!"
    }
  }
}
```



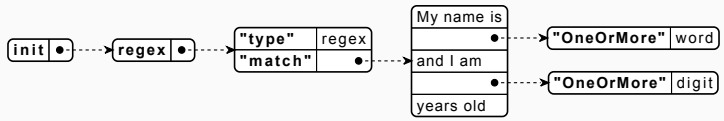
Regular Expression (RegEx)

The current implementation only considers binary validation of types and instances, as they either conform or do not conform. However it is typical to expect some degree of syntactical errors in the contents of a data structure, and this is where a `regular expression` can be applied.

The regular expression is a sequence that acts as pattern for finding or filtering a value of type String. This could be quite easily implemented in the current type semantics, by declaring types as objects containing an explicit regex "type" and "match" describing the sequence.

As an alternative to sequences of type String, the Swift 5.7 implementation of regex provides a grammar for arrayrised sequences. It also extrapolates the appropriate regex syntax from more natural human-readable text.

```
{
  "init": {
    "regex": {
      "type": "regex",
      "match": [
        "My name is",
        { "OneOrMore": "word" },
        "and I am",
        { "OneOrMore": "digit" },
        "years old"
      ]
    }
  }
}
```



Enumerated Type

The contents of a type instance can have requirements beyond its type and whether it is required or optional. These requirements in the current implementation are limited to minimum and maximum value ranges, for numbers and string or array lengths. However the TypeScript implementation provides enumerated types, which are the valid cases of a value.

This could be quite easily implemented in the current type semantics, by declaring types as objects containing an explicit "type" and arrayrised "case" values. If the validation was implemented in a text editor, this could illustrate nonconformance before data transmission, as is typical of enumerations in statically typed programming languages through compilation.

```
{
  "init": {
    "date": {
      "month": {
        "type": "string", "case": [ "january", "february", "march" ]
      }
    }
  },
  "data": {
    "date": {
      "month": "january"
    }
  }
}
```