Correcting errors by providing structural information to parsers

A practical example using JavaScript Object Notation

Mattij Ugen
m.ugen@student.utwente.nl

ABSTRACT
As humans are prone to making errors in manually encoding data, much work has gone into automatically correcting errors when handling the input data with a computer. Still, computers don’t always know what humans mean when an error is encountered. Ways of defining the structural properties of data have existed for quite some time. Although traditionally used for validation, structural information can be a valuable source for error correction in parsers. This paper describes an approach of using predefined structural information to make decisions on how to correct errors in input data.

Keywords
Human error, language recognition, error correction

1. INTRODUCTION
The concept of human errors in the world of computing is well known and has been around since the introduction of the first programming language. Most of the things that are intended to be read by a computer are tightly defined and strictly handled. The result of strict behaviour is that a single missing or extraneous character can often set a computer off, reporting an error or acting in a way not expected by the person who unwittingly specified that behaviour. Because a human languages differ so much from a programming languages, much work has gone into reporting useful errors to a programmer [1, 2] and ways to automatically correct errors that might be solvable [5, 8].

Apart from syntactical errors, a human might make certain errors that a computer will never identify as such. A parser that finds a character sequence in a way it recognizes would not report an error, whereas the application using the result of the parser might expect an integer. Seen as said integer was provided as "123" in stead of 123 did not throw off the parser.

This paper describes a way of providing a parser that accepts a language intended for use by computers with additional information about the input it is about to parse. By validating what has been found in the input string against the provided set of facts about what it should have been, a parser can identify errors and make informed decisions on what needs to be done with erroneous input.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2. MOTIVATION
Computers interact with each other to exchange information or specify actions needed to be taken. In order for both sides of this communication to understand each other, a communication protocol is usually defined. Such protocols are defined by human programmers, who define the rules in ways that make sense to them. Because of this, protocols and languages to be used by computers are often more or less readable by a human. Think of the common keywords if and while. This is mostly a good thing, as humans have to program the computer before they can use the previously defined protocols.

The human readability of languages that are meant for computers has the side effect that humans can easily interact with computers via the aforementioned protocols. Whereas manually typing a HTTP request in a terminal is not something that is common to do, specifying how a document or web page has to look is more so. It is here that the concept of human errors becomes a problem for the computer that has to interpret the code or specifications created by a human.

The motivation for this research is the ability to better detect and correct errors in data specified by a human.

3. PREVIOUS WORK
The communication protocols computers use to interact are always strictly defined. Interpretation errors are not always part of that definition. The specification of HTML 4.01, for example, states that error handling is not part of the standard definition [10]. Although the World Wide Web Consortium (W3C) provides a set of recommendations on how to treat erroneous data [9], user agent developers can develop their own error handling mechanisms.

The eXtensible Markup Language (XML) has its own standardized ways of defining structural properties about a document. Both Document Type Definitions (DTDs) [13] and XML-schemas [12] are capable of describing the structural content of an XML-document [6].

It seems as though little has been said on using this structural information as a means to look for error corrections; structural specifications are mostly used for validation of a document, stating whether the document is a valid instance of the document class defined by the structural definition [11].
4. RESEARCH QUESTIONS

To be able to correct errors, a definition of the word error is required. Although an error is a well known concept, there are two sorts of errors that play a role in this research. First, there is the syntactical error. Much research is done in the past and present to automatically correct syntax errors. Second, there is the validation error. When some structural information is known about the input data of the parser, a validator can check the input or result of the parser to this structural information. Should the validator find something that does not adhere to the structural information, a validation error occurs.

It is this second type of errors, the validation errors, that are covered by the methods described in this paper. The main research question arising from the previous is as follows:

*Can structural information about input that is to be handled by a computer aid the computer in making the right decision about correcting the error?*

To demonstrate the use of structural constraints to correct errors, a few subquestion have to be answered first:

1. **What errors can be corrected with the use of structural information?**
2. **What extra information is needed to correct the identified errors?**
3. **How would a parser use the extra information to correct the identified errors?**

5. METHODS

A traditional parser is usually constructed around a single input stream of either characters or tokens. A set of production rules defined in a grammar match the provided input into a set of terminal symbols and/or other production rules. Suppose we define a language $L$. The complete definition of this grammar is not of concern here, but suppose production rule $A$ is defined as follows:

$$ A \rightarrow B \ C \ D $$

Given this production rule, a string $s$ would be accepted by the parser if it can be produced by applying production rules $B$, $C$ and $D$, in that order.

Providing a parser for $L$ with a set of structural constraints, the parser will have more knowledge on the input it will have to parse. Aside from knowing what is acceptable and not, as defined by the grammar, it can now check for additional constraints defined in the structural information. Formally, it can be said that the parser can combine the grammar along with the additional constraints to narrow down the grammar to a language $L'$. Suppose the production rule for $A$ would still require the production rules $B$, $C'$ and $D$. With the added structural constraints, though, it becomes clear that not the entire range of values accepted by $C$ should be accepted, denoted by $C'$. The production rule for $A$ would now be defined as follows:

$$ A \rightarrow B \ C' \ D $$

When a value is encountered in production rule $C'$ that would not be accepted, the parser can report an error. Not accepting the encountered value would make the parser adhere to the provided structural constraints.

A slightly different mechanism to incorporate the structural constraints would however enable the parser to make an attempt at converting the encountered value to another value that is accepted by production rule $C'$. This mechanism involves creating a conversion function that is able to take an invalid value and convert this to a valid one. This can be visualized in a parse tree, where production rule $A$ is taken as the root of the tree and the conversion of some invalid value $C''$ to the accepted value $C'$ is inserted as a subtree of the production rule $C'$:

\[
\begin{array}{c}
A \\
\vdots \\
C' \\
\vdots \\
a
\end{array}
\]

In this tree, the input for $C''$ and the conversion function $f$ are shown below their symbols. For $C''$ this is an arbitrary terminal symbol $a$, for $f$ this is the empty string, denoted by $\epsilon$. The conversion function does not consume any of the input, it is combined with the result of $C''$ to create a result accepted by $C'$.

The key to this approach is the incorporation of the structural constraints to the grammar of the language and the implementation of the conversion function. The combination of the grammar and the structural constraints enables a parser to detect validation errors as described in section 4. With the use of the conversion function, a parser can attempt to correct the validation error instead of not accepting the input.

To demonstrate this approach, a prototype parser\(^1\) has been created that uses this approach to correct errors in a simple input language. The input language chosen for the parser is the JavaScript Object Notation, or JSON. This rather simple language was introduced in 2001 by Douglas Crockford [3], and standardized in RFC 4627 in 2006 [4].

In section 6 a number of error corrections are described. A number of these error corrections are implemented in the prototype. In section 7, the results of the error correction methods in the prototype implementation are discussed.

6. IMPLEMENTATION

As a demonstration of the proposed error corrections mentioned in this section, the following piece of JSON input is used as an example to perform the correction on:

```json
{
  "name": "John Doe",
  "age": 30,
  "sex": "Male"
}
```

This short JSON example describes a 30-year old male person named John Doe. Should there be some class definition of what this object is, it is not hard to come up with a definition called Person. This definition would hold information about what a person instance should contain. As the prototype parser needs such a definition to work with, a format for specifying this class has been used.

The JSON Schema Proposal [14], proposed by Kris Zyp as a means to specify structural information about a class of JSON objects, provides a usable set of possible statements on the input data. A person specified as in the example above is defined using the schema proposal as such:

\[\text{http://home.student.utwente.nl/m.ugen/JSONParser/}\]

\(^1\)The prototype is available at
This schema defines the properties of the person object and the types of those properties. Using the schema definition while parsing the input data provides several options for correcting errors. Although any difference between the structure of the input and the structural definition of the input could be used to alter the input in order to make it adhere to the structural definition, three methods of error correction are proposed in the following sections.

6.1 Type conversion

Because the types of all values, values within values and members of sequences are defined, a parser using the structural schema to check its input can convert types when a value of the wrong type is encountered. Both simple type conversions and more complex ones become possible. It might be possible for a user to specify the age of John Doe in the earlier example as a string value:

```
{ "name": "John Doe",
  "age": "30",
  "sex": "Male"
}
```

For a human, this reads the same; there’s still 30 in the object specification. A computer would however see this as a different type, storing a string in the resulting object. A parser that knows that the age of a person has to be specified as an integer could convert this string into an integer, if it happens to be a string containing only digits.

In this case, the conversion function \(f\) described in section 5 should be capable of converting a single value \(a\) with an arbitrary type to another single value \(b\) that has the required type.

A more complex type conversion would involve the requirement of a type that cannot be trivially converted from the encountered type. Should for example the name of a person be defined as a sequence of strings, representing a first name, possible middle names and a surname, the structural schema for a person would become as follows:

```
{ "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "integer" },
    "sex": { "type": "string" }
  }
}
```

The schema above requires the value of name to be a sequence and each item in the sequence a string. Logically, the previously provided value of "John Doe" would not be a valid value. Since the parser knows that the "John Doe" it found actually needed to be a sequence of strings, it could split the string into multiple parts by delimiting substrings on whitespace, providing the required sequence of strings.

The above example of splitting strings into a sequence of substrings would of course provide the right type, though knowing on what delimiter to split the string is not easy. In this example, splitting the string on whitespace would yield a correct sequence. However, the name of a person is not a general case, and deciding what delimiter to use when splitting a string is not something that can be done in a way that will always yield the right result. It does however demonstrate how such a complex type conversion could take place.

The more complex case requires a more complex conversion function in order to yield an acceptable value for John Doe’s name. Here, it should be capable of converting single values to and from complex values like sequences or objects.

The difference between the simple and complex methods of type conversion described above is the translation of a single values to complex values and vice versa. In the simple variant, the single value "30" is converted to the differently typed single value 30. With the more complex variant, however, the parser needs to make assumptions on the way the single invalid value is to be converted to a complex value. The same holds for a complex conversion the other way around: the parser needs to assume a certain way of concatenating the multiple values in a sequence or object to a required single value.

6.2 Defaults

Another way for a parser to make use of a provided set of structural information is to create more useful defaults for invalid values. Value types again play a big role here, providing a right answer to a wrong value. Here, too, both a simple and complex way to create a default value can be implemented. In the example of John Doe, a user might have specified John’s age in words:

```
{ "name": "John Doe",
  "age": "Thirty",
  "sex": "Male"
}
```

Similar to the previous section, this remains interpretable for a human; the object specification states that John Doe is thirty. A computer might have the capability to translate written numbers to actual numbers, but keeping in mind that a user might specify any text for a property that expects a number, this will not save the general case. Assuming a computer has no way of knowing what the user meant with "Thirty", there is no way to supply a value that still qualifies as 30. As "Thirty" is not a number, a parser can either drop the property or default to a predefined value. In the case of age, a parser could default to a value of -1, indicating that the supplied value was not valid, though still returning the result with a number as the age for John Doe.

For this example to return a valid result, the conversion function \(f\) needs to have access to default value for either the type required by the structural constraints or the default value for the encountered property.

For a more complex way of providing a default value for an erroneous value, the example of John Doe needs to be extended with some additional info. Suppose the definition for person would include credit card information for that person a follows:

```json
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "integer" },
    "sex": { "type": "string" },
    "card": { "type": "object" },
    "card_number": { "type": "string" }
  }
}
```
The credit card information for a person is incorporated in the person definition using an object definition containing a card name and a card number, assigned to the property \textit{credit} of a person. Should the credit card information of John Doe be missing, a parser at least knows what this information is supposed to look like. Reconstructing a missing piece of information about a person’s credit card information would not magically supply the correct information, but would at least create a result object that adheres to the structural schema. Again, values that indicate an invalid value in the input can be used here.

Filling in missing values is slightly different from a type conversion. Other than converting a supplied value to a different one, using default values allows the insertion of missing properties into the result value.

### 6.3 Deletion and addition

Apart from converting types and filling in missing values, a third method for correcting errors is possible when too little or too much values are found in the input. To demonstrate this, the example of John Doe once again needs to be extended to include a sequence of personal traits. If the structural schema would enforce a maximum or minimum number of traits described, the schema would become as follows:

```json
... "properties": {
  "traits": {
    "type": "array",
    "minItems": 2,
    "maxItems": 3
  }
...}
```

Should a parser now find either one or more than three items in the sequence of personal traits, it could try to make the input adhere to the schema by either inserting default values for missing items or omitting items that exceed the specified maximum. The case of an object rather than a sequence is similar: extraneous properties can be discarded. Missing properties can be handled by the previously mentioned method for filling in default values.

In order for this method to work, the conversion function \textit{\texttt{checkType}} requires knowledge of the bounds of a sequence. In the case of an object specification, the properties of the object should be specified by the structural constraints.

### 7. RESULTS

For the implementation of a prototype, the JavaScript programming language has been used because it is easy to build a simple user interface around a script like this and JavaScript has native support for evaluating JSON data. Being able to evaluate the JSON Schema used as structural information allows for better focus on the prototype itself. Additionally, the MooTools JavaScript framework [7] has been used to allow a smooth class definition and interaction with the demo user interface.

The use of the methods for error correction proposed in section 6 can be implemented in a parser by adding calls to a conversion function \textit{\texttt{convertType}} mentioned in section 5. After for example parsing a value for some property in JSON, a type checking function can be invoked with the parsed value, the required type and any additional structural information that might be of use. Afterwards, the possibly converted value can be checked for extraneous or missing properties.

This simple technique is specified in pseudocode as follows. It is assumed both input data and a structural schema are supplied to the algorithm. Also a table of type conversion functions and default values for types are present.

```pseudocode
begin
  while input is not empty do
    value := parse (input)
    checkType (value, schema)
    cleanUp (value, schema)
    advance input
  od
  where
    proc checkType (value, schema)
      if value.type is not schema.type then
        convertType (value, schema.type)
      fi
    end
    proc convertType (value, type)
      from conversionTable get conversion \from value.type to type
      apply conversion to value
    end
    proc cleanUp (value, schema)
      for property in value do
        if property not in schema then
          remove property
        fi
      od
      for property in schema do
        if property not in value then
          if schema.default is not nil then
            add schema.default to value
          else
            default := from defaults get \default for schema.type
            add default to value
          fi
        fi
      od
    end
end
```

In the prototype, both the functions \textit{\texttt{checkType}} and \textit{\texttt{convertType}} are present. The conversion table in the algorithm has been implemented as a matrix of type options, inserting pieces of conversion code into the matrix where types can easily be converted into one another. The \textit{\texttt{cleanUp}} function in the algorithm is inserted only for the object type, removing extraneous object properties and inserting default values for missing ones.

All three methods for error correction described in section 6 have been implemented in the prototype. They are all shortly described here.

The type conversion method as described in section 6.1
8. CONCLUSIONS

As an answer to the research questions stated in section 4, providing an additional set of structural facts about the input of a parser can aid it in resolving a validation error. The type of errors that are most easily fixed involve value types. The information the parser would need to correct this error is the required types of all the values in the input data. With this information, a type conversion from one type to the other is often relatively easy. For useful default values, the specification of such a default value—as made possible by the JSON Schema default property [14]—would greatly increase its accuracy compared to guessing solely based on the type. Deletions and additions, lastly, require the knowledge of the bounds of a sequence. This would also benefit from a default value, if present.

The implementation of the error correction methods in the prototype were relatively easy. As the type checking functions can all be invoked after the parsing of a value, a programmer will not have to alter the parsing algorithm to incorporate type conversions based on a supplied set of structural constraints. The insertion of default values and deletion of properties too, are invoked after parsing the value the methods are to be applied to.

Although mostly simple error corrections have been implemented in the prototype, more complex methods—like recursively filling a missing property of the object type with default values—require no more room than the simple ones. The difference in implementation being the requirement of more recursive function invocations to find default values for all properties rather than providing a default value from the schema or a simple default value for the required type.

Since JavaScript is a dynamically typed language from which JSON inherits its existence, JSON allows any type for any value. Error correction methods that deal with type conversions are therefore easily done, but perhaps errors also more easily made since the type of a value does have to be explicitly declared.

Both the approach and the prototype feature JSON and the JSON Schema Proposal as parser input and structural information respectively. Despite this, the proposed approach for error correction described in this paper should be applicable to any combination of input and corresponding structural information, depending on the language, the amount of information available in the structural information and the way a human might use it.

9. DISCUSSION

Applications that deal with input specified by humans could benefit from an easy way to correct errors in the input data. The applied correction rules need to fit the needs and nature of the application, however. The possibility of using a method that might make things worse than having erroneous input require careful selection and implementation of the approach described in this paper.

In certain applications, using a predefined set of rules about the input data can be an aid to the person or machine using the application. A service exposing a public API to the world might be able to accept more inputs when simple errors in the provided input can be easily fixed. Because the server-side knows the structural requirements of the input, the additional information about the input needed in the approach described here is available to this application. As long as default values will never lead to actions with side effects—actions other than querying for information—this approach would be safe and beneficial to the user of the service.

Some of the possible actions taken when input data does not meet the requirements set by a structural definition might not be useful in the real world. An amount of information that is larger than a defined maximum amount, for example, does not have to be a problem. If an application expects at most three items in a sequence, it can simply choose to ignore any item after the third. Should the information have to be sent to another machine after parsing on the other hand, reducing the size of the message could be beneficial. In the average case, leaving surplus information in place seems the safer approach.

Another problem arises with the use of default values for properties that were found to have an invalid value. Not providing an application using the parser result with an erroneous value could save the application a runtime error, but could also trick the application into using the defaulted value without checking if it meant anything other than just that value.

All in all, supplying a parser with an additional set of information about its input could save anyone or anything using the result an error, but has its drawbacks. While preventing the need for a certain amount of checks about the result, in some cases other checks might be required to interpret values that mean more than just that value. While the parser that provided the result will not know what to do when John Doe is found to be -1 years old, the application that needed the information about John would need an additional routine to handle that exact event. Not having the exact value the user supplied might even compromise the ability for the user to understand a message by the application stating that the age of John Doe was not supplied correctly.

Bearing the mentioned drawbacks in mind, the main result of supplying a parser with a set of structural facts about its input is moving certain checks and/or conversions from an application that uses the result of the parser to the parser itself. It can be argued that this is not a parser’s job. In certain cases, though, it can be beneficial for the person or machine requiring the parser result.
10. FUTURE WORK

The approach presented in this paper acts purely on invalidations of the input according to the structural definitions provided. The first type of error as described in section 4 is left out. As syntactical error correction in parsers is well-developed and proven to work, it could be possible that the approach described in this paper can be combined with the traditional error correction for better results. By looking at the structural definitions, a parser might be able to make a better informed decision on how to correct an encountered syntax error.

Furthermore, the approach described in this paper will most probably not be complete; there is a lot of information in a structural schema such as the JSON Schema Proposal. This paper only describes the use of a few of the possible restrictions. As an extension to the JSON Schema Proposal, certain properties meant for error corrections could be incorporated in the structural schema. Meaningful default values per type for missing values, value delimiters for sequence concatenations and other extensions could all increase the accuracy of corrected values.

Performing additional actions on top of parsing a certain input is of course slower than only parsing the input. It is possible, however, that performing conversions in the single pass of the parser is faster than parsing the input and using a second application to convert validation errors, requiring a second pass over the input data. A benchmark of a parser with and without the structural error correction methods proposed here would shed more light on this.

A last improvement would be the automatic generation of a parser with the proposed error correction methods already present. Automatic parser generators have been able to create a language recognizer for a while now. Adding a way of specifying how structural constraints corresponding to the grammar of the language might make it possible for the generator to incorporate type conversions or other error corrections based on the provided structural constraints. A generic means of specifying how structural information is defined and what information resides where would be required for such a generator.

11. ACKNOWLEDGEMENTS

I would like to thank Maarten Fokkinga and the peers in my research track for their valuable pointers and guidance during the Bachelor Referaat course.

12. REFERENCES


