Method

This chapter contains a detailed guide through the various steps and components of the Nickel Language Server (NLS). Being written in the same language (Rust[@rust]) as the Nickel interpreter allows NLS to integrate existing components for language analysis. Complementary, NLS is tightly coupled to Nickel's Syntax definition. Hence, in sec. **??** this chapter will first detail parts of the AST that are of particular interest for the LSP and require special handling. Based on that sec. **??** will introduce the main datastructure underlying all higher level LSP interactions and how the AST is transformed into this form. Finally, in sec. **??** the implementation of current LSP features is discussed on the basis of the previously reviewed compontents.

Nickel AST

Nickel's Syntax tree is a single sum type, i.e. an enumeration of node types. Each enumeration variant may refer to child nodes, representing a branch or hold terminal values in which case it is considered a leaf of the tree. Additionally, nodes are parsed and represented, wrapped in another structure that encodes the span of the node and all its potential children.

Basic Elements

The data types of the Nickel language are closely related to JSON. On the leaf level, Nickel defines `Boolean`, `Number`, `String` and `Null`. In addition to that the language implements native support for `Enum` values. Each of these are terminal leafs in the syntax tree.

Completing JSON compatibility, `List` and `Record` constructs are present as well. Records on a syntax level are HashMaps, uniquely associating an identifier with a sub-node.

These data types constitute a static subset of Nickel which allows writing JSON compatible expressions as shown in lst. 0.1.

Listing 0.1 Example of a static Nickel expression

```
{
  list = [ 1, "string", null],
  "enum value" = `Value
}
```

Building on that Nickel also supports variables and functions which make up the majority of the AST stem.

Meta Information

One key feature of Nickel is its gradual typing system [ref again?], which implies that values can be explicitly typed. Complementing type information it is possible to annotate values with contracts and additional meta-data such as documentation, default values and merge priority a special syntax as displayed in lst. 0.2.

Listing 0.2 Example of a static Nickel expression

```
let Contract = {
         foo | Num
             | doc "I am foo",
         hello | Str
               | default = "world"
       }
       | doc "Just an example Contract"
in
let value | #Contract = { foo = 9 }
in value == { foo = 9, hello = "world"}

> true
```

Internally, the addition of annotations wraps the annotated term in a `MetaValue` structure, that is creates an artificial tree node that describes its subtree. Concretely, the expression shown in lst. 0.3 translates to the AST in fig. 1. The green `MetaValue` box is a virtual node generated during parsing and not present in the untyped equivalent.

Listing 0.3 Example of a typed expression

```
let x: Num = 5 in x
```

Static access

Record Shorthand

Nickel supports a shorthand syntax to efficiently define nested records. As a comparison the example in lst. 0.4 uses the shorthand syntax with resolves to the semantically equivalent record defined in lst. 0.5

Listing 0.4 Nickel record using shorthand

```
{
  deeply.nested.record.field = true;
}
```

Yet, on a syntax level different Nickel generates a different representation.

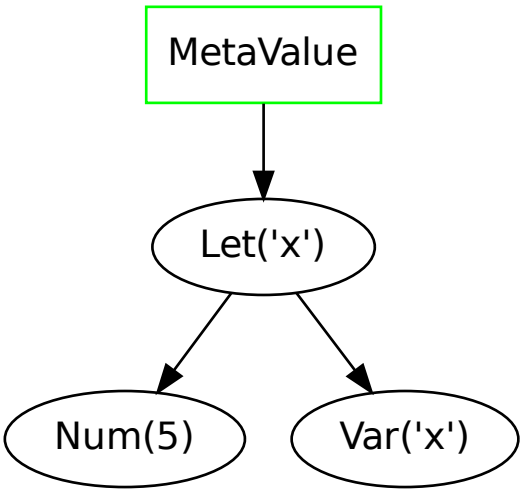

Figure 1: AST of typed expression

Linearization

States

Distinguished Elements

Transfer from AST

Retyping

Post-Processing

Resolving Elements

LSP Server

Diagnostics and Caching

Capabilities

Hover

Completion

Jump to Definition

Show references

Listing 0.5 Nickel record defined explicitly

```
{
  deeply = {
    nested = {
      record = {
        field = true
      }
    }
  }
}
```