eBNF grammar

Based on the sample program we can design the first version of eBNF grammar

Grammar

Based on holistic approach

```
<program>      := <metadata> <decl>+
<metadata>     := (<version> <author>) | (<author> <version>)
<version>      := `version:` `"` <int> `.` <int> `.` <int>
<author>       := `author:` `"` <string> `<` <string> `>`

<decl>         := <func_decl> | <model_decl> | <state_decl> | <enum_decl>
| <struct_decl>

<func_decl>    := <attrs>+ <vis> fn <type_decl> <ident> `(` <params>? `)`
<state_bound>? <st_block>? `{` <func_body `}`
<type_decl>    := <type> | `(` <param> `)`

<attrs>        := `@` `(` <attr_ident> `)` | `@init`
<attr_ident>   := <ident> | ( <attr_ident> `|` )*
<params>       := <param> | (params `,`)*
<param>        := <ident> `:` <type>
<vis>          := `pub` | `view` `(` <ident> <ident> `)`
<state_bound>  := `when` <ident> <ident> <arr> <ident> <ident>?
<func_body>    := (<statement>)*

<st_block>     := `st` <st_body>
<st_body>      := <cond> | `{` <st_list> `}`
<st_list>      := <cond> | (<st_list> `,`)*

<statement>    := <var> | <assign> | <if> | <for> | <foreach> | <return> |
<func_call> | <state_t>
<var>          := let `mut`? <var_ident> (`=` <expr>)?
<var_ident>    := (<ident> | <decon>)
<decon>        := `{` <decon_list> `}`
<decon_list>   := <ident> | (<decond_list> `,` )*

<assign>       := <ident> `=` <expr>
<if>           := `if` `(` <cond> `)` `{` <statement> `}` (`else` `{`
<statement> `}`)?
<foreach>      := `for` `(` `var_ident` `in` (<ident> | <range>) `)` `{`
<statement> `}`
<for>          := `for` `(` <var> `;` <cond> `;` <expr> `)` `{`
<statement> `}`
<range>        := `range` `(` <number> `to` <number> `)`
<cond>         := <expr> <rel> <expr>
<return>       := `return` <expr>
<state_t>      := <ident> `{` <struct_args> `}`
```

```
<struct_args>  := <expr> | (<struct_args> `,`)* | <arg_obj>
<struct_arg>   := <ident> `:` <expr>
<arg_obj>      := `..` <ident>

<model_decl>   := `model` <ident> `{` params `}` <st_block>?

<state_decl>   := `state` <ident> (`from` <ident> <ident>)?  <state_body>
<st_block>?
<state_body>   := `(` <ident> `)` |  `{` params `}`
<enum_decl>    := `enum` `{` <ident>+ `}`
<struct_decl>  := `struct` `{` params `}`

<type>         := `int` | `uint` | `float` | `char` | `string` | `hash`
             | `address` | `()` | `bool` | <set_type> | <list_type> |
<mapping_type>


<set_type>     := `Set` `<` <type> `>`
<list_type>    := `List` `<` <type> `>`
<mapping_type> := `Mapping` `<` <type> <mapping_rel> <type> `>`
<mapping_rel>  := (`>`)? `-` (`/`)? (`>`)? `>`

<char>         := ? UTF-8 char ?
<string>       := `"` <char>* `"`

<digit>        := [0-9]
<number>       := <digit>+

<bool>         := `true` | `false`
<rel>          := `==` | `!=` | `<` | `>` | `<=` | `>=` | `in`

<period>       := `.`
<float>        := <number> <period> <number>?

<func_call>    := <ident> `(` <args>? `)`
<args>         := <expr> | (<args> `,`)*
<func_pipe>    := <expr> (`:>` <func_call>)+


<plus>         := `+`
<minus>        := `-`
<div>          := `/`
<mul>          := `*`
<expr>         := <term> ( (<plus> | <minus>) <term> )*
<term>         := <factor> ( (<mul> | <div>) <factor> )*
<factor>       := <ident> | <constant> | <func_call> | <func_pipe> | `(`
<expr> `)`
<constant>     := <number> | <float> | <bool> | <string>
<ident>        := <char>+
<arr>          := `->`
```

Legend:

- `<ident>` - eBNF element
- `?` - optional element
- `( )` - grouping
- `+` - one or more
- `*` - zero or more
- `ident` - literal token