Zhaba-script syntax🐸

Примечания:

1. Определение синтаксической диаграммы вида: $<a_x>$ ::= b, где x ∈ S и S ∩ Z = S эквивалентна следующей записи:
   $<a_{S0}>$ ::= b
   $<a_{S1}>$ ::= b
   …
   $<a_{Sn}>$ ::= b
2. Использование синтаксической диаграммы вида: $<a_x>$, где x ∈ S и S ∩ Z = S эквивалентна следующей записи: [$<a_{S0}>$|$a_{S1}>$|…|$<a_{Sn}>$]
3. Данный синтаксис не учитывает препроцессинг потому что да.

```
<zhaba-script-program> ::=
   {<op-def0>
   |<type-def>
   |<impl-def>
   |<spacesx><nl>}

<spaces0> ::= ""
<spacesn,n > 0> ::= " "<spacesn-1>
<nl> ::= "\n"
<op-char> ::=
  |"~"|","|"."|"+"|"-"|"*"|"\"
  |"%"|"<"|">"|"="|"^"|"&"|":"
  |"|"|"/"|"!"|"#"|"$"|"@"|"?"
<digit>  ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
<letter> ::=
  |"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"
  |"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|"b"
  |"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"
  |"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"|"_"
<id> ::=
  | <letter>{<letter>|<digit>}
  | <op-char>{<op-char>}

<op-defn> ::= <op-t>[<type>]<id>{<arg>}<blockx + n,x > 0>
<op-t> ::= "fn"|"lop"|"rop"|("bop"<int>)
<arg> ::= <type><id>
```

```
<type-def> ::= "type"<id>{<id>}<type-block_{x,x > 0})>
<type-block_n> ::=
  | <nl><type-block-ln_n>{<type-block-ln_n>}
  | ":"<nl><type-block-content>{<type-block-ln_n>}
  | ":"<type-block-content><nl>
<type-block-ln_n>   ::= <spaces_n><type-block-content><nl>
<type-block-content> ::= <type-literal><id>{<id>}<nl>

<impl-def> ::= "impl"<type-literal><impl-block_{x,x > 0}>
<impl-block_n> ::=
  (":"<op-def_{x,x > 0}>|":"<nl>)<impl-block-item_n>{ impl-block-item_n}
<impl-block-item_n> ::=
  | <op-def_n>
  | <spaces_{x,x >= 0}><nl>

<int> ::= <digit>{<digit>}
<i8-literal>  ::= <int>"i8"
<i16-literal> ::= <int>"i16"
<i32-literal> ::= <int>"i32"
<i64-literal> ::= <int>["i64"|"i"]
<u8-literal>  ::= <int>"u8"
<u16-literal> ::= <int>"u16"
<u32-literal> ::= <int>"u32"
<u64-literal> ::= <int>("u64"|"u")
<float> ::=
  | <digit>{<digit>}"."<digit>{<digit>}
  | "."<digit>{<digit>}
  | <digit>{<digit>}"."
<f32-literal> ::= (<float>|<int>)"f32"
<f64-literal> ::= (<float>|<int>)["f64"|"f"]
<type-literal> ::= <base-type>|<user-type>
<user-type> ::=
<id>["<"<type-literal>{<type-literal>}">")]{"P"|"*"}["R"|"&"]
<base-type> ::=
"int"|"i8"|"i16"|"i32"|"i64"|"u8"|"u16"|"u32"|"u64"|"char"|"str"
<str-literal> ::= "'"{<shielded-char>|<char-not-slash>}"'"
<shielded-char> ::= "\\"<char>
```

```
<literal> ::=
  | <type-literal>
  | <i8-literal>
  | <i16-literal>
  | <i32-literal>
  | <i64-literal>
  | <u8-literal>
  | <u16-literal>
  | <u32-literal>
  | <u64-literal>
  | <f32-literal>
  | <f64-literal>
  | <ch-literal>
  | <str-literal>

<block_n> ::=
  | <nl><block-ln_n>{<block-ln_n>}
  | ":"<nl><block-content>{<block-ln_n>}
  | ":"<block-content><nl>
<block-ln_n> ::= <spaces_n><block-content_n><nl>
<block-content_n> ::=
  |<exp>
  |<if_n>
  |<loop_n>
  |<explicit-var-decl>
  |<ret>

<ret> ::= “<<<”<exp>

<if_n> ::= "?"<exp><block_{n+x, x>0}>{<elif_{n+x, x>0}>}[<else_{n+x, x>0}>]
<elif_n> ::= "|"<exp><block_{n+x, x>0}>
<else_n> ::= "\"<block>
```

Примечание:
В выражениях <optional-comma> может быть вставлена между 2 токенами если левый токен - (“)”|литерал|идентификатор) и правый токен - (“(”|литерал|идентификатор).

```
<optional-comma> ::= [","]
```

```
<loop_n> ::= "@"<loop-exp><block_{n+x,x>0}>
<loop-exp> ::=
  | <loop-exp-while>
  | <loop-exp-foreach>
  | <loop-exp-for>
<loop-exp-while> ::= <exp>
<loop-exp-foreach> ::= <id><optional-comma><exp>
<loop-exp-for> ::= <exp><optional-comma><exp><optional-comma><exp>

<expl-var-decl> ::=
  (<type-literal>|"auto")<expl-var-decl-tuple>
<expl-var-decl-item> ::= <id>|<id>"="<exp>
<expl-var-decl-tuple> ::=
   <expl-var-decl-tuple><optional-comma><expl-var-decl-item>
```

Примечание:
Согласно следующему определению <exp> и <tuple> эквивалентны, но подразумевается использование <exp> когда тип не void и это не перечисление <exp>, и <tuple> когда это некоторое перечисление <exp>.

```
<exp> ::=
  | "("<exp>")"
  | <literal>
  | <tuple>
  | <var>
  | <type-cast>
  | <lop-call>
  | <rop-call>
  | <bop-call>
  | <fn-call>
  | <member-call>
  | <member-access>
  | <implicit-var-decl>

<type-cast> ::= <exp>"as"<type-literal>
<lop-call> ::= <id><tuple>
<roo-call> ::= <tuple><id>
<bop-call> ::= <exp><id><tuple>
```

```
<implicit-var-decl> ::= <id>":="<exp>
<member-access> ::= <exp>"."<id>
<member-call> ::= <exp>"."<id>"("[<tuple>]")"
<fn-call> ::= <id>"("[<tuple>]")"
<tuple> ::=
  | <exp>
  | <tuple><optional-comma><exp>
<var> ::= <id>
```