pinky

A TOY SCRIPTING LANGUAGE TO TEACH ABOUT COMPILERS

REFERENCE

Syntax

Grammar

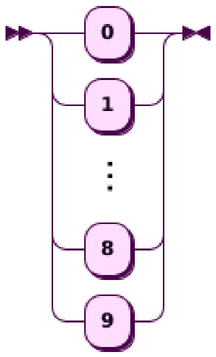
Grammar

Pinky's grammar is designed to be simple and sane to reduce implementation friction and help you get your feet wet when writing your first interpreter or compiler..

Below you'll find the grammar in BNF form paired with a syntax diagram (a.k.a. railroad diagram) for each symbol.

BNF

digit:

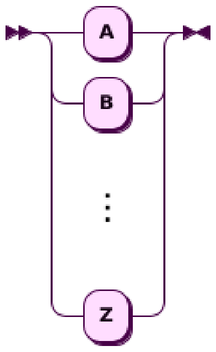


```
digit    ::= '0' | '1' | ... | '9'
```

referenced by:

- alphanum
- integer

upper:

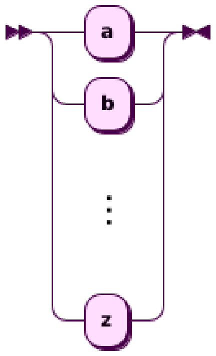


```
upper    ::= 'A' | 'B' | ... | 'Z'
```

referenced by:

- alpha

lower:

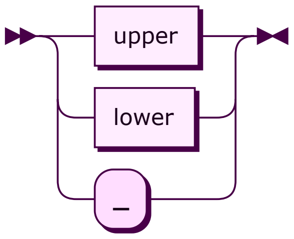


```
lower    ::= 'a' | 'b' | ... | 'z'
```

referenced by:

- alpha

alpha:

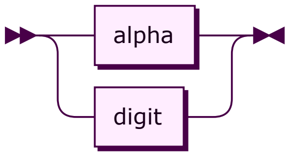


```
alpha    ::= upper
           | lower
           | '_'
```

referenced by:

- alnum
- identifier

alnum:

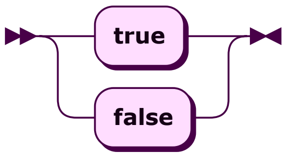


```
alnum    ::= alpha
           | digit
```

referenced by:

- identifier

bool:



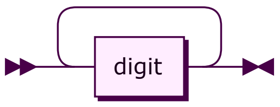
```
bool     ::= 'true'
           | 'false'
```

referenced by:

- literal

integer:

For the grammar, an *integer* is simply a sequence of one or more digits. Negative numbers are implemented using the unary minus operator.

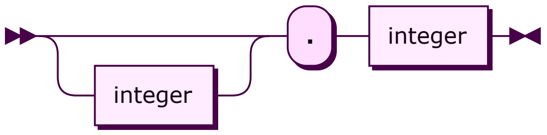


```
integer  ::= digit+
```

referenced by:

- float
- literal

float:

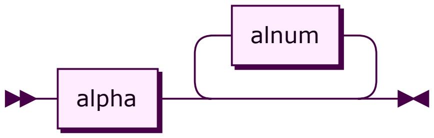


```
float    ::= integer? '.' integer
```

referenced by:

- literal

identifier:

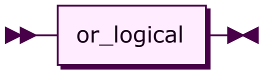


```
identifier
         ::= alpha alnum*
```

referenced by:

- assign
- func_call
- func_name
- params
- primary

expr:

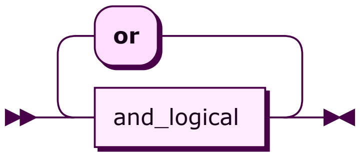


```
expr     ::= or_logical
```

referenced by:

- args
- assign
- expr_stmt
- for_stmt
- grouping
- if_stmt
- print_stmt
- println_stmt
- ret_stmt
- while_stmt

or_logical:

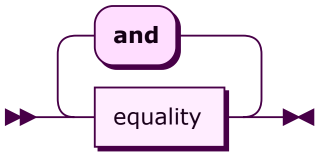


```
or_logical
         ::= and_logical ( 'or' and_logical )*
```

referenced by:

- expr

and_logical:

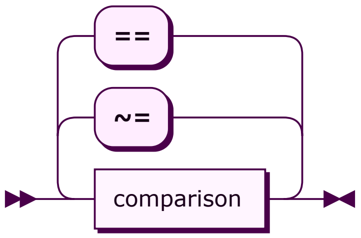


```
and_logical
         ::= equality ( 'and' equality )*
```

referenced by:

- or_logical

equality:

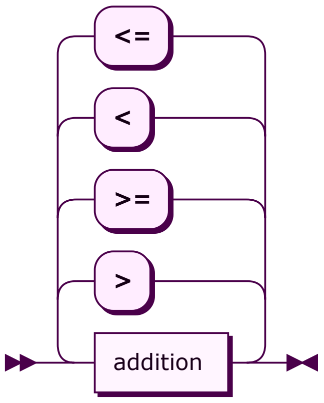


```
equality ::= comparison ( ( '~=' | '==' ) comparison )*
```

referenced by:

- and_logical

comparison:

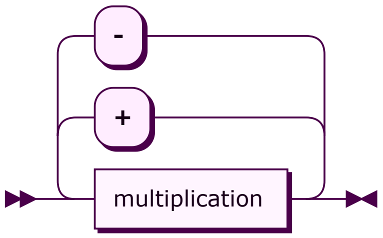


```
comparison
         ::= addition ( ( '>' | '>=' | '<' | '<=' ) addition )*
```

referenced by:

- equality

addition:

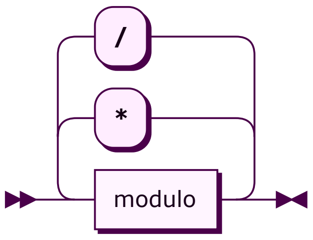


```
addition ::= multiplication ( ( '+' | '-' ) multiplication )*
```

referenced by:

- comparison

multiplication:

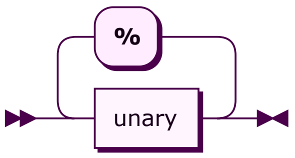


```
multiplication
         ::= modulo ( ( '*' | '/' ) modulo )*
```

referenced by:

- addition

modulo:

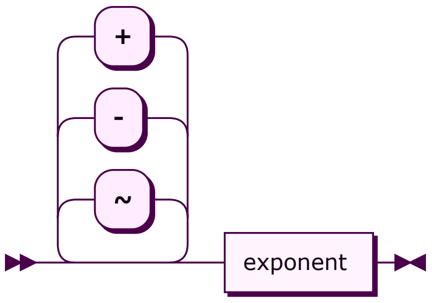


```
modulo   ::= unary ( '%' unary )*
```

referenced by:

- multiplication

unary:

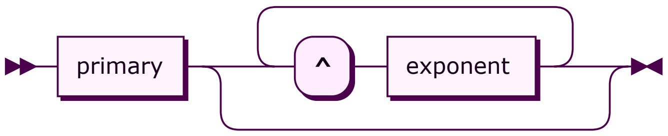


```
unary    ::= ( '~' | '-' | '+' )* exponent
```

referenced by:

- modulo

exponent:

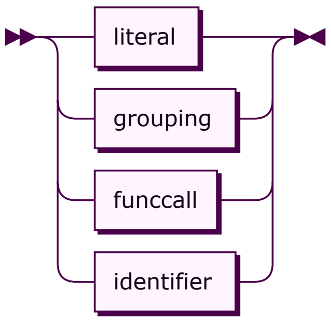


```
exponent ::= primary ( '^' exponent )*
```

referenced by:

- exponent
- unary

primary:

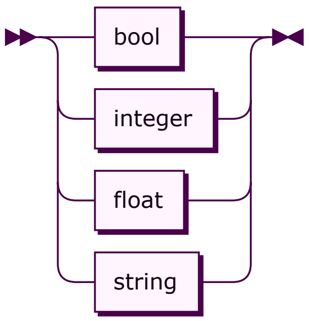


```
primary  ::= literal
           | grouping
           | funccall
           | identifier
```

referenced by:

- exponent

literal:

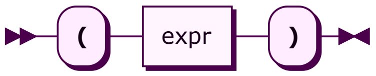


```
literal  ::= bool
           | integer
           | float
           | string
```

referenced by:

- primary

grouping:

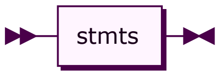


```
grouping ::= '(' expr ')'
```

referenced by:

- primary

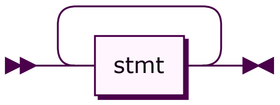
program:



```
program  ::= stmts
```

no references

stmts:

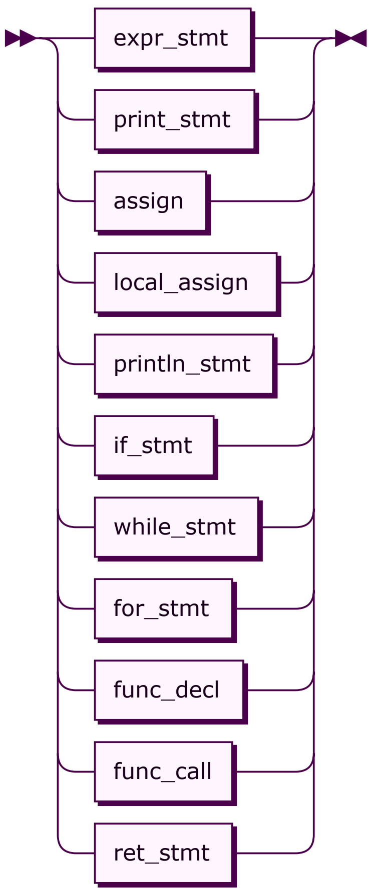


```
stmts    ::= stmt+
```

referenced by:

- for_stmt
- func_decl
- if_stmt
- program
- while_stmt

stmt:



```
stmt     ::= expr_stmt
           | print_stmt
           | assign
           | local_assign
           | println_stmt
           | if_stmt
           | while_stmt
           | for_stmt
           | func_decl
           | func_call
           | ret_stmt
```

referenced by:

- stmts

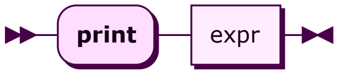
expr_stmt:

```
expr_stmt
         ::= expr
```

referenced by:

- stmt

print_stmt:

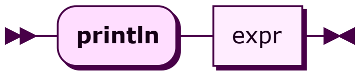


```
print_stmt
         ::= 'print' expr
```

referenced by:

- stmt

println_stmt:

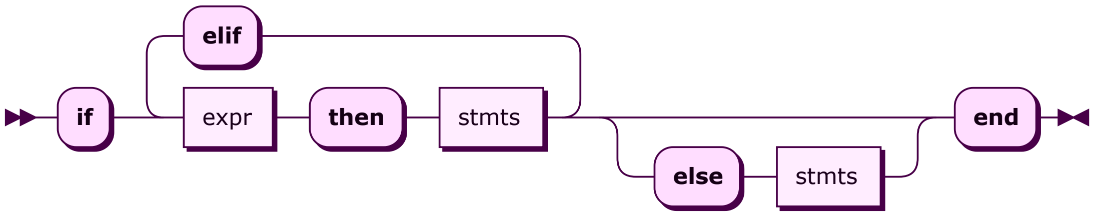


```
println_stmt
         ::= 'println' expr
```

referenced by:

- stmt

if_stmt:

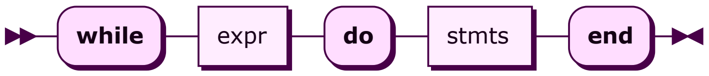


```
if_stmt  ::= 'if' expr 'then' stmts
             ( 'elif' expr 'then' stmts )*
             ( 'else' stmts )? 'end'
```

referenced by:

- stmt

while_stmt:

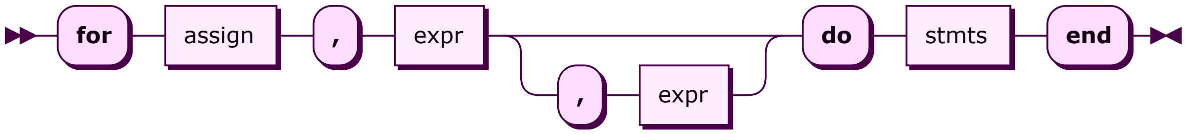


```
while_stmt
         ::= 'while' expr 'do' stmts 'end'
```

referenced by:

- stmt

for_stmt:

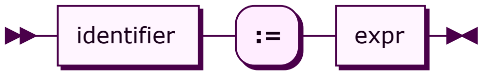


```
for_stmt ::= 'for' assign ',' expr ( ',' expr )? 'do' stmts 'end'
```

referenced by:

- stmt

assign:

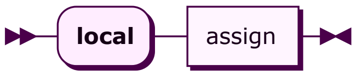


```
assign   ::= identifier ':=' expr
```

referenced by:

- for_stmt
- local_assign
- stmt

local_assign:

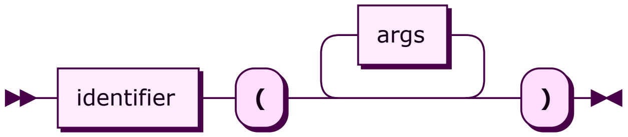


```
local_assign
         ::= 'local' assign
```

referenced by:

- stmt

func_call:

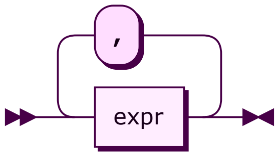


```
func_call
         ::= identifier '(' args* ')'
```

referenced by:

- stmt

args:

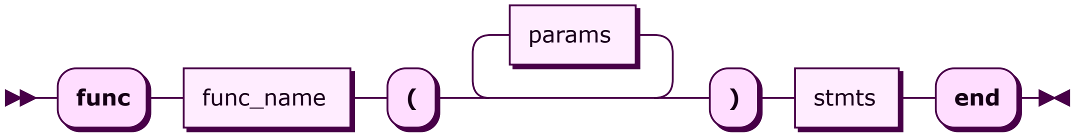


```
args     ::= expr ( ',' expr )*
```

referenced by:

- func_call

func_decl:

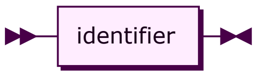


```
func_decl
         ::= 'func' func_name '(' params* ')' stmts 'end'
```

referenced by:

- stmt

func_name:

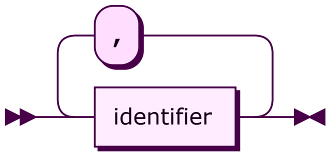


```
func_name
         ::= identifier
```

referenced by:

- func_decl

params:

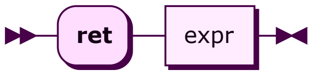


```
params   ::= identifier ( ',' identifier )*
```

referenced by:

- func_decl

ret_stmt:



```
ret_stmt ::= 'ret' expr
```

referenced by:

- stmt