

## Założenia

- Język statycznie, silnie typowany.
- Brak funkcji main - kod wykonywany jest od początku pliku (jak w Python).
- Wywoływana może być jedynie funkcja zdefiniowana wcześniej.
- Zmienne muszą być zainicjalizowane w momencie deklaracji.
- Zmienne są domyślnie mutowalne.
- const przy zmiennej struct odnosi się również do jej pól.
- Możliwe jest definiowanie zagnieżdżonych funkcji. Funkcja zagnieżdżona widoczna jest jedynie z wewnątrz funkcji, w której została zdefiniowana.
- Niedozwolone jest przeciążanie funkcji.

## Przykłady kodu

### Typy danych i operacje

```
bool b = not false or 1 == 1 and true != true;
int i = 3 + 2 * 4.89 as int;
float f = 2 * (2.0 / 2);

print i;
print f;
print b;
```

Output:

```
11
2.0
false
```

### Stałe

```
const float pi = 3.14;
```

### Typ znakowy

```
str w = "Hello\n\"world\"";
print w;

str v = "Hello" + " " + "wo";
```

```
v = v + "rld";  
print v;
```

Output:

```
Hello  
"world"  
Hello world
```

## Komentarze

```
int i = 1; # Single-line comment
```

## Instrukcja warunkowa i pętla

```
int i = 4;  
  
while i > 0 {  
    print i;  
    if i == 3 {  
        i = i - 1;  
    }  
    i = i - 1;  
}  
  
print i;
```

Output:

```
4  
3  
1
```

## Struktura

```
struct Point {  
    int x,  
    int y  
}  
  
Point p = {7, 2};
```

```
p.y = 1;
print p.y;

p.y = p.x;
print p.y;
```

Output:

```
1
7
```

## Funkcje

```
int add_one(int num) {
    return num + 1;
}

void add_one_ref(ref int num) {
    num = num + 1;
}

void multi_parameter(int a, str b, bool c) {

}

int i = 3;
int res = add_one(i);    # Pass by value
print res;

add_one_ref(ref i);      # Pass by reference
print i;
```

Output:

```
4
4
```

## Rekord wariantowy

```
variant Number {
    int,
    float,
```

```

    str
}

void foo(Number n) {
    if n is int {
        int i = 2 * n as int;
        print i;
    }
    if n is float {
        float f = 0.5 * n as float;
        print f;
    }
}

Number a = 2.5 as Number;
foo(a);

a = 5 as Number;
foo(a);

```

Output:

```

1.25
10

```

## Rekord wariantowy ze strukturą

```

struct Point {
    int x,
    int y
}

struct None { }

variant Any {
    Point,
    None
}

Point p = {0, 1};
Any a = p as Any;

int y = (p as Point).y;

```

## Przykrywanie zmiennych

```
void foo() {  
    int i = 5;  
    print i;  
}  
  
int i = 3;  
print i;  
foo();
```

Output:

```
3  
5
```

## Rekursja

```
void count_down_to_zero(int i) {  
    print i;  
    if i == 0 {  
        return;  
    }  
    count_down_to_zero(i - 1);  
}  
  
count_down_to_zero(3);
```

Output:

```
3  
2  
1  
0
```

## Komponenty

- źródło
- analizator leksykalny
- filtr usuwający komentarze
- analizator składniowy
- interpreter

## Konwersja typów

in	int	float	bool	str
int	-	x	x	
float	x	-	x	
bool	x	x	-	
str				-

Wszystkie konwersje muszą być jawne.

## Przykład komunikatu o błędzie

```
const int c = 5;  
c = 10;
```

Output:

```
err: Cannot assign to const variable  
main.rp [8:3]
```

## Rodzaje błędów

- składniowe
- czasu wykonania

## Przekazywanie zmiennych do funkcji

- Domyślnie argumenty przekazywane są przez wartość.
- Poprzedzenie parametru w deklaracji funkcji słowem kluczowym `ref` oznacza przekazywanie go przez referencję. Wtedy też, przy wywołaniu funkcji, odpowiedni argument należy poprzedzić słowem kluczowym `ref`.

# Gramatyka

## Część składniowa

PROGRAM = STMTS

STMTS = { STMT }

STMT =  
| IF\_STMT  
| WHILE\_STMT  
| RET\_STMT  
| PRINT\_STMT  
| CONST\_VAR\_DEF  
| VOID\_FUNC  
| DEF\_OR\_ASGN  
| BUILT\_IN\_DEF  
| STRUCT\_DEF  
| VNT\_DEF

IF\_STMT = **if** EXPR '{' STMTS '}'

WHILE\_STMT = **while** EXPR '{' STMTS '}'

RET\_STMT = **return** [ EXPR ] ';'

PRINT\_STMT = **print** [ EXPR ] ';'

CONST\_VAR\_DEF = **const** TYPE ID ASGN

VOID\_FUNC = **void** ID FUNC\_DEF

DEF\_OR\_ASGN = ID ( FIELD\_ASGN  
| DEF  
| FUNC\_CALL ';' )

FIELD\_ASGN = { '.' ID } ASGN

ASGN = '=' EXPR ';'

BUILT\_IN\_DEF = BUILT\_IN\_TYPE DEF

DEF = ID ( FUNC\_DEF  
| ASGN ) # Definicja zmiennej

FUNC\_DEF =        '(' PARAMS ')' '{' STMTS '}'

PARAMS =        [ PARAM { ',' PARAM } ]

PARAM =        [ **ref** ] TYPE ID

FUNC\_CALL =      '(' ARGS ')'

STRUCT\_DEF =     **struct** ID '{' FIELDS '}'

FIELDS =        [ FIELD { ',' FIELD } ]

FIELD =        TYPE ID

VNT\_DEF =        **variant** ID '{' TYPES '}'

TYPES =        TYPE { ',' TYPE }

EXPR =        DISJ | STRUCT\_INIT

STRUCT\_INIT =    '{' { EXPRS } '}'

EXPRS =        [ EXPR { ',' EXPR } ]

DISJ =        CONJ { **or** CONJ }

CONJ =        EQ { **and** EQ }

EQ =            REL [ '==' REL ]  
                | REL [ '!=' REL ]

REL =            ADD [ '<' ADD ]  
                  | ADD [ '>' ADD ]  
                  | ADD [ '<=' ADD ]  
                  | ADD [ '>=' ADD ]

ADD =            TERM { '+' TERM }  
                  | TERM { '-' TERM }

TERM =            FACTOR { '\*' FACTOR }  
                  | FACTOR { '/' FACTOR }



```

FACTOR =          [ '-' | not ] UNARY

UNARY =           SRC [ as TYPE ]          # Konwersja
                  | SRC [ is TYPE ]        # Sprawdzanie typu

SRC =             CNTNR { '.' ID }          # Pole struktury

CNTNR =           '(' EXPR ')'
                  | CONST
                  | CALL_OR_VAR

CALL_OR_VAR =     ID [ FUNC_CALL ]

ARGS =            [ ARG { ',' ARG } ]

ARG =             [ ref ] EXPR

```

## Część leksykalna

```

BUILT_IN_TYPE = int | float | str | bool

TYPE =           BUILT_IN_TYPE | ID

ID =            LETTER { LETTER | '_' | DIGIT }

LETTER =        [a-zA-Z]

CONST =         FLOAT_CONST
                | INT_CONST
                | BOOL_CONST
                | STR_CONST

FLOAT_CONST =   INT_CONST '.' DIGIT { DIGIT }

INT_CONST =     '0' | NON_ZERO { DIGIT }

NON_ZERO =      [1-9]

DIGIT =         [0-9]

BOOL_CONST =    true | false

```

```
STR_CONST =      ' ' { ANY } ' '
```

```
ANY =          .
```