TKOM-dokumentacja-końcowa

Autor: Łukasz Suchołbiak 325236

Opis języka

Język jest statycznie i silnie typowany. Zmienne są w nim domyślnie niemutowalne, oraz są przekazywane do funkcji przez referencje.

Typy danych

W skład języka wchodzą następujące typy danych:

- **całkowity:** `int`
- **zmiennoprzecinkowy:** `float`
- **logiczny:** `bool`
  - przyjmuje wartość `true` lub `false`
- **typ znakowy:** `string`
- **funkcja**: `function<{typy_argumenów_po_przecinkach}:{typ_zwracacny}>``
  - np `function<int, int: int>`

Operatory

Arytmetyczne

- `*` - mnożenie
- `+` - dodawanie
- `-` - odejmowanie
- `/` - dzielenie
- `-` - unarny minus

Logiczne

- `not` - logiczna negacja
- `and` - logiczny and
- `or` - logiczne or

Porównania

- `==` - równość
- `!=` - nierówność
- `>` - większość
- `<` - mniejszość
- `<=` - mniejsze lub równe
- `>=` większe lub równe

Funkcji wyższego rzędu

- `>>` - bind front
- `&` - kompozycja funkcji

Inne

- `()` - wywołanie funkcji
- `as` - konwersja typu

Tabela priorytetów operatorów

Priorytet	Operator
1	wywołanie funkcji: `()`
2	bind front: `>>`
3	kompozycja funkcji: `&`
4	unarne: `-`, `not`
5	konwersja typu: `as`
6	mnożenie, dzielenie: `*`, `\`
7	dodawanie, odejmowanie: `+`, `-`
8	równość, nierówność: `==`, `!=`
9	porównywanie: `<`, `<=`, `>`, `>=`
10	logiczny and: `and`
11	logiczny or: `or`

Tabela Typów Wspieranych Przez Operatory

Każdy z operatorów ma wybrane typy, które wspiera. Poniższa tabela przedstawia wsparcie operatorów dla typu

operator \ typ	integer	float	boolean	string	function
`*`	TAK	TAK	NIE	NIE	NIE
`+`	TAK	TAK	NIE	TAK	NIE
`-`	TAK	TAK	NIE	NIE	NIE
`/`	TAK	TAK	NIE	NIE	NIE
`==`	TAK	TAK	TAK	TAK	NIE
`!=`	TAK	TAK	TAK	TAK	NIE
`>`	TAK	TAK	TAK	TAK	NIE
`<`	TAK	TAK	TAK	TAK	NIE
`<=`	TAK	TAK	TAK	TAK	NIE
`>=`	TAK	TAK	TAK	TAK	NIE
`not`	NIE	NIE	TAK	NIE	NIE
`and`	NIE	NIE	TAK	NIE	NIE
`or`	NIE	NIE	TAK	NIE	NIE
`()`	NIE	NIE	NIE	NIE	TAK
`&`	NIE	NIE	NIE	NIE	TAK
`>>`	NIE	NIE	NIE	NIE	TAK
`-` unarny	TAK	TAK	NIE	NIE	NIE

Operatory działające na 2 argumentach **wymagają**, aby argumenty te były **tego samego typu**.

Konwersja typów

Wykorzystywana aby przekonwertować pewną wartość na inny typ. Konwersja musi być jawnie zadeklarowana.

Tabela Konwersji Typów

typ źródłowy\typ docelowy	int	float	string	bool	function	Możliwy błąd
int	-	TAK	TAK	TAK	NIE	NIE
float	TAK	-	TAK	TAK	NIE	TAK:int
string	TAK	TAK	-	TAK	NIE	TAK:int,float
bool	TAK	TAK	TAK	-	NIE	NIE
function	NIE	NIE	NIE	NIE	-	NIE

Realizacja konwersji

- z typu `int` na:
  - `float`: konwersja do liczby zmiennoprzecinkowej
  - `string`: zapisanie liczby w postaci ciągu znaków
  - `bool`: wartość `0` rzutuje na `false`, inne `true`
- `float`
  - `int`: konwersja z obcięcie, wartości po przecinku (floor)
    - możliwy błąd przekroczenia zakresu typu int
  - `string`: zapis jeszcze do ustalenia
  - `bool`: wartość `0.0` rzutuje `false`, inne wartości `true`
- `string`
  - `int`: zeskanowanie liczby w formacie dziesiętnym
    - błąd gdy ciąg znaków nie jest poprawną liczbą całkowitą
  - `float`: zeskanowanie liczby zmiennoprzecinkowej
    - błąd gdy ciąg znaków nie jest poprawną liczbą zmiennoprzecinkową
  - `bool`: jeśli ciąg pusty to `false`, inaczej `true`
- `bool`
  - `int`: `true` rzutuje na `1`, `false` na `0`
  - `float`: `true` rzutuje na `0.0`, `false` na `1.0`

Operatory funkcji wyższego rzędu

- `&` - operator składania funkcji
- `>>` - operator bind front
  omówione dokładniej w sekcji o funkcjach

Typ znakowy: `string`

- zapisywany między `""`
  - np. `"Hello World"`
- Do zapisu niektórych znaków wykorzystywany jest escaping
- Dostępne sekwencje escape:
  - `\n` - nowa linia,
  - `\t` - tabulacja,
  - `\"` - cudzysłów,
  - `\\` - backslash

Komentarze

- komentarz zaczyna się od wartości `#`, od tego miejsca trwa do końca linii
  Przykład:

```
4 + 8;# ta część należy już do komentarza
```

Słowa kluczowe języka

- `def`
- `mut`
- `as`
- `return`
- `and`
- `or`
- `not`
- `int`
- `float`
- `string`
- `bool`
- `function`
- `none`
- `true`
- `false`
- `if`
- `else`
- `for`
- `break`
- `continue`

Instrukcje

Pojedyncze instrukcje zakończone są `;`.

Zmienne

Zmienne są domyślnie stałe. Próba modyfikacji zawartości zmiennej zainicjowanej domyślnie zakończy się błędem. Aby zmienna była mutowlana należy ją zainicjowac z użyciem słowa kluczowego `mut`.

Nazywanie zmiennych

- składają się ze znaków ze zbioru - `[a-zA-Z0-9_]`, z których pierwszy znak nie może być cyfrą
- Nazwy zmiennych nie mogą być słowem kluczowym
- Nazwa nie może być zajęta przez funkcję globalną(w tym wbudowaną) oraz przez inną zmienną należącą do tego samego zakresu
  Przykłady:
- poprawne nazwy: `_private`, `_`, `_1`, `numerek1`, `snake_case`, `camelCase`, `PascalCase`, `AAAAAAAA`, `_007`
- niepoprawne nazwy: `true`, `0`, `0klient`, `9_8_7`, `8mila`, `with.dot`, `ca$h`, etc.

Inicjowanie zmiennych

Zmienne niemutowalne

- Zmienna tworzone poprzez `let {nazwa_zmiennej}: {typ_zmiennej} = wartość`
  Przykłady:

```
let x: int = 12;
x = 4; # BŁĄD
```

Tworzenie zmiennej mutowalnej

- Tworzenie zmiennej, której zawartość może być modyfikowana używane jest przez dodanie `mut` przed nazwą zmiennej podczas inicjalizacji
  Przykład:

```
let mut x: int = 12;
x = 3; # OK
```

Scope - zakres widoczności

Określany jest poprzez `{}`. Zmienne zainicjowane w danym zakresie są widoczne na poziomie tego zakresu oraz na poziomie zakresów potomnych (będących wewnątrz tego zakresu).
Kiedy kończy się zakres zmienne zainicjowane w niej są niszczone
Przykład:

```
def main() -> none {
    let a: int = 3;
    {
        let mut b: int = a + 4; # OK, a jest widoczne
    }
    b + 1; # BŁĄD b już nie isnieje
}
```

Przykrywanie - shadowing

Język pozwala na "przykrycie" zmiennej w danym zakresie. Aby przykrywanie było jasno odróżnione od modyfikacji zawartości/przypisanie realizowane jest poprzez definicje zmiennej. Przykrywanie jest możliwe tylko w potomnym zakresie. Po przykryciu odwołania przez identyfikator będą do zmiennej przykrywającej.
Przykład:

```
let a: int = 5;
{
    let mut a: int = 7; # OK, poprawnie przykryta
    a = a + 1;
}# przy opuszczeniu scopu przykrywające a zostaje zniszczone

a + 3; # 8 - w tym scopie widoczne a = 5

let mut a: int = 10; # BŁĄD - próba przykrycia w tym samym scopie
```

Instrukcje warunkowe `if`, `else if`, `else`

Instrukcja warunkowa w języku zrealizowana jest poprzez:

- `if (<warunek>) {<ciało do wykonania jeśl warunek spełniony>}`
- opcjonalne `else if (<warnunek>) {<ciało do wykonania>}`
- opcjonalny `else {<ciało do wykonania żaden z powyższych warunków nie został spełniony>}`
  Wymagane jest aby warunekiem było wyrażenie zwijające się do wartości logicznej - `bool`

Przykłady:

```
let mut a: int = 12;

if (a < 13) {
    a = a + 13;
}
```

```
let animal: string = "cat";

if (animal == "cat"){
    print("meaow");
}else if (animal == "cow") {
    print("MUU");
} else {
    print("bark");
}
```

Pętle

- `for ({zmienna}; {warunek}; {operacja na zmiennej})`
- jako uproszczenie składni zmienną określa się tylko przez nazwę, typ i wartość początkową
  - pomijane słówka let i mut ponieważ wiadomo, że inicjujemy zmienną, oraz, że musi być mutowalna
- `break` - przerywa wykonanie pętli
- `continue` - przechodzi do następnej iteracji
  Przykład:

```
for (i: int = 0; i < 7; i = i + 1) {
    if (i == 2){
        continue;
    }
    print(i);
}
```

Funkcje

Definiowanie funkcji globalnych

Język standardowo posiada wbudowane w zbiorze funkcji globalnych. Aby dołączyć do tego zbioru własną funkcję trzeba to zrobić według schematu:

1. Sygnatura funkcji: `def {nazwa_funkcji}({parametry_funkcji}) -> {zwracany typ, lub none jeśli nic nie zwraca}`
   - schemat standardowego argumentu to `{nazwa_argumentu}: {typ_argumentu}`
2. Następnie scope - `{}`, wewnątrz którego znajduje się kod funkcji
   - wewnątrz tego scopu można odwoływać się do argumentów funkcji
   przykład:

```
def sum_two(a: int, b: int) -> int {
    return a + b;
}
...
```

```
let a: int = 9;
let b: int = sum_two(a, 9); # OK
sum_two(a, b); # OK
sum_two(3, 4); # OK
```

Mutowalne parametry

Aby zainicjować parametr jako multowalny, jego mutowalność musi zostać wskazana przez słówko kluczowe `mut`. W języku zmienne przekazywane są do funkcji przez referencje. Jeśli jako argument wywołania funkcji przekazujemy zmienną, ona również musi być mutowalna. W przypadku r-wartości argument jest przekazywany przez wartość i nie stanowi to problemu - w zakresie funkcji zostaje tworzona zmienna typu mutowalnego o danej wartości.

```
def increment(mut a: int) -> none {
    a = a + 1; # OK - a jest oznaczone jako mutowalne
}
...
let mut x: int = 4;
let y: int = 5;
increment(x); # OK
increment(y); # BŁĄD - x nie jest mutowalne
increment(5 + 2); # OK
```

Schemat przyjmowania jako argumenty funkcji

rodzaj deklracji argumentu\ podane przy wywołaniu	Zmienna niemutowalna (domyślna)	Zmienna mutowalna	R-wartość
Domyślna	przyjmuje	przyjmuje	przyjmuje
Jako mutowalny	nie przyjmuje	przyjmuje	przyjmuje

Redefinicja funkcji

Język **nie zezwala** na redefinicje, ani przeciążanie funkcji. Funkcja w programie może być zdefiniowana tylko raz.
Przykłady:

```
def foo(word: string) -> string {
    return "Foo::" + word;
}

def foo(word: string) -> string { # BŁĄD - redefinicja
    return word + "::ooF";
}
```

```
def make_greetings(name: string) -> string {
    return "Hello " + name + "!";
}

def make_greetings(name1: string, name2: string) -> string { # BŁĄD - redefinicja
    return "Hello " + name1 + " and " + name2";
}
```

Rekursywne wołanie funkcji

Wewnątrz ciała funkcji możliwe są odwołania, oraz wywołania tej funkcji.
Przykład:

```
def factorial(num: int) -> int {
    if (num is 2) {
        return 2;
    }

    return factorial(num - 1) * n;
}
```

Mechanizm funkcji wyższego rzędu

Język zezwala na przekazywanie funkcji do funkcji, zwracanie funkcji z funkcji oraz na zapisanie funkcji do zmiennej. Warunkiem jest zgodność typów funkcji

Przykład:

```
def get_str_printer(s: string) -> function<none:none> {
    return (s) >> print; # funkcją zwraca bind front s na funkcję print
}
...
    let hello_printer: function<none:none> = get_str_printer("hello");
    hello_printer(); # OK, może być zawołana, wyprintuje "hello"
```

Operator składania funkcji

Składa ze sobą 2 funkcje. Warunkiem poprawności jest to, druga w kolejności funkcja przyjmowała tylko 1 argument, oraz aby jego typ był zgodny z wartością zwracana przez pierwszą funkcję.
Typ otrzymanej funkcji: `function<<typy przyjmowane przez f1>:<typ zwracany przez f2>>`

- schemat: `<func1> & <func2>`
  Przykład:

```
def sum3(a: int, b: int, c: int) -> int {
    return a + b + c;
}

def mul2(a: int) -> int {
    return a * 2;
}

let sum3_mul2: function<int, int, int:int> = sum3 & mul2; # sum3_mul2(1, 2, 3) ==
mul2(sum3(1, 2, 3))

let mul4: function<int:int> = mul2 & mul2;
```

Operator bind front

Jego wynikiem jest stworzenie funkcji z ustalonymi pierwszymi argumentami wejściowymi. Aby zachowana była poprawność - liczba przypisywanych argumentów nie może być większa od liczby przyjmowanych parametrów przez funckję.

Bind front realizowany jest zgodnie ze schematem `(<argumenty>) >> <funkcja której argumenty bindujemy>`
Wynikiem operatora jest funkcja przyjmująca pozostałe parametry, które nie zostały zbindowane, zachowująca ten sam typ zwracanej wartości

Przykład:

```
def sum_three(a: int, b: int, c: int) -> int {
    return a + b + c;
}

let add_5_and_2: function<int:int> = (5, 2) >> sum_two;

add_5_and_2(4); # daje 9
```

Biblioteka standardowa języka - funkcje wbudowane

Obsługa strumieni I/O

- Relizowana przez funkcje **print()** oraz **input()**

print

- Typ: `function<string:none>`
- Działanie: Wypisuje zawartość podanego stringa na wyjście standardowe

input

- Typ: `function<none:string>`
- Działanie: Czyta linię podaną przez użytkownika i ją zwraca.

Obsługa stringów

is_int

- Typ: `function<string:bool>`
- Działanie: Sprawdza czy podany string jest liczbą typu całkowitego.

is_float

- Typ: `function<string:bool>`
- Działanie: Sprawcza czy podany string jest liczbą typu zmiennoprzecinkowego

lower

- Typ: `function<string:string>`
- Działanie: Zwraca stringa wejściowego, ze wszystkimi literami zmienionymi na małe

upper

- Typ: `function<string:string>`
- Działanie: Zwraca stringa wejściowego, ze wszystkimi literami zmienionymi na duże

capitalized

- Typ: `function<string:string>`
- Działanie: Zwraca stringa ze zmienioną pierwszą literą na dużą, a resztę na małe

Obsługa typu zmiennoprzecinkowego

round

- Typ: `function<float, int:float>`
- Działanie: Zwraca wartość zaokrągloną do n-tego miejsca po przecinku (n określone przez parametr typu całkowitego)

Wymagania od programu

Program musi posiadać funkcję main o typie: `function<none:int>`. To od niej rozpocznie się interpretacja programu.

Specyfikacja języka - EBNF

```
program = { function_definition };

statement = code_block
          | ( variable_declaration, ";" )
          | ( expression, ";" )
          | ( assignment, ";" )
          | if_statement
          | for_loop
          | ( return_statement, ";")
          | ( break, ";")
          | ( continue, ";" );


function_definition  = function_signature, code_block;
function_signature   = def, identifier, "(", parameter_list, ")", "->",
return_type;

parameter_list       = [ typed_identifier, { ",", typed_identifier } ];
typed_identifier     = [ mut ], identifier, ":", type;

variable_declaration = let, typed_identifier, assign;

assignment       = identifier, assign;
assign           = asgn, expression;

if_statement     = if, condition, code_block, [else, ( code_block | if_statement
)];
condition        = "(", expression, ")";

for_loop         = for, "(", loop_var_decl, ";", expression, ";", assignment,
")", code_block;
loop_var_decl    = identifier, ":", type, assign;

return_statement = return, [ expression ];

code_block       = "{", { statement }, "}";

expression              = logical_or_expression;
logical_or_expression   = logical_and_expression, { or, logical_and_expression };
logical_and_expression  = equality_expression, { and, equality_expression };
equality_expression     = comparison_expression, [ equality_operator,
comparison_expression ];
comparison_expression   = term, [ comparison_operator, term ];
```

```
term                    = factor, { additive_operator, factor };
factor                  = cast, { multiplicative_operator, cast };
cast                    = unary, [ as, type ];
unary                   = function_composition | ( unary_operator,
function_composition );
function_composition    = bind_front, { fcomp, bind_front };
bind_front              = function_call | ( arg_list, bindf, function_call);
function_call           = primary, { arg_list };

primary                 = identifier
                        | literal
                        | "(", expression, ")";

arg_list      = "(", [arguments], ")";
arguments     = argument, {",", argument};
argument      = expression;

return_type   = type | none;
type          = int
              | float
              | bool
              | string
              | function_type;
```

```
literal = literal_integer
        | literal_float
        | literal_bool
        | literal_string;

literal_integer = "0" | ( non_zero_digit, { digit } );
literal_float   = digit, {digit} , ".", digit, { digit };
literal_bool    = "true" | "false";
literal_string  = '"', { escape_sequence | string_char}, '"';

escape_sequence = "\", escape_chars;
escape_chars    = '"', "n", "t", "\";
string_char     = letter | digit | special_chars;

identifier      = ( "_" | letter ), { "_" | letter | digit};

function_type   = function, "<", argument_types, ":", return_type, ">";
argument_types  = ([ mut ], type, {",", [ mut ], type})
                | none;


equality_operator       = eq
                        | neq;

unary_operator          = minus
                        | not;

comparison_operator     = lt
                        | gt
                        | leq
                        | geq;
```

```
multiplicative_operator = mul
                        | div;

additive_operator       = plus
                        | minus;


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";

digit          = "0"
               | non_zero_digit;

non_zero_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

special_chars  = "!" | "#" | "$" | "%" | "&" | "'" | "(" | ")" | "*" | "+" | ","
| "-"
               | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "]"
| "^"
               | "_" | "`" | "{" | "|" | "}" | "~";

not      = "not";
or       = "or";
and      = "and";

minus    = "-";
plus     = "+";
mul      = "*";
div      = "/";
asgn     = "=";

eq       = "==";
neq      = "!=";
lt       = "<";
gt       = ">";
leq      = "<=";
geq      = ">=";

fcomp    = "&";
bindf    = ">>";

def      = "def";
let      = "let";
mut      = "mut";
as       = "as";
if       = "if";
else     = "else";
for      = "for";
```

```
return   = "return";
break    = "break";
continue = "continue";

int      = "int";
float    = "float";
bool     = "bool";
string   = "string";
none     = "none";
function = "function";
```

Przykładowe programy

fibonacci - rekursywne wołanie

```
def nth_fibonacci(n: int) -> int {
    if (n <= 1) {
        return n;
    }

    return nth_fibonacci(n - 1) + nthFibonacci(n - 2);
}

def main() -> int {
    let n: int = 5;

    print("For " + n as string + " sequence number is " + nth_fibonacci(n) as
string);
    return 0;
}
```

Output:

```
For 5 sequence number is 5
```

counter - przykrycie

```
def increment_counter(mut counter: int) -> none {
    counter = counter + 1;
}

def main() -> int {
    let mut counter: int = 0;

    {
        let mut counter: int = 10;
        increment_counter(counter); # counter called with shadowed value
        print("In child scope: " + counter as string);
    }

    print("In parent scope: " + counter as string); # counter = 0
```

```
    return 0;
}
```

Output

```
In child scope: 11
In parent scope: 0
```

kombinacja bind front i składania funkcji

```
def add(a: int, b: int) -> int {
    return a + b;
}

def half(a: int) -> float {
    return a as float / 2.0 ;
}

def main() -> int {
    let composed: function<int:float> = (4) >> add & half;

    print(composed(7) as string);

    return 0;
}
```

Output:

```
5.5
```

lodówka - petla

```
def occupy_fridge() -> none {
    let mut slices_left: int = 15;
    let mut option: string = "brak wyboru";

    print("Otwierasz lodowke i zastanawiasz sie co zjesc");
    print("Zauwazasz ze masz tylko ser");


    for (i: int = 0; i < 20; i = i + 1) {
        if (slices_left < 5) {
            print("siostra: no wez tez cos dla mnie zostaw!");
            break;
        }
        print("\n\n\n");
        print("W lodowce jest " + slices_left as string + " plastrow sera");
        print("Stoisz tu " + i as string + " minut");
        print("Wybor:\n-zjedz\n-czekaj\n-odejdz");
        option = input();

        if (option == "zjedz") {
            print("*jesz ser*");
            slices_left = slices_left - 1;
```

```
        } else if (option == "odejdz") {
            break;
        } else {
            print("czekasz, a domownicy się denerwują");
        }
    }

    print("odchodzisz i zamykasz za soba lodowke");
}

def main () -> int {
    occupy_fridge();
    return 0;
}
```

until yes

```
def main() -> int {
    let mut cont: bool = true;
    let mut postfix: string = "st";

    # dla przykladu, ze warunek nie musi zawierac i
    for (i: int = 1; cont; i = i + 1) {
        if (i == 2) {
            postfix = "nd";
        } else if ( i == 3) {
            postfix = "rd";
        } else if (i == 4) {
            postfix = "th";
        }
        print("asking " + i as string + postfix + " time");
        print("yes or no?");

        if (input() == "yes") {
            cont = false;
        }
    }

    return 0;
}
```

Interpreter

Sposób uruchomienia interpretera

Program przyjmie na wejście plik źródłowy lub tekst z wejścia standardowego w zależności od wybranych flag
Dla plików źródłowych:

```
./tkm_interpreter <plik źródłowy> <opcje>
```

Dostępne opcje

- `-h` `--help` - domyślna opcja, wyświetla dostępne opcje programu

- `-s` `--stdin` - czytanie programu ze standardowego wejścia
- `-v` `--verbose` - zwiększona ilość informacji w trakcie wykonania - informacje o zbudowanych tokenach oraz wyświetlenie zbudowanej struktury programu
  Przykład uruchomienia z pliku źródłowego:

```
./tkm_interpreter forloop.tkm -v
```

Przykład uruchomienia z wejścia standardowego:

```
./tkm_interpreter -s -v <<EOF
> def main() -> int {
>     print("hello world");
>
>     return 0;
> }
> EOF
```

Implementacja

Source Handler

Leniwie - na żądanie `Lexera` czyta znak ze źródła. Zwraca ten znak wraz z pozycją.

Lexer

Leniwie - na żądanie `Parsera` buduje Token i go zwraca. Przy wystąpnieniu błędu leksykalnego rzuca wyjątek.

Parser

Buduje strukturę/reprezentację programu pobierając kolejne tokeny od `Lexera`. Przy wystąpieniu błędu składniowego rzuca odpowiedni wyjątek.

Interpreter

Implementuje wzorzec wizytatora. Interpretuje program zgodnie z jego przepływem. W przypadku wystąpienia błędów semantycznych rzuca odpowiednim wyjątkiem.

Exceptions

Zawiera zdefiniowane wyjątki programu. Wyjątki dzielą się na 4 typy:

- `LexerException` - wyjątki rzucone w wyniku błędu leksykalnego
- `ParserException` - wyjątki rzucone w wyniku błędu składniowego
- `InterpreterException` - wyjątki rzucone na skutek błędu semantycznego
- `ImplementationError` - wyjątki rzucone na skutek błędów implementacyjnych

SafeExec

Namespace udostępniający funkcję `run_safe` w której uruchamiany jest cały program. W przypadku wystąpienia błędu wypisuje jego komunikat i kończy wykonanie.

CliApp

Uruchamia program na podstawie opcji wywołania.

Obsługa błędów

Na każdym z etapów kompilacji może zostać zgłoszona informacja o błędzie. W takiej sytuacji interpretacja zostaje przerwana, a informacja o błędzie zostaje wyświetlona użytkownikowi.

Informacja o błędzie

- linia z informacją o błędzie: `[error] {informacja o błędzie} at: [{LINIA}:{KOLUMNA}]`
- linia z błędem

Przykłady błędów

Dla uproszczenia przykładów, zakładamy że instrukcje znajdują się wewnątrz funkcji

Próba modyfikacji wartości zmiennej niemutowalnej

```
let a: int = 4;
a = 10;
```

Informacja o błędzie:

```
[error] RuntimeError: Cannot assign to immutable: a at: [3:5]
```

Użycie operatora przypisania do literału

```
4 = 10;
```

Informacja o błędzie:

```
[error] SyntaxError: Invalid assignment target at: [2:7]
```

Użycie niepoprawnych znaków w identyfikatorze

```
let ca$h: int = 32;
```

Informacja o błędzie:

```
[error] LexicalError: Unexpected char:$ at: [2:11]
```

Nazwa zmiennej jest słowem kluczowym

```
let if: float = 4.01;
```

Informacja o błędzie:

```
[error] SyntaxError: Expected identifier with specified type at: [1:5]
```

Wołanie funkcji podając argument typu innego niż przyjmowany

```
def my_func(a: int) -> int {
    return a + 1;
}
...
    my_func("4");
```

Informacja o błędzie:

```
[error] RuntimeError: FunctionCall argument types do not match param types.
Arguments types: (string), Param types: (int) at: [6:5]
```

Wołanie funkcji ze zbyt wieloma argumentami

```
def my_func(a: int) -> int {
    return a + 1;
}

my_func(4, 5, 6)
```

Informacja o błędzie:

```
[error] RuntimeError: FunctionCall argument types do not match param types.
Arguments types: (int, int, int), Param types: (int) at: [6:5]
```

Użycie operatora bind front bez podania funkcji

```
let fun: function<int:int> = (4, 3) >>;
```

Informacja o błędzie:

```
[error] SyntaxError: Expected bind target at: [1:39]
```

Redefinicja funkcji

```
def my_add(a: int, b: int) -> int {
    return a + b;
}

def my_add(a: int, b: int, c: int) -> int {
    return a + b + c;
}
```

Informacja o błędzie:

```
[error] RuntimeError: my_add is already defined and cannot be redefined here.
Redefinition at: [5:1]
```

Opis testowania

Analizator leksykalny

Testy budowania pojedynczych tokenów, oraz budowanie tokenów dla przykładowego programu wraz ze sprawdzaniem zgodności pozycji zbudowanych tokenów

Analizator składniowy

Testy z wykorzystaniem udawanego leksera z weryfikacją, czy zbudowana struktura przez parser jest taka, jak oczekiwana(z wykorzystaniem wizytatora do testów zczytującego strukturę programu).

Interpreter

Testy interpretacji programu na podstawie tekstowego strumienia imitującego źródło przy wykorzystaniu `Lexera` oraz `Parsera`. Weryfikacja poprawności wyniku programu oraz przebadanie, czy sytuacje błędne skutkują rzuceniem odpowiednich wyjątków (Testy na około 2000 linii).
Testy poszczególnych metod wykorzystywanych przez interpreter.