Autor: Łukasz Suchołbiak 325236

Temat projektu

Tematem projektu jest stworzenie:

1. Języka o statycznym i silnym typowaniu, w którym zmienne są domyślnie niemutowalne, oraz przekazywane do funkcji przez referencję. Język ma zapewniać wsparcie dla mechanizmu funkcji wyższego rzędu, oraz posiadać 2 wbudowane operatory dla tego mechanizmu.
2. Interpretera obsługującego ten język, obsługującego 2 źródła.

Założenia języka

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

Logiczne

- `not` - negacja
- `and` - logiczny and
- `or` - logiczne or
- `==` - równość
- `!=` - nierówność
- `>` - większość
- `<` - mniejszość
- `<=` - mniejsze lub równe
- `>=` większe lub równe

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
`*`	TAK	TAK	NIE	NIE
`+`	TAK	TAK	NIE	TAK
`-`	TAK	TAK	NIE	NIE
`/`	TAK	TAK	NIE	NIE
`==`	TAK	TAK	TAK	TAK
`!=`	TAK	TAK	TAK	TAK
`>`	TAK	TAK	NIE	TAK
`<`	TAK	TAK	NIE	TAK
`<=`	TAK	TAK	NIE	TAK
`>=`	TAK	TAK	NIE	TAK
`not`	NIE	NIE	TAK	NIE
`and`	NIE	NIE	TAK	NIE
`or`	NIE	NIE	TAK	NIE

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

Konwersja typów

Wykorzystywana gdy zrzutować pewną wartość na inny typ. Aby nastąpiła konwersja typu, język wymaga aby programista jawnie ją zadeklarował.

Operator konwersji typu - `as`

- Szablon wykorzystania: `{wartość} as {typ}` Przykłady:
- `4 as float`
- `"40000" as int`
- `70 as string` - rezultat: `"70"`

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,boolean
bool	TAK	TAK	TAK	-	NIE	NIE
function	NIE	NIE	NIE	NIE	-	

Realizacja konwersji

- int
  - -> float: prosta konwersja
  - -> string: zapisanie liczby w postaci ciągu znaków (np `123 as string == "123"`)
  - -> bool: wartość `0` rzutuje na `false`, inaczej `true`
- float
  - -> int: 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 floata
    - błąd gdy ciąg znaków nie jest poprawną liczbą zmiennoprzecinkową
  - -> bool:
    - błąd gdy ciąg znaków jest inny niż "true" lub "false"
- bool
  - -> int: `true` rzutuje na `1`, `false` na `0`
  - -> float: `true` rzutuje na `0.0`, `false` na `1.0`
  - -> string: `true` rzutuje na `"true"`, `false` rzutuje na `"false"`

Operatory funkcji wyższego rzędu

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

String

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

Komentarze

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

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

W dalszej części komentarze będą używane do opisu konktretnych sytuacji w kodzie lub oznaczenia

Zmienne

Nazywanie zmiennych

- składają się ze znaków ze zbioru - `[a-zA-Z0-9_]`
  - ale pierwszy znak nie może być cyfrą
- nie mogą być słowem kluczowym Przykłady
- poprawne nazwy: `_private`, `_`, `_1`, `numerek1`, `snake_case`, `camelCase`, `PascalCase`, `AAAAAAAA`, `_007`
- niepoprawne nazwy: `0`, `0klient`, `9_8_7`, `8mila`, `with.dot`, `ca$h`, etc.

Inicjowanie zmiennych

- Zmienna tworzone poprzez `let {nazwa_zmiennej}: {typ_zmiennej} = wartość`
- Domyślnie tworzone zmienne są niemutowalne, próba przypisania wartości do zmiennej spowoduje błąd 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

- `{}` określa zakres widoczności/scope
- Zmienne zainicjowane w danym zakresie są widoczne:
  - na poziomie tego zakresu
  - 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

- pozwala na użycie tej samej nazwy zmiennej
- przykrywanie jest możliwe tylko w innym scopie i jest zreazizowane poprzez użycie składni takiej samej jak przy inicjowaniu zmiennej (żeby nie pomylić go z przypisaniem nowej wartości) 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

- Zrealizowane standardowo przy użyciu `if ({warunek}) {scope i kod wewnątrz} else {scope i kod wewnątrz}`
- zapewnienie `else` jest opcjonalne
- jako warunek może być podany tylko statement zwijający się do wartości logicznej - `bool`
-

Przykłady:

```
let mut a: int = 12;

if (a < 13) {
```

```
    a = a + 13;
}
```

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

if (animal is "cat"){
    print("meaow");
} else {
    print("bark");
}
```

Łączenie instrukcji warunkowych

- zrealizowane prosto przez dodanie `if ({warunek})` po else
    - if nie musi być wewnątz `{}` Przykład:

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

if (animal is "cat"){
    print("meaow");
} else if (animal is "dog"){
    print("bark");
} else if (animal is "cow"){
    print("muuu");
} else {
    print("*silence*");
}
```

Pętle

- standardowe 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) {
    print(i);
}
```

- Jednak nie można modyfikować wartości zmiennej na której operuje pętla wewnątrz jej ciała

- pomimo, że iterujemy po wartości, więc będzie modyfikowana, to wewnątrz ciała pętli traktowana jest jako zmienna niemutowalna Przykłady:

```
for (i: int = 0; i < 7; i = i + 1) {
    i = i + 1; # BŁĄD
}
```

Funkcje

Definiowanie funkcji

- w definicji funkcji określana jest ilość, oraz typy przyjmowanych argumentów, oraz wartość zwracana przez tę funkcję
- schemat definicji:
  1. `def {nazwa_funkcji}({argumenty_funkcji}) -> {zwracany typ}`
     - schemat standardowego argumentu to `{nazwa_argumentu}: {typ_argumentu}`
  2. a 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 argumenty

- do argumentów zadeklarowanych w powyższy sposób nie można stosować operatora przypisanie `=` -> nie można modyfikować ich wartości

```
def increment(a: int) -> none {
    a = a + 1; # BŁĄD
}
```

- Aby to było możliwe mutowalność musi zostać wskazana poprzez dodanie `mut` przed nazwę argumentu
  - jednak ogranicza to elementy, które mogą zostać podane jako argument tej funkcji do zmiennych mutowalnych

```
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, ale
increment(y); # BŁĄD - x nie jest mutowalne
increment(5); # BŁĄD - literał nie może wejść jako argument mut
```

Schemat przyjmowania jako argumenty funkcji

rodzaj deklracji argumentu\ podane przy wywołaniu	Zmienna (domyślna)	Zmienna mutowalna	Literał
Domyślna	przyjmuje	przyjmuje	przyjmuje
Jako mutowalny	nie przyjmuje	przyjmuje	nie przyjmuje

Redefinicja funkcji

Język **nie zezwala** na redefinicję funkcji. Funkcja w programie może być zdefiniowana tylko raz.

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

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

Przeciążanie funkcji

Język **nie zezwala na przeciążanie funkcji**, próba przeciążenia zostanie potraktowana jako redefinicja funkcji i będzie skutkowała zgłoszeniem błędu o redefinicji funkcji Przykład:

```
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

- możliwe wykonywanie przez wywołanie własnej funkcji wewnątrz ciała 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
- musi być zdefiniowany typ funkcji wejściowej: `function<{typy_zmiennych_po_przecinkach}:{typ_zwracanej_wartości}>`
    - np. `function<int, int, bool:int>`

Operator składania funkcji

- składa ze sobą dwie funkcje - wyjście z pierwszej wchodzi na wejście drugiej
    - może być wołany też dla tej samej funkcji
- druga funkcja musi przyjmować tylko 1 argument
- wartość zwracana z funkcji 1 musi być tego samego typu, co argument wejściowy funkcji 2
- typ otrzymanej funkcji: `function<{typy przyjmowane przez f1}:{typ zwracany przez f2}`
- schemat: `{nazwa pierwszej funkcji} & {nazwa drugiej funkcji}` 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

- powoduje stworzenie funkcji z ustalonymi pierwszymi argumentami wejściowymi

- relizowany za pomocą `({argumenty}) >> {nazwa_funkcji_do_ktorej_przywiązuje_argumenty}` 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: Przyjmuje wartość podaną przez użytkownika z wejścia standardowego, czyta do wystąpienia pierwszego newline'a

Obsługa stringów

is_int

- Typ: `function<string:bool>`
- Działanie: Sprawdza czy podany string jest liczbą typu całkowitego, obsługuje również czytanie liczby ujemnej.

is_float

- Typ: `function<string:bool>`
- Działanie: Sprawcza czy podany string jest liczbą typu zmiennoprzecinkowego
- **czy ma obsługiwać notację matematyczną????** - chyba tak

is_bool

- Typ: `function<string:bool>`
- Działanie: Sprawdza czy podany string jest wartością logiczną

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

capitalize

- Typ: `function<string:string>`
- Działanie: Zmienia pierwszą literę stringa na dużą, a resztę na małe

EBNF

```
program = { statement };

statement = function_definition
          | 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     = identifier, ":", type;

variable_declaration = let, [ mut ], 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    = typed_identifier, 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                   = primary | ( unary_operator, primary );
primary                 = identifier
                        | literal
                        | "(", expression, ")"
                        | function_call
                        | function_composition
                        | bind_front;

function_composition    = identifier, { fcomp, identifier };
bind_front              = arg_list, bindf, identifier;

function_call = identifier, arg_list;
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  = (type, {",", 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";
```

Analiza wymagań

Wymagania języka

1. **Język jest statycznie typowany** - typ zmiennej jest określany podczas tworzenia tej zmiennej i jest niezmienny przez cały czas jej życia.
2. **Język jest silnie typowany** - typy zmiennych są ściśle określone i użycie zmiennej w kontekście innego typu wymaga skonwertowania jej do innego typu. Operacje są zdefiniowane dla konkretnych zestawów typów
3. **Zmienne w języku są domyślnie niemutowalne** - wartość można im przypisać wyłącznie raz. Zmienna musi być oznaczona jako mutowalna, aby wielokrotnie przypisywać do niej wartości.
4. **Wsparcie dla mechanizmu funkcji wyższego rzędu (HOF)** - funkcja jest typem danych i może być przekazywana jako argument wywołania lub wartosc zwracana przez funkcję.
5. **Język posiada 2 wbudowane operatory dla mechanizmu HOF** - wybrane operatory to **operator składania funkcji**(function composition), oraz **operator bind front**

Wymagania interpretera

Analizator leksykalny

1. Obsługa 2 typów źródeł - plik oraz ciąg znaków.
2. Leniwa tokenizacja - tokeny generowane na żądanie.
3. Obsługa escapingu w ciągach tekstowych.
4. Parametryzowalne ograniczenia na długość identyfikatorów, oraz stałych tekstowych

Analizator składniowy

1. Weryfikacja poprawności składniowej.
2. Budowa drzewa składniowego.
3. Możliwość zrzucania budowanej struktury drzewa na wyjście konsoli.

Analizator semantyczny

1. Weryfikacja poprawności semantycznej drzewa składniowego.

Interpreter

1. Funkcje wbudowane traktowane tak samo jak definiowane przez użytkownika.

Koncepcja realizacji

Rozróżniane tokeny

Słowa kluczowe: `let`, `mut`, `def`, `as`, `if`, `else`, `return`, `for break`, `continue`, `int`, `float`, `bool`, `string`, `none`, `function` **Operatory**: `+`, `-`, `/`, `*`, `not`, `or`, `and`, `==`, `!=`, `>`, `<`, `<=`, `>=`, `&`, `>>`, `=` **Separatory i delimetry:** `,` , `;`, `{`, `}`, `(`, `)`, `"` **Identyfikatory**: `identifier` - nazwy zmiennych lub funkcji niezaczynające się cyfrą składające się ze znaków alfabetu łacińskiego, podłogi, oraz cyfr **Literały:** `literal_float`, `literal_int`, `literal_bool`, `literal_string`

Struktura projektu

- Moduł interface
  - Przyjmuje źródło kodu tekstowego w postacie strumienia pliku lub ciągu znaków
  - Leniwie zwraca kolejne znaki
  - Zezwala na podejrzenie następnego znaku
  - Pilnuje pozycji w tekście
- Moduł leksera
  - Leniwie tworzy kolejne tokeny przy pomocy modułu interface
- Moduł parsera
  - Pobiera tokeny od modułu leksera
  - Weryfikuje poprawność składniową
  - Tworzy drzewo składniowe
- Moduł analizatora semantycznego
  - Przyjmuje drzewo składniowe od modułu parsera
  - Weryfikuje poprawnośc semantyczną struktury składniowej
- Moduł obsługi błędów:
  - Klasyfikacja błędów
  - Przechowywanie informacji o błędach (rodzaj błedu, miejsce wystąpienia)

    - Przerwanie programu w przypadku błędu krytycznego
    - Wyświetlenie informacji o błędach.

Testowanie

- **testy jednostkowe** - testowanie czy pojedyncze komponenty zwracają wyniki zgodne z oczekiwaniami
- **testy integracyjne** - testowanie całego ciągu transformacji lub jego segmentów

Obsługa błędów

Na każdym z etapów kompilacji może zostać zgłoszona informacja o błędzie. W przypadku wystąpienia błędów zastosowane zostaną poprawki i program będzie kontynuował działanie w celu zwrócenia większej ilości informacji diagnostycznych.

Informacja o błędzie

- linia z informacją o błędzie: `Error:{plik-jeśli czytamy z pliku} {LINIA}:{KOLUMNA} {OPIS BŁĘDU}`
- linia z błędem
- podkreślenie wystąpienia błędu poprzez `^`

Przy błędach niezapewnionych wartości oczekiwanych może zostać dostawiona wartość Dostawiane wartości:

- `int` - `1`
- float - `1.0`
- string - `""`
- bool - `true` W przypadku niezapewnienia wartości typu funkcji, instrukcja jest pomijana.

Przykłady błędów

dla uproszczenia przykładów, zakładam że plik wejściowy ma nazwę `main.tkom`

Próba modyfikacji wartości zmiennej niemutowalnej

```
let a = 4;
a = 10;
```

Informacja o błędzie:

```
a = 10;
  ^
Error:main.tkom 2:3 Assignment to immutable variable
```

Obsługa: Pominięcie instrukcji

Użycie operatora przypisania do literału

```
4 = 10;
```

Informacja o błędzie:

```
400 = 10
    ^
Error:main.tkom 1:5 Assigment to literal
```

Obsługa: Pominięcie instrukcji

Użycie niepoprawnych znaków w identyfikatorze

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

Informacja o błędzie:

```
let ca$h: int = 32;
      ^
Error:main.tkom 1:5 Illegal symol in identifier
```

Obsługa: Zastąpienie nielegalnego symbolu przez _

Nazwa zmiennej jest słowem kluczowym

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

Informacja o błędzie:

```
let if: float = 4.01;
    ^
Error:main.tkom 1:5 in identifier
```

Obsługa: Zastąpienie każdego znaku słowa kluczowego przez _

Wołanie funkcji podając argument typu innego niż przyjmowany

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

my_func("4");
```

Informacja o błędzie:

```
my_func("4");
        ^
Error:main.tkom 5:9 Function call with illegal argument type
```

Obsługa: Dodanie konwersji typu, `"4"` -> `"4" as int`

Wołanie funkcji ze zbyt wieloma argumentami

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

my_func(4, 5, 6)
```

Informacja o błędzie:

```
my_func(4, 5, 6)
       ^
Error:main.tkom 5:8 Function call with too many arguments
```

Obsługa: Usunięcie dodatkowych elementów z listy argumentów

Użycie operatora bind front bez podania funkcji

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

Informacja o błędzie:

```
let fun: function<int:int> = (4, 3) >>;
                                    ^
Error:main.tkom 1:39 Bind front operator without function identifier
```

Obsługa: Pominięcie instrukcji

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:

```
def my_add(a: int, b: int, c: int) -> int {
    ^
Error:main.tkom 5:5 Function redefinition
```

Obsługa: dodanie `_` na końcu identyfikatora

Sposób uruchomienia

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

```
tkm <pliki źródłowe> <flagi>
```

Flaga `-o` pozwoli na okreslenie nazwy pliku wyjściowego

Dla tekstu z wejścia standardowego:

```
tkm - <flagi>
```

Przykłady

fibonacci

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

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

```
}

let n: int = 5;

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

Output:

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

counter - przykrycie

```
let mut counter: int = 0;

def increment_counter() -> none {
    counter = counter + 1;
}

{
    let mut counter: int = 10;
    increment_counter();
    print("In child scope: " + counter as string); # counter = 10 - funkcja
ma dostęp tylko do globalnego countera
}

print("In parent scope: " + counter as string); # counter = 1
```

Output

```
In child scope: 10
In parent scope: 1
```

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;
}

let composed: function<int:float> = (4) >> add & half;
```

```
print(composed(7))
```

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("W lodowce jest " + slices_left + " plastrow sera");
        print("Stoisz tu " + i as string + " minut");
        print("Wybor:\n- zjedz\n-czekaj\nodejdz");
        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");
}

occupy_fridge()
```

until yes

```
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;
    }
}
```