Język na JPP

K.Mykitiuk

April 2019

1 Opis języka

Język imperatywny ze statyczną kontrolą typów. Funckje i typy są obywatelami pierwszej kategorii. Inspirowany Pytonem i Lispem. Operatory są lewostronne o priorytetach

1. ()
2. * / %
3. + -
4. == != <><= >= !== ===
5. &&
6. ||
7. =

Błędy wykanania są obsługiwane przez interpreter.

2 Elementy języka

Oczekiwana punktacja za poszczególne elementy

1. Program imperatywny na 15 pkt z pętlą for z constem oraz whilem (+15pkt)
2. Część na 20pkt (+5pkt)
3. Statyczne typowanie (+4pkt)

4. dowolnie zagnieżdżone definicje funkcji (+2pkt)
5. pythonowe krotki z rozpakowywaniem (+2pkt)
6. funcke jako argumenty (+2pkt)
7. funckje jako zwracane obiekty (+2pkt)
8. dedukcja typu z auto (+1pkt)
9. type_of i typy jako obywatel pierwszej kategorii (+4 pkt)
   (a) deklaracje z użyciem type_of (+2pkt)
   (b) Arytmetyka na type_of (+1pkt)
   (c) Wołanie typów (+1pkt)
10. binidng (+2pkt)

Razem 38pkt... funkcjonalność dodatkową wykonam w miarę możliwości. Elemetny będę dodawał w kolejności z listy.

3 Fragmenty kodu

3.1 Komentarze

Komentarze jednoliniowe zaczynają sie # Komentarze wieloliniwoe objete są #/ /#

3.2 Deklaracje

```
int z1 = 5;
```

Typy proste mają dedukcje.

```
auto z2 = 5;
```

Stałe, nie można ich nadpisać.

```
const auto z3 = "123";
```

Tuple też mają dedukcje!

```
auto a1 = (1, 2, 3);
auto a2 = 1, 2, false;
```

Funkcje nie mają dedukcji - typ jest podany dalej.

```
auto f1 = func(int a, int b) -> int {return a;};
(int, int) -> int f2 = func(int a, int b) -> int {return a;};
```

Działa coś na kształt aliasów.

```
auto new_type1 = int;
type_of(new_type1) a = 5;
auto new_type2 = () -> (int, int);
type_of(new_type2) g1 = func() -> (int, int) {return (1,2);};
type_of(g1) g2 = func() -> (int,int) {return 2,3;};
```

3.3 Typy

Da się rozpakowywać tuple.

```
int a, b = (1, 2);
```

Istnieje operator porównania typów ====
Typy są obywatelem peirwszej kategorii, da się na nie wprowadzać aliasy, porównywać je, dodawać je do siebie i mnożyć przez liczbę dodatnią.

```
type_of(a) == type_of(a);
auto n = 5;
type_of(n) == int;
auto int_str = int + string;
int_str == (int, string);
auto int_5 = int * 3;
int_5 == (int, int, int);
```

Jakie są prawa dla type_of.

```
auto int1 = int;
type_of(int1) == int
int1 == int
```

Nie można użyć type_of na nazwie funkcji w czasie deklaracji jej typu: auto f = func(type_of(f)) -¿ int return 1; .

3.4 Funkcje

Funckje są rekursywne, lambdy nie (bo nie mają nazwy). Obie wersje są statycznie wiązane. Argumenty są przekazywane by-name. Funckja/lambda musi coś zwracać. Funckja ma dostęp do środowiska, ale nie może go zmienić.

```
auto f = func(int -> int a) -> int -> int {
    return func(int i) -> int {return a(i);};
```

```
};
f === (int -> int) -> int -> int
auto g = func(((int -> int) -> int -> int) a) { return \(int -> int c, i
g === ((int->int)->int->int)->((int->int), int)->int
```

Typ int-¿int-¿int-¿int oznacza funkcję którą wywołujemy int d = f(a)(b)(c). Typ int,int,int-¿int oznacza funckję którą wywołujemy int d = f(a, b, c). Obie funckje mają inny typ.

```
auto c = 1,2;
auto ff = func(type_of(c) x) -> type_of(c) {return x;};
ff === (int,int) -> (int,int);
auto gg = func(int a, int b) -> (int, int) {return a,b};
ff === gg;
```

3.5 Wywoływanie

Wołać można funckje.

```
auto f = func() -> int {return 1;};
f()
```

Wołać można typy.

```
auto a = int(1)
a() == 1
```

Typy są w pewien sposób przeciążone.

```
int(1) === () -> int
```

Przydatne do bindingu.

```
auto f = func(int a, int b) -> int {return a+b;};
auto g = f(1);
g === int -> int;
```

I lazy evaluation!

```
auto loop = func() -> int {while True {} return 0;};
auto new_type = int * 3;
auto f = new_type(1,2,3);
auto g = new_type(1,loop());
f === () -> (int, int, int);
g === int -> (int, int, int); Nie zatnie sie.
g(1); Zatnie sie!
```

4 Gramatyka

4.1 Literały

⟨*literal*⟩ ::= ⟨*integer-literal*⟩ | ⟨*string-literal*⟩ | ⟨*bool-literal*⟩

⟨*integer-literal*⟩ ::= [`-`] ⟨*digit*⟩ {⟨*digit*⟩}

⟨*digit*⟩ ::= `0` | ... | `9`

⟨*string-literal*⟩ ::= `"`, {⟨*ASCII-character*⟩}, `"`

⟨*bool-literal*⟩ ::= `true` | `false`

4.2 Typy

⟨*type*⟩ ::= ⟨*arg-type*⟩ | ⟨*func-type*⟩

⟨*arg-type*⟩ = ⟨*simple-type*⟩ | ⟨*tuple-type*⟩ |`type_of(` ⟨*identifier*⟩ `)`

⟨*simple-type*⟩ ::= `int` | `bool` | `string`

⟨*tuple-type*⟩ ::= `(` ⟨*type*⟩, ⟨*type*⟩ {`,` ⟨*type*⟩ } `)`

⟨*func-type*⟩ ::= ⟨*func-arg-type*⟩ `->` ⟨*type*⟩ | `()` `->` ⟨*type*⟩

⟨*func-arg-type*⟩ ::= ⟨*arg-type*⟩ | `(` ⟨*func-type*⟩ `)`

4.3 Wyrażenia

⟨*expr*⟩ = ⟨*arithmetic-expr*⟩ | ⟨*logical-expr*⟩

⟨*logical-expr*⟩ ::= ⟨*logical-and*⟩ {`||` ⟨*logical-and*⟩}

⟨*logical-and*⟩ ::= ⟨*logical-rel*⟩ {`&&` ⟨*logical-rel*⟩}

⟨*logical-rel*⟩ ::= ⟨*arithmetic-expr*⟩ ⟨*rel-operator*⟩ ⟨*arithmetic-expr*⟩

⟨*rel-operator*⟩ ::= < | > | `<=` | `>=` | `==` | `!=` | `===` | `!==`

⟨*arithmetic-expr*⟩ ::= ⟨*mul-expr*⟩ {(`+` | `-`) ⟨*arithmetic-expr*⟩}

⟨*mul-expr*⟩ ::= ⟨*term*⟩ {(`*` | `/` | `%`) ⟨*term*⟩}

⟨*term*⟩ ::= ⟨*literal*⟩ | ⟨*identifier*⟩ | ⟨*type*⟩ | ⟨*paranthesis-expr*⟩ | ⟨*lambda-expr*⟩ | ⟨*tuple-expr*⟩ | ⟨*call-expr*⟩

⟨*call-expr*⟩ ::= ⟨*term*⟩ `(` [⟨*expr-list*⟩] `)`

⟨*tuple-expr*⟩ ::= ⟨*expr*⟩, ⟨*expr-list*⟩

⟨*expr-list*⟩ ::= ⟨*expr*⟩ {`,` ⟨*expr*⟩}

⟨*parenthesis-expr*⟩ ::= `(` ⟨*expr*⟩ `)`

⟨*lambda-expr*⟩ ::= `func`, ⟨*arguments-list*⟩ `->` ⟨*type*⟩ ⟨*block*⟩

4.4 Statystyka

⟨*statement*⟩ ::= ((declaration¿ | ⟨*assign*⟩ | ⟨*expr*⟩ | ⟨*return*⟩ | ⟨*print*⟩) `;`) | ⟨*flow*⟩

⟨*print*⟩ ::= print ⟨*expr-list*⟩

⟨*assign*⟩ ::= ⟨*identifier-list*⟩ `=` ⟨*expr*⟩

⟨*flow-statement*⟩ ::= ⟨*if*⟩ | ⟨*while*⟩ | ⟨*for*⟩

⟨*if*⟩ ::= `if` ⟨*expr*⟩ ⟨*block*⟩ [{`elif` ⟨*expr*⟩ ⟨*block*⟩} `else` ⟨*block*⟩]

⟨*while*⟩ ::= `while` ⟨*expr*⟩ ⟨*block*⟩

⟨*for*⟩ ::= `for` ⟨*identifier*⟩ `in` ⟨*range*⟩ ⟨*block*⟩

⟨*range*⟩ ::= ⟨*expr*⟩ `..` ⟨*expr*⟩

⟨*return*⟩ ::= `return` ⟨*expr-list*⟩

⟨*block*⟩ ::= `{` {⟨*statement*⟩]} `}`

4.5 Deklaracje

⟨*declaration*⟩ ::= [`const`] (⟨*type*⟩|`auto`) ⟨*identifier-list*⟩ `=` ⟨*expr*⟩

⟨*arguments-list*⟩ ::= `(` [⟨*typed-identifier*⟩ {`,` ⟨*typed-identifier*⟩}] `)`

⟨*typed-identifier*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩

⟨*identifier-list*⟩ ::= ⟨*identifier*⟩ { `,` ⟨*identifier*⟩ }

⟨*identifier*⟩ ::= (⟨*lower-letter*⟩ | `_`), {⟨*letter*⟩ | ⟨*digit*⟩ | `_`}

⟨*letter*⟩ ::= ⟨*upper-letter*⟩ | ⟨*lower-letter*⟩

⟨*upper-letter*⟩ ::= `A` | ... | `Z`

⟨*lower-letter*⟩ ::= `a` | ... | `z`

4.6 Program

⟨*Program*⟩ ::= `main` ⟨*block*⟩