1. Koncepcja

Głównym konceptem języka jest zachowanie "Single exit-point rule".

Zazwyczaj w językach strukturalnych stosowanie się do tej reguły może być problematyczne, ale chcieliśmy znaleźć takie połączenie które pozwoli zachować strukturalność kodu, czytelność i pozwoli stosować zasadę SEP.

W tym celu zdecydowaliśmy się na zastosowanie idei znanej z programowania funkcyjnego – Function Guards, która występuje w językach Haskell oraz Elixir.

1. Zmienne

* Zmienne inicjalizowane za pomocą słowa let, nazwy zmiennej oraz {}

Np. let a { 10 }; // int

let b { 1.0 }; // float

let c { true }; // bool

* Zmienne domyślnie są niemutowalne, aby można możliwe było je edytowanie należy zainicjować je za pomocą dodatkowego słowa mut (przed let):

Np. let x { 10 }; // stała

let y { 20 }; // zmienna mutowalna

* Silna typizacja – typ określany przy inicjalizowaniu zmiennej
* Zmiana wartości zmiennej, która wcześniej nie była zadeklarowana kończy się błędem – domyślnie nie da się ich edytować poprzez znak =

//idk czy tak ma być czy jednak powinno się dać, ale wydawało mi się że skoro domyślnie zmiennych nie można edytować to skoro zmienna nie została zadeklarowana jako zmienna to nie można jej „nadpisać”

* Występujące typy to:
* int
* float
* bool

1. Operatory

* Operatory równości: x != y oraz x == y
* Dodawanie/odejmowanie: x + y oraz x – y
* Mnożenie/dzielenie: x \* y oraz x / y
* Suma logiczna: x && y
* Alternatywa: x || y
* Negacja: !x

1. Sposób tworzenia funkcji:

Funkcje składają się z „bloków”:

* Definicja funkcji za pomocą słowa kluczowego func oraz nazwy funkcji
* Argumenty funkcji // czy tutaj powinny dojść wymagania na typ argumentu???
* Warunki względem argumentu
* Blok operacji
* Zwracanie wartości

Każdy z powyższych bloków oprócz deklaracji jest opcjonalny.

Zgodnie z ideą funkcje powinny być tworzone od najbardziej uogólnij a wersje dla szczególnych przypadków powinny być zdefiniowane poniżej oraz od najstarszej do aktualnej wersji. Kolejne zdefiniowanie funkcji nadpisuje poprzednie. Nie ważne czy poszczególne wersje są zdefiniowane dla takich samych czy innych argumentów. Dla danych argumentów wykonywana jest najnowsza funkcja dla której w/w argumenty spełniają warunki.

Przykład 1:

func fib // definicja funkcji

with x // argumenty funkcji

when x >= 0 // GUARDY na funkcje

returns fib(x-1) + fib(x-2) // rekurencja

func fib

with x

when x <= 1

returns 0 // Single exit point

func fib

with x

when x == 1

returns 1 // zwracanie wartości

W powyższym przypadku dla x > 1 wykona się podstawowa wersja funkcji, natomiast dla

x == 1 nadpisujemy dwukrotnie sposób oblicznia funkcji fib. Ostatecznie fib(1) zwróci wartość 1 – wykona ostatnią ze zdefiniowanych wersji.

Przykład 2:

|  |  |
| --- | --- |
| func func\_with\_flows  {  let xd { 123 };  if( xd > 10 )  {  println(xd);  }  mut let mutable\_var { 1 };  while( mutable\_var < 10 \* xd )  {  mutable\_var = mutable\_var \* xd;  }  for( mut let i = 0; i < 0; i = i + 1 )  {  mutable\_var = mutable\_var - xd;  }  } | // dfinicja funkcji  // blok operacji |

Jak widać porównując powyższe przykłady każdy z bloków funkcji występujących po deklaracji można pominąć.