Metody i algorytmy kompilacji

JJ language

Jakub Sordyl Jolanta Śliwa

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 stosować zasadę SEP.
   W tym celu zdecydowaliśmy się na zastosowanie idei znanej z programowania funkcyjnego – Function Guards, która występuje w językach Haskell oraz Elixir.
   W ramach projektu będziemy dążyć do stworzenia interpretera.

2. 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 było je edytowanie należy zainicjować je za pomocą dodatkowego słowa mut (przed let):
     Np. let x { 10 }; // stała
     mut let y { 20 }; // zmienna mutowalna
   - Silna typizacja
   - Zmiana wartości zmiennej, która wcześniej nie była zadeklarowana kończy się błędem
   - Występujące typy to:
     - int
     - float
     - bool
   - Możliwa konwersja typów za pomocą wyrażenia let nazwa { wartość as typ }:
     Np. let x { 1 as float };
3. 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

4. Sposób tworzenia funkcji:
   Funkcje składają się z „bloków”:
   - Definicja funkcji za pomocą słowa kluczowego func oraz nazwy funkcji
   - Argumenty funkcji
   - Warunki względem argumentu
   - Blok operacji
   - Zwracanie wartości

   Każdy z powyższych bloków oprócz deklaracji jest opcjonalny.

   Analogiczne jak w przypadku zmiennych, aby przekazywane wartości mogły być modyfikowane należy przed nazwą zmiennej użyć słowa „mut”.

   Przykład 1:

```
func add
with mut x
{
   x = x + 2;

}
```

   Zgodnie z ideą funkcje powinny być tworzone od najbardziej uogólnionej, 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 2:

```
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 3:

```
func func_with_flows                    // definicja funkcji
{
   let xd { 123 };
   if( xd > 10 )
   {
      println(xd);
   }                                    // blok operacji

   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;
   }
}
```

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

Przykład 4:

```
func main
{
   let x { 10 };
   mut let y { 100 };

   println( fib(y) );

   // x = 12 - error bo const          // komentarze
   // y = 12 - ok bo mutable
}
returns fib(x)
```