Język Nemerle

Język Nemerle

Z wykorzystaniem materiałów www.nemerle.org

Wszystko to co najlepsze

- Umożliwia pisać imperatywnie i funkcjonalnie
- Konstrukcje takie jak:
  - Obiekty
  - Warianty
  - Dopasowywanie wzorca
  - Polimorfizm parametryczny
  - Wyrażenia Lambda
- Zaawansowany system makr

Dlaczego Jeszcze Jeden język

- Przejrzysta „przemysłowa” składnia oparta na znanym C#
- Większość języków funkcjonalnych została „dostosowana” do CLI, Nemerle jest napisane specjalnie dla niego.
- Duży nacisk na komunikaty o błędach (jak na język hybrydowy)

Kilka małych zmian

C#

```
int x = 3;
string y = "foo";
FooBarQux fbq = make_fbq ();

int x = 3;
string y = "foo";
FooBarQux fbq = make_fbq ();

expr_1 = expr_2 = expr_3;



new Class (parms)

new Class [size]

new Class [size]
```

Nemerle

```
def x = 3;
def y = "foo";
def fbq = make_fbq ();

mutable x = 3;
mutable y = "foo";
mutable fbq = make_fbq();

def tmp = expr_3;
expr_1 = tmp;
expr_2 = tmp;

Class (parms)

array (size)

array (size) : array [Class]
```

Kilka małych zmian

C#

```
new Type[] { expr_1,
  expr_1, ..., expr_n }

(type) expr

(type) expr

class Foo {
  public Foo (int x)
  { ... }
}

switch(value)
{
  case 0: return "a";
  case 1: return "b";
  case 2: return "c";
}
```

Nemerle

```
array [expr_1, ...,
       expr_1, expr_n]

expr :> type

expr : type

class Foo {
  public this (x : int)
  { ... }
}

match(value)
{
  | 0 => "a";
  | 1 => "b";
  | 2 => "c";
}
```

I jeszcze mniejszych

```
variant RgbColor {
  | Red
  | Yellow
  | Green
  | Different {
      red : float;
      green : float;
      blue : float;
    }
}
```

Kod wariantu w C#

```
class Color {
  class Red : Color { }
  class Green : Color { }
  class Yellow : Color { }
  class Different : Color {
    float red, green, blue;
    public Different (float r,
                      float g,
                      float b) {
      red = r;
      green = g;
      blue = b;
    }
  }
}
```

Wydajniejsze tworzenie kodu

- Proste programy możemy zapisać jak w SMLu (bez klas czy funkcji Main)
- Dzięki automatycznemu typowaniu i możliwością pisania funkcyjnie programy są krótsze i czytelniejsze

```
def silnia(a)
{
 | 1 => 1;
 | x => x*silnia(x-1);
}
```

- Bezpośrednie wyrażanie prostszych typów np. typ funkcji z int w parę stringów możemy zapisać int -> string * string
- Nawet w obrębie CIL uznawane są krotki
- Można tez zrzucić obliczanie stałych wyrażeń na czas kompilacji

Automatyczne typowanie ?

- Czy to nie jest czasem nierozstrzygalne?
- Zapewne jest wolne ?
- C# 3.0 ma cukier, ale brakuje mu silnika typującego.

Zrzucanie obliczeń na kompilator

- Jeżeli wyrażamy jakieś stałe to nie ma sensu ich za każdym razem obliczać
- `def v = f()` a potem używać v? To zawsze odczyt zmiennej v w momencie jej użycia
- Ale przecież możemy obliczyć to w trakcie kompilacji!
- Pozwala to również na weryfikację parametrów w trakcie kompilacji

System metaprogramowania (makr)

- Technologia wykorzystywana w językach Lisp, Scheme, C, C++, Haskell
- Upraszcza budowę kompilatora
- Nemerlowa implementacja operuje na AST
- W łatwy i zrozumiały sposób można zdefiniować swoje konstrukcje lub rozszerzenia języka

Jedna z prostszych konstrukcji

- Piszemy

```
using (stream = System.IO.FileStream ("file.txt"))
{
  lock (this) {  ...  }
}
```

- Zamiast

```
def stream = System.IO.FileStream ("file.txt", System.IO.FileMode.Open);
try {
  System.Threading.Monitor.Enter (this);
  try {
    ...
  } finally {
    System.Threading.Monitor.Exit (this)
  }
} finally {
  def disp = (stream : System.IDisposable);
  when (disp != null)
    disp.Dispose ()
  }
}
```

Metaprogramowanie a .NET

- Makra mogą operować również na klasach, interfejsach, typach i metodach
- Dzięki składni atrybutów C# wyglada to znajomo
- Większość danych z `System.Reflection` znamy już w momencie kompilacji, po co obciążać nimi wykonanie głównego programu