Szorgalmi feladatok

Grafikus Processzorok Tudományos Célú programozása II. 2018.

Beadási Határidő: November 9.

Ha bármelyik feladattal kapcsolatban kérdés merül fel, e-mailben / Neptun üzenetben lehet pontosítást kérni.

1. Típus szintű filter:

Írjuk meg azt a template típust, ami C++ típus szintű listákon egy feltétel alapján szűr. A feltétel egy olyan template, ami igazat, vagy hamisat ad vissza egy belső fordítás idejű bool-al. Pl.:

```
template<typename T> struct IsEven;
template<int n> struct IsEven<Int<n>>
{
	static const bool res = (n % 2 == 0);
};
```

A filter template kap egy ilyen feltételt és egy listát, és visszaadja azt a listát, amiben csak azok az elemek vannak, amelyekre a feltétel teljesül:

```
using Result = typename

Filter< IsEven, List<Int<1>, Int<2>, Int<3>, Int<4>> >::result;
```

2. C++ érték szintű n-áris zip:

Ha csak két tárolót akarunk össze zip-elni, azt elég könnyű megírni, de mi van akkor, ha n darabot akarunk egy n-áris függvénnyel elemenként zippelni? Írjuk meg ezt az std::vector-ra, vagy az std::array-ra, vagy az órai tuple-re, ellenőrizzük, hogy mind az n db tároló azonos hosszú-e (ha nem, akkor adjunk hibát).

```
int main()
{
	auto t = std::vector<int>({2, 3, 4});
	auto v = std::vector<int>({1, 0, 1});
	auto u = std::vector<int>({1, -2, 5});
	auto l = [](auto x, auto y, auto z){ return x+y*y+z; };
	auto s = nzip(l, t, u, v);
}
```

3. C++ érték szintű n-áris zip paraméter pack-okra:

A C++ paraméter pack szintaxisa könnyen használható arra, hogy elemenként hattassunk egy függvényt:

```
[](auto f, auto... args){ f(args)...; }
```

Viszont ezzel egyszerre csak egy darab pack írható le, mert egy argumentum listában csak egy pack szerepelhet, ha másképp nem feloldható hogy mettől meddig tart egy paraméter lista. Találjuk ki, hogyan lehet mégis zip-et (n-áris zip-et, ismert n-re) írni ezzel a módszerrel.

A visszatérési értéket egy másik n-áris függvénnyel „fogyasszuk” el:

```
[](auto fres, auto f, auto... args){ return fres( f(args)... ); }
```

4. C++17 fold expressions:

A C++17-től kezdve létezik olyan konstrukció, hogy paraméter pack-okon lehet foldolni is:

https://en.cppreference.com/w/cpp/language/fold

Ez csak a beépített bináris operátorokkal tud azonban foldolni. Emlékezzünk, a beépített operátorokat lehet overload-olni (túlterhelni). Írjunk egy olyan könyvtárat, ami fold-ot valósít meg tetszőleges két argumentumos lambdára, egy paraméter pack-on, és a foldolást folx expression-el végzi!

Például ennek kell működnie:

```
auto l = [](auto x, auto y){ return x+y*y; };
auto res = foldl(l, 0, 1, 2, 3, 4, 5);
```

Az második argumentum a fold kezdőértéke, a továbbiak a paraméter pack részei. Engedjük meg, hogy ezek más típusúak lehessenek, tehát pl. ez is működjön:

```
auto printer = [](auto x, auto y){ *x << y << "\n"; return x; };
auto res = foldl(printer, &std::cout, 1, 2, 3, 4, 5);
```

5. Lambda kalkulus EDSL:

Csináljunk egy olyan lambda kalkulus kiértékelőt, ami ezúttal C++ beágyazott nyelvként működik, azaz nem input streamből, hanem magából a forráskódból dolgozik, és pl. az alábbi értelmes legyen vele:

```
int main()
{
	auto f = symbol("f");
	auto x = symbol("x");
	auto n = symbol("n");

	auto one = la(f, la(x, app(f, x)));
```

```
	auto succ = la(n, la(f, la(x, app(f, app(n, app(f, x))))));
	auto two = app(succ, one);

	//és az érdekes rész:
	auto result1 = one( [](auto x){ return x+1; }, 1.0 ); //result1 == 2.0;
	auto result2 = two( [](auto x){ return x+1; }, 1.0 ); //result2 == 3.0;
}
```

Kerüljünk minden dinamikus, pointeres struktúrát.

6.: Implementáljunk egy általános monoid osztályt, ami azt tudja, hogy tárolja a kétváltozós műveletet és az egységelemet, és automatikusan van rá operator*, amivel lehet két azonos monoidból származó elemet szorozgatni.

A következő kódoknak kell működnie:

```
auto my_monoid = make_monoid(
[](auto x, auto y){ return x+y; },/*ez a lambda a kétváltozós művelet*/
[](){ return 0; } ); /*ez a lambda az egység elem*/
auto x = my_monoid(5);
auto y = my_monoid(9);
auto z = x * y; //z-ben levő érték 14
auto w = y * my_monoid(); //a default konstruktor az egységelem, a w-ben levő érték 9
```

Fontos: két különböző függvénnyel létrehozott monoid objektumot ne lehessen összeszorozni!

Írjuk meg a foldl-foldr-t, hogy a fenti monoid osztállyal működjön.

Kód példa:

```
std::array<int, 3> A = {1, 2, 5};
auto sum = foldl( my_monoid, A );
```

a sum-ban levő érték 8, és sum is egy my_monoid példány. (nem muszáj a típusokat ellenőrizni, ha a monoid csupa generikus lambdával van felépítve).

7.: Csináljuk meg most monoid helyett a 6-os feladatot testre is analóg módon, automatikusan létezzen +, -, *, / operátorok és az egységelemek. A konstruktorban a felhasználónak a két darab bináris műveletet, azoknak az inverzeit és a 2 egységelemet kell visszaadnia.

8.: Fix pont kombinátor

a) std::function-el

b) std::function nélkül, pl. lambdákkal, vagy structokkal

A következő kódnak kell működnie (típushelyesen!, nyilván más rekurzív prototípusokkal is hasonlóan):

```
auto x = fix( factorial_proto )( 4.0 ); //x == 24.0
```

9.: Fix pont kombinátor típus szinten

A következő kódnak kell működnie, ahol a Factorial_Proto egy template, ami vár egy template-template-t és egy típust, és van benne result using vagy typedef, amin keresztül ki lehet venni az eredmény típust (kell hozzá egy a rekurziót lezáró típus szintű If-et is implementálni):

```
using X = Apply< Fix< Factorial_Proto >, Int<4> >; // tipusa Int<24>
```

10. Levi-Civita tenzor előállítása metaprogramként

Az órán bemutatottal szemben, ahol ε_{ijk} le volt tárolva elemenként, írjunk egy olyan template metaprogramot, amely a szimbólum definíciójából állítja elő a megfelelő indexű elemet, akárhány dimenzióban, tehát pl.

```
epsilon( Int<0>(), Int<1>() ) //= Int<1>()
epsilon( Int<1>(), Int<1>() ) //= Int<0>()
epsilon( Int<1>(), Int<0>(), Int<2>() ) //= Int<-1>()
epsilon( Int<0>(), Int<1>(), Int<3>(), Int<2>() ) //= Int<-1>()
```

Ajánlott algoritmus a ciklus felbontás, l. itt és itt. Használhatunk constexpr függvényeket is.

11. Lambda kalkulus kiértékelő template metaprogram:

Hasonló az előzőekhez, de ezúttal teljesen compile time:, ilyesmi kódnak kell működnie vele:

```
using One = La<'f', La<'x', App<Sym<'f'>, Sym<'x'>>>>;
using Succ = La<'n', La<'f', La<'x', App<Sym<'f'>, App<Sym<'n'>, App<Sym<'f'>,
Sym<'x'>>>>>>>;
using Two = Ev<App<Succ, One>>; //Az Ev végzi el a valódi kiértékelést

//ha létezik egy ilyen template:
template<typename A, typename B> struct Add;
template<int n, int m> struct Add<Int<n>, Int<m>>{ using result = Int<n+m>; };

//akkor működik ez:
```

```
using Result1 = Ev<One<Add, Int<1> > >; //Result1 == Int<2>;
using Result2 = Ev<Two<Add, Int<1> > >; //Result1 == Int<3>;
```