1 Gramatika

⟨*policy*⟩ ::= ⟨*simple_policy*⟩
| ⟨*type_policy*⟩
| ⟨*policy*⟩ '`cross`' ⟨*policy*⟩
| ⟨*policy*⟩ '`plus`' ⟨*policy*⟩

⟨*simple_policy*⟩ ::= ⟨*array*⟩
| ⟨*tuple*⟩
| ⟨*vector*⟩

⟨*array*⟩ ::= '`array<`' ⟨*N*⟩ '`>`'

⟨*tuple*⟩ ::= '`tuple<`' ⟨*type_list*⟩ '`>`'

⟨*vector*⟩ ::= '`vector`'

⟨*type_policy*⟩ ::= ⟨*type*⟩

⟨**simple_policy**⟩ definuje jednoduchou strukturu s běžnou sémantikou.

⟨**type_policy**⟩ definuje typ dat ve struktuře, funguje na základě definování typového "módu", kdy všechny následující data sdílejí daný typ, tuto funkci má i ⟨**tuple**⟩.

cross a **plus** jsou operátory kombinující jednotlivé policy. **cross** má přednost a jeho význam je, že struktura definovaná levým operandem je složena z podstruktur definovaných pravým operandem. Význam **plus** je jednoduché řetězení operandů za sebou.

2 iterace

Jednoduchá iterace je definovaná pro jednoduché policy a pro všechny policy, jež využívají pouze **cross** operátor. Iterace skrze tuple je prováděna paralelně, zatímco skrze array sekvenčně s intuitivní definicí procházení kombinace tohoto dle definicí výše (jako u matic vždy iterátor řádků, sloupců; iterátory vícerozměrným systémem + s fixovanou danou souřadnicí/souřadnicemi).

Procházení je definováno pro operátor **plus** pouze, jsou-li po sobě jdoucí policy stejného typu, pak je prováděno sekvenčně.

Uvažujme následující příklad pro ukázání iterace na SoA a AoS:

```
using policy1 = tuple<float, int, char> cross array<N>; // SoA
using policy2 = array<N> cross tuple<float, int, char>; // AoS

auto i1 = policy1::begin(data);
auto i2 = policy2::begin(data);
```

potom platí (vedle očekávaných vlastností), že i1->get<n>() a i1->get<n>() jsou stejného typu (kde $n \in [0..2]$) a je definováno lineární uspořádání na sjednocení množin obou iterátorů (? i operace na nich) tak, že:

$$\text{policy1::begin(data)} == \text{policy2::begin(data)} \wedge \text{i1} == \text{i2} \rightarrow {+}{+}\text{i1} == {+}{+}\text{i2}$$

3 at() funkce

Funkce at() je definována rekurzivním voláním pomocí template například následujícím způsobem (subject to change; zde jen velice zjednodušeně pro array), měla by korespondovat s implementací iterátorů:

```
struct cross;

template<unsigned length> struct array;
template<typename ...Ts> struct shape;

template<typename T, typename... Ts, unsigned n>
struct shape<T, array<n>, cross, Ts...> {
private:
    using sub_shape = shape<T, Ts...>;

public:
    static constexpr unsigned size = sub_shape::size * n;

    template<typename index, typename ...indices>
    static T &get(T *memory, index first, indices... rest) {
        return sub_shape::at(
            memory + (sub_shape::size * first),
            rest...);
    }
};

template<typename T, unsigned n>
struct shape<T, array<n>> {
public:
    static constexpr unsigned size = n;

    template<typename index>
    static T &at(T *memory, index first) {
        return memory[first];
    }
};
```