

# 1 Gramatika

Policy jsou definovány jednoduchou gramatikou definující strukturu dat (rozšiřitelnou o strategie procházení, zarovnání, atd.).

$$\langle policy \rangle ::= \text{'policy'} \langle shape \rangle \text{'>'}$$
$$\begin{aligned} \langle shape \rangle ::= & \langle tag \rangle \\ & | \langle tag \rangle \text{' , plus , ' } \langle shape \rangle \\ & | \langle tag \rangle \text{' , cross , ' } \langle shape \rangle \\ & | \langle tag \rangle \text{' , ' } \langle shape \rangle \text{ ('cross' assumed if not in tuple, 'plus' otherwise)} \end{aligned}$$
$$\begin{aligned} \langle tag \rangle ::= & \langle simple\_tag \rangle \\ & | \langle type\_tag \rangle \end{aligned}$$
$$\begin{aligned} \langle simple\_tag \rangle ::= & \langle array \rangle \\ & | \langle tuple \rangle \\ & | \langle vector \rangle \end{aligned}$$
$$\langle array \rangle ::= \text{'array'} \langle N \rangle \text{'>'}$$
$$\langle tuple \rangle ::= \text{'tuple'} \langle shape \rangle \text{'>'}$$
$$\langle vector \rangle ::= \text{'vector'}$$
$$\langle type\_tag \rangle ::= \langle type \rangle$$

$\langle simple\_tag \rangle$  definuje jednoduchou strukturu s běžnou sémantikou.

$\langle type\_tag \rangle$  definuje typ dat ve struktuře, funguje na základě definování typového “módu”, kdy všechna následující data sdílejí daný typ, podobnou funkci má i  $\langle tuple \rangle$ , je-li použit pouze s  $\langle type\_tag \rangle$ .

**cross** a **plus** jsou operátory kombinující jednotlivé tagy. **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. Uvnitř  $\langle type\_tag \rangle$  má **plus** operátor funkci čárky.

## 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 kompatibilního typu (vysvětleno příkladem níže), pak je prováděno sekvenčně.

## 2.1 Příklad

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 `i2->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:

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

## 3 Funkce at()

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:

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

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

    template<typename index, typename ...indices>
    static T &at(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>> {
    static constexpr unsigned size = n;

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