Koncepty programování

Obsah

1 Úvod

2 Programovací paradigma

3 Sémantika

3.1 Thunks

3.2 Closure

3.3 Lazy evaluation

Lazy evaluation neboli líné vyhodnocení

4 Typový systém

4.1 Function marchaling

5 Běhové prostředí

5.1 Allokátory

6 Metaprogramování

6.1 Reflexe

6.2 Homoikonicita

Homoikonicita (z řeckého homo ■ stejný a eikon ■ reprezentace) je vlastnost některých programovacích jazyků, kde má kód a datová struktura stejnou reprezentaci. V takových jazycích je program sám o sobě platnou datovou strukturou, se kterou lze manipulovat, číst ji a generovat, stejně jako s jakýmkoliv jiným typem dat.

Tento princip je naprostým základem jazyků, jako je **Lisp**, a umožňuje vytvářet jazyky, které jsou neuvěřitelně flexibilní a rozšiřitelné. Místo, aby byl kód jen text, je to objekt, se kterým může program pracovat.

6.2.1 Rozdíl mezi homoikonickým a nehomoikonickým jazykem

Většina běžných jazyků, jako je C, Python nebo Java, je nehomoikonická. V nich je kód pouze text, který je pro běhový systém srozumitelný až po složitém procesu lexikální analýzy, parsování a kompilace. Nelze do proměnné uložit kód jako takový, aniž by byl nejprve uložen jako řetězec, který je pak nutné znovu parsovat.

Naopak, v homoikonickém jazyce je stejný kód reprezentován jako datová struktura, například seznam. Kompilátor nebo interpret pak pracuje s touto strukturou přímo.

6.2.2 Klíčové důsledky homoikonicity

- **Zjednodušený parser**: Protože kód má stejnou strukturu jako data (např. v Lispu jako seznam), je gramatika jazyka mnohem jednodušší. Kompilátor nemusí řešit složité syntaktické detaily a může se zaměřit na to, co kód dělá.
- **Výkonné metaprogramování**: Homoikonicita je nezbytným předpokladem pro efektivní metaprogramování. To je psaní kódu, který píše, analyzuje nebo transformuje jiný kód. V takových jazycích lze vytvářet makra, což jsou programy, které vezmou kód (jako data), provedou na něm libovolnou transformaci (například ho přepíšou) a vrátí upravený kód, který se následně vyhodnotí.
- **Jazyk, který se rozšiřuje sám**: Protože se kód dá jednoduše manipulovat, lze v samotném jazyce vytvářet nové syntaktické konstrukce, které by jinak musely být vestavěnou součástí jádra. Tím se jazyk stává neuvěřitelně mocným a adaptabilním, protože jeho jádro může zůstat minimalistické. Díky tomu lze daný jazyk pomocí knihoven s předpřipravenými makry přizpůsobit k efektivnímu řešení určitého problému. To znamená, že není odkázán pouze na efektivní řešení problému pro který byl navržen (například jazyk C pro systémové programování a jazyk Python pro manipulaci s daty a rapid prototyping).

6.2.3 Homoikonicita v dynamicky typovaném systému

V dynamicky typovaném jazyce, jako je Lisp, se typy kontrolují až za běhu. To dává programátorům obrovskou flexibilitu, ale s sebou nese i rizika. V jazyce Lisp, který je dynamicky typovaný je kód reprezentován jako seznam. Makro vezme tento seznam (data), a protože typy se neřeší v době kompilace, může s ním dělat prakticky cokoliv. Například může v seznamu vyhledávat, měnit pořadí prvků nebo dokonce sčítat čísla a řetězce. Chyba se objeví až v okamžiku, kdy se interpret upravený kód pokusí vyhodnotit.

6.2.4 Homoikonicita ve staticky typovaném systému

Zavedením speciálního primitivního typu, jako je **stmt** (statement), se kód stává platnou a typově bezpečnou hodnotou. Makro, které očekává datový typ stmt, může tento typ

vzít, manipulovat s ním a vrátit nový stmt. Kompilátor pak provede statickou typovou kontrolu na nově vygenerovaném kódu. Makro sice může přepsat kód tak, aby byl typově nekonzistentní, ale v místě použití dojde v době kompilace k vyvolání syntaktické nebo sémantické chyby. Například makro, které má vrátit kód s typem i32, by nemohlo vrátit kód, který obsahuje string. Kompilátor by takovou operaci označil za chybnou. To platí i pro volání funkcí. Pokud makro přepíše volání funkce soucet, která očekává dva i32, na soucet(1, "ahoj"), kompilátor okamžitě nahlásí chybu, protože typy nesedí.

6.2.5 Homoikonicita v procesu překladu

Programovací jazyky, respetive kompilátory a interprety, které jsou založeny homoikonicitě fungují trochu odlišně než nehomoikonické. Rozdíl je již v reprezentaci kódu ve formě abstraktního syntaktického stromu. Abstraktní syntaktický strom pro homoikonické jazyky je tvořen rekurzivním seznamem, který se skládá z jednotlivých syntaktických elementů. Například kód:

```
a: i32 = 1 + 2
```

je v AST reprezentování jako seznam, na jehož prvky lze přistupovat

```
{ {type: ID value: a} {type: Symbol value: :} {type: Keyword value: i32}
{type: Symbol value: =} { {type: Number value: 1} {type: Symbol value: +} {type:
Number value: 2} } }
```

Tento AST ve formě rekurzivního seznamu pak vstupuje do další fáze, které se říká **preprocesing** kde jsou všechny makra rozbalena a jsou z nich vytvořeny funkce. Makra, která přijímají na svém vstupu nějakýseznam příkazů může tento seznam upravit, manipulovat s ním a nebo jej vykonat. Do další fáze překladu jde již hotový abstraktní syntaktický strom, který je možné běžným způsobemběžným způsobem přeložit na instrukce dané platformy a nebo vykonat ve virtuálním stroji interpreteru.

6.3 Složené typy a datové struktury