Monolog

Alexey Kachaev, E1

Obsah

Úvod

Monolog je jednoduchý, interpretovaný jazyk, podobný C svou syntaxi a konstrukcemi.

```
// Minimální hello world

println("Hello, World!");
```

Stručný popis programu

Běh programu probíha následovně:

1. Načtení zdrojového kodu (ze souboru nebo klavesnice)
2. Lexikální analýza *(lexing)*
3. Syntaxová analýza *(parsing)*
4. Semantická analýza a vygenerování syntaxového stromu (AST) *(semantic analysis)*
5. Interpretace prochazením AST *(tree-walk interpretation)*

Program podporuje režim REPL a vykonávaní ze specifikováného souboru.

```
1. monolog FILENAME
2. monolog scan FILENAME
3. monolog parse FILENAME
4. monolog repl
```

1. Spustí soubor s názvem `FILENAME`. V příkazovém řádku vratí 0 v případě uspěchu, poslední hodnotu předanou zabudované funkcí `void exit(int exit_code)`, nebo -1 v případě chyby za běhu.
2. Načté soubor `FILENAME` a vypíše posloupnost tokenů.
3. Načté soubor `FILENAME` a vypíše jeho syntaxový strom.
4. Spustí v režímu REPL. V tomto režímu uživatel interaktivně zadává příkazy, program pak každý zpracovává a vykonává. Veškere proměnné a funkce jsou pamatováný a použitelný mezi přikazama.

Také v režímu REPL interpretátor hned se neukončí v připadě chyby.

Originální zadání

- Aritmetické operátory (`+, -, *, /, %`)
- Inkrementace/dekrementace (`++, --`)
- Logické operátory (`&&, ||, ==, !=, >, <, >=, <=, !`)
- Spojování řetězců (`"hello" + " " + "world"`)
- Závorky (`2 * (2 + 3) / (3 - 4)`)
- Speciální operátor `#`, který vrátí počet prvku v poli nebo délku řetězce
- Speciální operátor `$`, který konvertuje celé číslo v řetězec. Použití u jiného typu vyvolá chybu při parsingu.
- Speciální operátor `*`, který vytěží uloženou hodnotu ve volitelném typu (viz dále). Pokud tento typ neobsahuje hodnotu, program ukončí se s chybou. Použití u void vyvolá chybu při parsingu.
- Operátor `[index]`, který se používá k indexování prvku v polích nebo znaků v řetězcích. Záporná hodnota nebo mimo hranice ukončí program s chybou.
- Datové typy: `int, void, string, pole` ([type, n], `kde type je uchovávaný typ,nje počet prvku). Pokud typ obsahuje na konci znak?, jedná se o volitelný typ. Objekt takového typu buď obsahuje hodnotu specifikovaného typu, nebo neobsahuje (přesněji řečeno obsahuje speciální hodnotunil`‘).
- Cykly `while` a `for`

- Podmínky `if`, `else if`, `else`
- Funkce s 0 nebo více parametry a návratovou hodnotou (`return_type name()` nebo `return_type name(param1, param2, ...);`)
- Komentáře začínají `//`
- Zabudovaná funkce `print`, která přijímá řetězec a vypíše ho. (`void print(string s);`)
- Zabudovaná funkce `println`, která přijímá řetězec a vypíše ho spolu s newline znakem. (`void println(string s);`)
- Zabudovaná funkce `input_int`, která načte celé číslo a vrátí ho (`int? input_int();`)
- Zabudovaná funkce `input_string`, která načte řetězec a vrátí ho (`string input_string();`).

Odchylky a změny

- Rozhodl jsem implementovat dynamické pole (nebo přesněji seznamy), které nemají pevně stanovený počet prvků a mohou se rozšiřovat dle potřeby, místo statických, které jsem hodlal implementovat nejdřív. Podrobně o tom je v 6. kapitole.
- Navíc k seznamům jsem přidal operatory `+=` a `-=`. Operátor `+=` přidá prvek na konec seznamu. Operátor `-=` na pravé straně bere číslo, což je počet prvku k odstranění z konce seznamu.
- Nová zabudovaná funkce `void exit(int code)`, která ukončí program uprostřed vykonávání.
- Zabudovaná funkce `input_string` teď ma navratový typ `string?`. Vrací `nil` v případě chyby ve čtení vstupu.

Kompilace interpretátoru ze zdrojového kodu

TODO: doplnit

Výrazy a příkazy

Monolog je **statický typovaný**, což znamená, že všechny proměnné a funkce mají pevně přiřazený typ, který se specifikuje explicitně při deklaraci/definici.

Jediný podporovaný paradigma je **imperativní programování** (procedurální) - vykonávání posloupností příkazu, které mohou přímo měnit stav programu.

Monolog je stavěn na výrazech a příkazech:

- výraz je název pro kombinaci operátoru, konstant, proměnných a funkcí, a dá se vyčíslit jeho hodnotu.
- příkaz vyjadřuje činnost, která ma být provedená. Může se skládat z výrazu.

Oboje mohou způsobit tzv. **vedlejší učínky** - jev, když výraz/příkaz ovlivňuje i jinej stav programu (např. hodnoty jiných proměnných) kromě své hodnoty.

Příkazy vykonávají se sekvenčně.

Význam výrazu

Význam výrazu také může záviset na tom, kde a jak je použit. Například,

```
int a = list[5];
```

výraz `list[5]` vratí hodnotu prvku s indexem 5 v seznamu `list`.

Ale příkaz

```
list[5] = 115;
```

výraz `list[5]` v tomto případe nevrací hodnotu, ale je interpretován jako destinace, kam má byt uloženo číslo 115.

To samé platí pro proměnné a indexování řetězcu (viz dále).

Datové typy

Monolog nemá možnost definovat vlastní typy, ale obsahuje zabudované:

1. celé číslo `int` - 64-bitové číslo se znamínkem
2. řetězec `string` - měnitelná posloupnost znaků (bytů)
3. volitelný typ `T?`, kde `T` je libovolný typ
4. seznamy `[T]`, kde `T` je libovolný typ
5. prázdný typ `void`

Rekurzivita typu je podporovaná, takže deklarace jako `int???????`, `[[[int?]?]]?` nejsou zakázaný.

Celá čísla

Primitivný typ `int` je určen pro prácí s celými čísly.

V Monologu, celá čísla jsou 64-bitová a mají znamínko.

Řetězce

Složený typ `string` je posloupnost znaku (hodnoty typu `int`).

Každý řetězec ma jednu vlastnost - **délka** - počet znaků v řetězci

Prázdný typ

Primitivný typ `void` je určen pro reprezentaci hodnot, které nemají hodnotu.

Volitelný typ

Volitelný typ `T?` je **složený datový typ**, který:

1. buď obsahuje hodnotu typu `T`,
2. nebo obsahuje hodnotu speciálního typu `nil` *(prázdnost)*.

Seznam

Seznam `[T]` je **složený datový typ**, který obsahuje prvky typu `T`, takže je zároveň **homogenní**.

Operátory

V Monologu jsou binární (`a + 2`), unární (`-b`) a sufixové operátory (`a++` nebo `list[5]`). Každý typ ma uřcitou sadu podporovaných operatorů.

Aritmetické operátory

Binární

Levá strana	Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`int`	`int`	`+`	provede sčítání	NE	`int`
`int`	`int`	`-`	provede odčítání	NE	`int`
`int`	`int`	`*`	provede násobení	NE	`int`
`int`	`int`	`/`	provede dělení	NE	`int`
`int`	`int`	`%`	provede dělení a vratí zbytek	NE	`int`

Unární

Levá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`int`	`+`	vrací hodnotu výrazu	NE	`int`
`int`	`-`	změní znamínko výrazu na opačné	NE	`int`

Logické a relační operátory

Tyto operace vracejí 1 pokud výraz je pravdivý, 0 pokud ne.

Binární

Levá strana	Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`int`	`int`	`==`	jestli hodnoty operandů jsou stejné	NE	`int`
`int`	`int`	`!=`	jestli hodnoty operandů nejsou stejné	NE	`int`
`int`	`int`	`<`	pokud první operand je menší než druhý	NE	`int`
`int`	`int`	`>`	pokud první operand je větší než druhý	NE	`int`
`int`	`int`	`<=`	pokud první operand je menší nebo rovný druhýmu	NE	`int`
`int`	`int`	`>=`	pokud první operand je větší nebo rovný druhýmu	NE	`int`
`int`	`int`	`&&`	provede logickou konjunkcí	NE	`int`
`int`	`int`	`||`	provede logickou disjunkcí	NE	`int`

Unární

Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`int`	`!`	provede logickou negaci	NE	`int`

Seznamové operátory

Binární

Levá strana	Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`[T]`	`T`	`+=`	vloží hodnotu pravé strany na konec seznamu	ANO	`void`
`[T]`	`int`	`-=`	smaže zadaný počet prvku z konce seznamu	ANO	`void`

- `-=`: pokud zadaný počet prvek je větší nebo roven počtu prvku seznamu, smažou se všechny prvky a seznam bude prazdný.

Unární

Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`[T]`	`#`	vratí počet prvků	NE	`int`

Sufixové

Levá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`[T]`	`[int]`	vratí odkaz na prvek, uložený v seznamu	NE	`T`

- `[int]`: tento operátor je **indexovací** a očekává uvnitř výraz typu `int`, který je pořadovaný index. Důležitý je, že hodnota indexu musí být v rozmezí $[0, N)$, kde N je počet prvků v danném seznamu.

Řetězcové operátory

Binární

Operace `==` a `!=` vracejí 1 pokud výraz je pravdivý, 0 pokud ne.

Levá strana	Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`string`	`string`	`+`	připoji pravý řetězec k levýmu	Ne	`string`
`string`	`string`	`==`	jestli délky **a** obsahy řetězcu jsou stejné	NE	`int`
`string`	`string`	`!=`	jestli délky **a** obsahy řetězcu nejsou stejné	NE	`int`

Unární

Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`string`	`#`	vratí délku řetězce (počet znaků)	NE	`int`

Sufixové

Levá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`string`	`[int]`	vratí odkaz na znak, uložený v seznamu	NE	`int`

- `[int]`: tento operátor je **indexovací** a očekává uvnitř výraz typu `int`, který je pořadovaný index. Důležitý je, že hodnota indexu musí být v rozmezí $[0, N)$, kde N je délka řetězce.

Operátory konverze

Unární

Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`int`	`$`	vytvoři nový řetězec, který obsahuje číslo	NE	`string`

Operátory přiřazení

Pokud výsledkem výrazu bude destinace, jako třeba proměnná nebo prvek v seznamu, a je na pravý straně, dá se změnit hodnotu, která je umistěna v destinaci.

Levá strana	Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
proměnná typu `T`	`T`	`=`	změní hodnotu proměnné	ANO	`T`
prvek v poli typu `T`	`T`	`=`	změní hodnotu uloženou v poli	ANO	`T`
znak v řetězci	`int`	`=`	změní specifikovaný znak	ANO	`int`

Při přiřazování hodnoty, hodnota se **kopiruje**. Takže, když například proměnné `b` přiřadíme hodnotu proměnné `a`, následně modifikace hodnoty `b` nebude ovlivňovat hodnotu `a`.

Řídicí příkazy

Monolog obsahuje základni příkazy pro větvení a cyklování kodu:

- Větvení
  - `if`
  - `else`
- Cyklování
  - `while`
  - `for`

if, else

Syntaxe

```
if-statement ::= 'if' '(' condition-expr ')' statement? ()?
```

`if`

- `podmínka` je výraz.
- `hlavní-tělo` je výraz nebo příkaz, je opcionální.
- `alternativní-tělo` je výraz nebo příkaz, je opcionální.

Chování

1. Příkaz `if` ověří, jestli podmínka (výraz v závorkach) je pravdivý (tj. nenulový). Pokud ano, vykoná se jeho hlavní tělo.

Tělo nemusí být.

2. Příkaz `if` ověří, jestli podmínka (výraz v závorkach) je pravdivý (tj. nenulový). Pokud ano, vykoná se jeho tělo.

Pokud není (tj. je nulové), vykoná se `else` větev (alternativní tělo).

Oba těla nemusejí být.

while

Syntaxe

```
while (podmínka)
    tělo
```

- `podmínka` je výraz, je opcionální.
- `tělo` je výraz nebo příkaz, je opcionální.

Chování

Příkaz `while` ověří, jestli podmínka (výraz v závorkach) je pravdivý (tj. nenulový). Pokud ano, vykoná se jeho hlavní tělo.

Pak opětovně zkontroluje podmínku, jestli je pravdivá. Pokud ano, tento proces se zopakuje. Pokud není, ukončí se.

Tělo nemusí být.

for

Syntaxe

```
for-statement ::= 'for' '(' init-clause? ';' condition? ';' iter-expr? ')' for-body
init-clause ::= expression | declaration
condition ::= expression
iter-expr ::= expression
```

- `inicializační-příkaz` je výraz nebo deklarace, je opcionální.
- `podmínka` je výraz, je opcionální.
- `iterační-příkaz` je výraz, je opcionální.
- `tělo` je výraz nebo příkaz, je opcionální.

Chování

1. Pokud `inicializační-příkaz` je uveden, příkaz `for` nejdřív vykoná jeho.
2. Pak, pokud výraz `podmínka` je uveden, ověří, zda je pravdivý.
3. Pokud `podmínka` je pravdivá nebo není uvedena, výkona `tělo`.
4. Po výkonávání těla, vykoná `iterační-příkaz`
5. Příkaz `for` ověří, jestli podmínka (výraz v závorkach) je pravdivý (tj. nenulový). Pokud ano, vykoná se jeho tělo (1. případ).
6. Příkaz `if` ověří, jestli podmínka (výraz v závorkach) je pravdivý (tj. nenulový).

Vázba jmen a entit

Deklarace je zavedení jednoho nebo více jmen, které má přiřazený význam a určité vlastnosti.

Monolog podporuje deklarace **proměnných** a **funkcí**

Proměnné

`variable-declaration ::= type-specifier identifier ('=' expression)? ';'`

Proměnné vytvářejí vazbu mezi jmenem a určitou entitou (hodnotou). Každá proměnná má uživatelem zadaný typ `type-specifier`, a opcionálně vychozí hodnotu, danou výrazem.

Pokud proměnná je deklarovaná bez počáteční hodnoty, její výchozí hodnota je vynulovaná:

Typ	Výchozí hodnota
`int`	`0`
`string`	`""`
`void`	`-`
`[T]`	`[]`
`T?`	`nil`

Použití proměnné ve výrazu dosadí její hodnotu.

```
// deklarace proměnné typu int s jmenem "a", vychozí hodnota je 0.
int a;

// deklarace proměnné typu int s jmenem "c", hodnotou které je součet hodnot proměnných a, b.
int c = a + b;

// deklarace proměnné typu string s jmenem "city", hodnotou je řetězec "Prague".
string city = "Prague";

// deklarace proměnné typu seznamu, který obsahuje seznam pruku volitelného typu int.
[[int?]] matrix;
```

Funkce

```
function-declaration ::= type-specifier identifier '(' param-decl-list ')' statement
param-decl ::= type-specifier identifier
param-decl-list ::= param-decl ','? | (param-decl ',')+ param-decl ','?

function-call ::= identifier '(' arg-list ')'
arg-list ::= expression ','? | (expression ',')+ expression ','?
```

Funkce váže jmeno a určitý kus kodu, který může mít předem definované parametry (`param-decl-list`), které může využit.

Volání funkce znamena vykonat určitou funkcí, a pokud má definované parametry, vykonat s určitými argumentemi.

Při volání funkce, typ každého argumentu se musí schodovat s typem parametru, jehož pozici zaujímá.

Rozsah platnosti

Rozsah platnosti je část zdrojového kodu, ve které jsou definované proměnné (tj. uplatňuje se vázba jména s entitou).

V každém programu napsanem v Monologu existuje alespoň jeden rozsah, zvaný **globální rozsah**. Globální rozsah má stejné vlastnosti jako i rozsahy vytvořené uživatelem.

Každá funkce vytváří nový rozsah platnosti pro své parametry.

```
// Zápis:

int sum(int x, int y) {
    return x + y;
}

int z = sum(arg1, arg2);

// Význam:

int z;

{
    int x = arg1; // arg1 má být typu int
    int y = arg2; // arg2 má být typu int

    {
        z = x + y; // return x + y;
    }
}
```

Vyvolávání funkce vytváří rozsah platnosti hned po globálním rozsahu, což dovoluje vyhnout se situací, když funkce má přístup k rozsahu volajícího, což není intuitivní a obvykle nechtěné chováni.

Rezoluce jmen

Rezoluce jmen znamená zjištění, na jakou entitu se odkazuje jméno. Monolog rezoluci provádí tak, že nejdřív hledá jmeno v současném rozsahu, pak, pokud existuje vyšší rozsah, hleda v něm a opakuje to až do globálního rozsahu, kde také provádí rezoluci. Pokud nebyla zjištěna entita, na kterou by odkazovalo danné jméno, je to považováno za semantickou chybu a program je špatně formulovan.

V případě funkcí, funkce může byt deklarováná jenom v globalním rozsahu, proto rezoluce jméne funkce provádí se jenom v něm.

Novy rozsah platnosti lze definovat pomocí **bloku** - skupinování příkazu.

Blok - skupinování příkazu

```
block-statement ::= '{' block-item* '}'
block-item ::= statement ';'? | (statement ';')+ statement ';'?
```

Blok vytváří nový rozsah platnosti a rozsah životnosti (viz dále) a pak sekvenčně vykonává každý příkaz nebo výraz.

```
// globální rozsah

int x;
int y;

// rozsah
{
    int z = x + y;

    // podrozsah
    {
        string w = $x + $y + $z;
    }

}
```

Paměťový Model

Paměťový model v Monologu je stavěn na základě rozsahu platnosti.

Rozsah žitovnosti pokrývá celý rozsah platnosti, a obsahuje všechny hodnoty a proměnné, které byly vytvořeny/deklarováný v příslušnem rozsahu platnosti.

```
// globální rozsah

int x;
int y;

// rozsah 1
{
    int z = x + y;

    // rozsah 2
    {
        string w = $x + $y +:$z;

        // životnost proměnné w končí tady
    }

    // životnost proměnné z končí tady
}

// konec zdrojového kodu programu
// životnost proměnných x a y končí tady
```

Každá hodnota má určit

Každá funkce a `for` cyklus definují vlastní oblasti.