Monolog

Alexey Kachaev, E1

Obsah

Pro učitelé

Myslím si, že celkem mám projekt dokončený, proto považoval bych to za final draft. Samozřejmně, počítam s možnými výtkami a návrhy pro vylepšení.

Implementace neodpovídá 100% původnímu zadání, ale odchylka je velice malá, a implementoval jsem dokonce trochu víc, než mělo být.

Kapitola Kompilace interpretátoru ze zdrojového kodu popisuje jak zkompilovat projekt.

Kapitola [Detaily implementace] popisuje implementaci.

Původní 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.

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.

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.

Operátory pro volitelné typy

Unární

Pravá strana	Operátor	Operace	Vedlejší učínky	Výsledný typ
`T?`	`*`	vytěží data z volitelného typu	NE	`T`

- **POZNÁMKA**: použití tohoto operátoru je zakázáno v případě, jestli objekt volitelného typu je prázdný.

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`
proměnná nebo prvek v poli typu `T?`	`T`	`=`	uloží hodnotu do volitelného typu	ANO	`T`
proměnná nebo prvek v poli typu `T?`	`nil`	`==`	ověří, zda levá straná je prazdná	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`.

Priorita operátoru a asociativita

Monolog respektuje prioritu a asociativitu operátorů, zejmena u matematických.

Následující tabulka uvádí prioritu a asociativitu všech operátorů. Operátory jsou uvedeny sestupně shora dolů, od nejvyšší priority po nejnižší.

Priorita	Operátor(y)	Popis	Asociativita
1	`++ -- () []`	Sufixové operátory	zleva doprava
2	`+ - ! # $ * ++ --`	Prefixové operátory	zprava doleva
3	`* / %`	Násobení, dělení	zleva doprava
4	`+ -`	Sčítání, odčítání	zleva doprava
5	`< <= > >=`	Relační operátory	zleva doprava
6	`== !=`	Rovnost, nerovnost	zleva doprava
7	`&&`	Konjunkce	zleva doprava
8	`||`	Disjunkce	zleva doprava
9	`=`	Přiřazování	zprava doleva

Řídicí příkazy

Monolog obsahuje základni příkazy pro větvení a cyklování kodu:

- Větvení
  - `if` - podminečné vykonávání kodu.
  - `else` - alternativní cesta kodu.
- Cyklování
  - `while` - cyklování.
  - `for` - iterace/cyklování.
- Další

- `return` - návrat z funkce.
- `break` - ukončení cyklu.
- `continue` - přeskočení těla cyklu.

if, else

```
if-statement ::= 'if' '(' expression ')' statement? else-statement?
else-statement ::= 'else' statement?
```

- Ověří, jestli podmínka je pravdivá. Pokud ano, vykoná se tělo.
- Pokud podmínka není pravdivá, vykoná se alternativní tělo, dané větví `else`.

while

```
while-statement ::= 'while' '(' expression ')' statement?
```

- Ověří, jestli podmínka je pravdivá.. Pokud ano, vykoná se tělo.
- Po vykonání těla, Opětovně ověří podmínku. V případě, že je stále pravdivá, tento proces se zopakuje. Pokud není, cyklus se ukončí.

for

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

- Pokud `init-clause` je dán, nejdřív vykona jeho.
- Pokud výraz `condition` je dán, ověří zda je podmínka pravdivá. Pokud `condition` není, jeho vychozí hodnotou bude čislo 1.
- Pokud podmínka je pravdivá, vykoná tělo.
- Hned po vykonávání těla, vykoná výraz `iter-expr`, pokud je dán.
- Opětovně ověří podmínku. V případě, že je stále pravdivá, tento proces se zopakuje. Pokud není, cyklus se ukončí.

return

```
return-statement ::= 'return' expression?
```

- Tento příkaz může se vyskytovat jenom ve funkcích.
- Způsobí, že vykonávání opustí aktuální funkcí a bude pokřacovat hned po místu v kodě, kde byla funkce vyvoláná.
- Pokud funkce má týp rozdilný od `void`, tento příkaz musi obsahovat návratový výraz, hodnota kterého bude vracená
- Pokud funkce má typ `void`, tento příkaz musí být bez návratového výrazu.

break

```
break-statement ::= 'break'
```

- Tento příkaz může se vyskytovat jenom v cyklech.
- Způsobí, že vykonávání opustí aktualní cyklus a bude pokračovat hned po konci tela cyklu.

continue

```
continue-statement ::= 'continue'
```

- Tento příkaz může se vyskytovat jenom v cyklech.
- Způsobí, že vykonávání přeskočí zbytek těla cyklu a cyklus bude opakovan.
- Po přeskočení, `while` ověří pravdivost podmínky.
- Po přeskočení, `for` nebude vykonávát iterační příkaz, ověří pravdivost podmínky.

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";

// volitelná proměnná, je prazdná (vychozí hodnota je `nil`).
string? jmeno = nil;
jmeno = "ahoj";

// deklarace proměnné typu seznamu, který obsahuje seznam prvku 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.

Když funkce má parametry a je vyvoláváná syntaxi `function-call`, na místo parametru jsou předáváný tzv. argumenty, a interpretátor pak vytvoří proměnné s názvem parametru a hodnotou přislušného argumentu, a kód funkce pak bude moci využit tyto proměnné (parametry).

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

```
// Funkce s názvem foo, bez parametrů, návratový typ je void,
// tělem je blok (viz následujicí sekce).
void foo() {
    println("Hello, World!");
}

// Funkce s parametry a návratovým typem int.
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Vyvolání funkce
foo();

// Vyvolání funkce s parametry
int m = max(115, 94); // argumenty jsou a = 115, b = 94.

void bar() {
    // Použití proměnné a funkce uvnitř funkce,
    // deklarované v globálním rozsahu. (viz následujicí sekce).
    if (max(m, 5)) {
        println("A");
    } else {
        println("B");
    }
}
```

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 a jejích rodičovským rozsahum (kromě globálního), což je kontraintuitivní a obvykle nechtěné chování.

Protože cyklus for dovoluje deklarovat proměnné, on taký vytváří nový rozsah, ale ten je podrozsahem rozsahu, ve kterém se vyskytuje tento cyklus:

```
// Zápis:

int z;

for (int i = 0; i < 10; ++i) {
    z = z + i * i;
}

println($z);

// Význam:

int z;

{
    int i = 0;

    while (i < 10) {
        z = z + i * i;
        ++i;
    }
}

println($z);
```

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 životnosti**, které úzce souvisejí s rozsahy platnosti.

Rozsah životnosti

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

Konec životnosti znamená, ze hodnota nebo proměnná se uvolnějí z pamětí a přestanou existovat, a pamět, kterou zaujímalí, interpretátor bude moci opětovně využit.

```
// 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
```

Statické a dynamické hodnoty

Podle využití paměti, hodnoty se dělí na:

1. statické
   - celá čísla (`int`)
   - prázdný typ (`void`)
   - `nil`
   - prazdné volitelné typy (`T?`)
2. dynamické
   - řetězce (`string`)
   - seznamy (`[T]`)
   - neprázdné volitelné typy (`T?`)

Dynamické hodnoty se uvolňují, když končí jejích rozsah životnosti. Pokud dynamická hodnota je hodnotou proměnné, hodnota bude uvolněná spolu s proměnnou.

Předávání argumentů u funkcí

Argumenty předávájí se takovým způsobem, že buď se kopírují, nebo předávájí se odkazem - změná parametru uvnitř funkce ovlivní hodnotu argumentu u volájicího.

Pokud hodnota argumentu je výsledkem nějakého výrazu a nemá vázané jmeno, tento argument bude vždy zkopírován. Pokud ale argument je proměnná, v závislosti od její typu, bude předan odkazem a změna hodnoty argumentu bude ovlivňovat proměnnu/prvek. Pokud argument je prvek v seznamu/řetězci, argument je vždy předáván odkazem. Take

Původ argumentu	Typ	Typ argumentu
Výsledek výrazu	`T`	kopie
Proměnná	`T`, `T` != `int`, `void` nebo `nil`	odkáz
Proměnná	`T`, `T` = `int`, `void` nebo `nil`	kopie
Prvek v seznamu nebo řetězci	`T`	odkáz

Zabudováné funkce

Monolog obsahuje zabudované funkce, které jsou všude přístupné.

print

```
void print(string s);
```

Vypíše řetězec `s` do standárdního výstupu.

println

```
void println(string s);
```

Vypíše řetězec `s` do standárdního výstupu spolu se znakem přenosu řádku.

exit

```
void exit(int code);
```

Ukončí program s hodnotou, danou parametrem `code`.

input_int

```
int? input_int();
```

Načté celé číslo ze standárdního vstupu. V případě, že celé číslo bude špatně zadano, nebo v průběhu načítání se stane chyba vstupu/výstupu, vratí `nil`.

Tato funkce je blokovací.

input_string

```
string? input_string();
```

Načté řetězec ze standárdního vstupu. V případě chyby vstupu/výstupu, vratí `nil`.

Tato funkce je blokovací.

Detaily implementace

Interpretátor je napsan v jazyce C, použitá norma je C11 (ISO/IEC 9899:2011).

Implementace nevyužívá rozšíření pro specifické konkretní kompilátor, proto kod by mělo být možný zkompilovat i pomocí jiných kompilátorů jako MSVC. GCC a Clang jsou podporovány.

Struktura projektu:

```
docs/                   dokumentace
  Makefile              Makefile pro generování PDF tohoto dokumentu
  prirucka.md           zdrojový kod tohoto dokumentu
include/
  monolog/              .h soubory projektu
    ...
src/                    .c soubory projektu
  ...
tests/                  testovací programy
  ...
third-party/            knihovny třetích stran
  ...
CMakeLists.txt          hlavní kompilační soubor
```

Lexer

Související hlavičkové soubory: `ast.h, lexer.h, source_info.h`

Související zdrojové soubory: `ast.c, lexer.c`

Úkolem **lexeru** je převest zdrojový kod (text, nejspíše psaný člověkem) do podoby, se kterou se dá jednoduše pracovat. Rovnou s textem není vhodný, protože to by komplikovalo kód a není to triviální.

Lexer převádí text na posloupnost tzv. **tokenů**. Laicky řečeno, token je v podstatě slovo - nejmenší jednotka v gramatice jazyku, která ma smysl.

Token je struktura, která uchovává odkaz na výskyt slova, druh slova (číslo, název atd.), číslo řádku a kolonky, a případně jiné informace:

```c
/* Druh tokenu */
typedef enum TokenKind {
    TOKEN_EOF,
    TOKEN_INTEGER,
    TOKEN_IDENTIFIER,
    ...
} TokenKind;

typedef struct SourceInfo {
    int line; /* rádek */
    int col; /* kolonka */
} SourceInfo;

typedef struct Token {
    /* druh */
    TokenKind kind;
    /* odkaz na výskyt ve zdrojovém kodu */
    const char *src;
    /* délka slova */
    size_t len;
    /* jestli je to validní token */
```

```
    bool valid;
    /* řádek, kolonka */
    SourceInfo src_info;
} Token;
```

Lexer funguje velice jednoduše: ověřuje současný znak, a podle něj určuje, jak to má pokračovat. Např. pokud slovo začíná na čislici, zřejmě se jedná o číslo.

```
Token next_token(Lexer *self) {
    /* přeskočit bílé znáky */
    find_begin_of_data(self);

    if (at_eof(self)) {
        return token_eof;
    } else if (is_digit(self->ch)) {
        return integer(self);
    } else if (is_identifier(self->ch)) {
        return identifier(self);
    } else if (is_operator(self->ch)) {
        return operator(self);
    } else if (self->ch == '"') {
        return string(self);
    }

    return invalid(self);
}
```

Ukázka funkce `integer()`, která lexuje číslo. Na stejným principu jsou založený ostatní.

```
Token integer(Lexer *self) {
    Token tok = new_token(TOKEN_INTEGER);

    while (!at_eof(self) && /* jestli lexer není na konci kódu */
           !is_ws(self->ch) && /* jestli aktuální znak není bílý znak */
           !is_operator(self->ch)) /* jestli aktuální znak není operátor */
    {
        /* jestli jsme našli, znak který není čislici,
         * číslo není ve validní formě
         */
        if (!is_digit(self->ch)) {
            tok.valid = false;
        }

        /* získat další znak */
        advance(self);
        ++tok.len;
    }

    return tok;
}
```

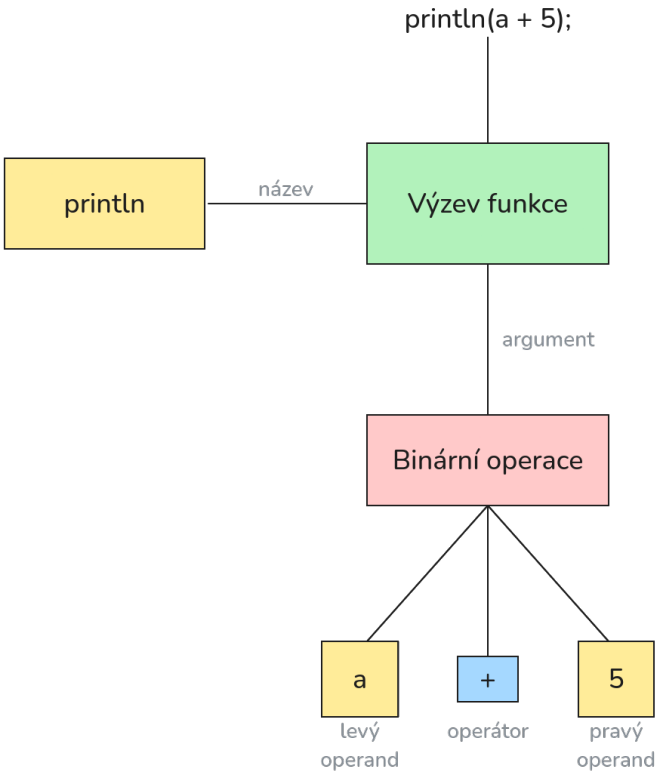
Ve výsledku, lexer vratí pole tokenů, které pak bude potřebovat parser.

Parser

Parser vytvoří tzv. **syntaxový strom** (dále AST, z anglického *abstract syntax tree*).

AST je stromová datová struktura, kde každý uzel vysokourovňově reprezentuje určitou část kodu.

Například, výraz `println(a + 5)` jde reprezentovat jako uzel `BinaryOp`, který reprezentuje binární operaci. On by měl dva uzly - levý operand a pravý operand - a pak znak operátoru.



Obrázek 1: Ukázka AST

Uzel ve stromu je reprezentovan strukturou `AstNode`:

```
typedef enum AstNodeKind {
    AST_NODE_INTEGER,
    AST_NODE_STRING,
    AST_NODE_BINARY,
    ...
} AstNodeKind;

typedef struct AstNode {
    AstNodeKind kind;
    Token tok;

    /* anonymní union */
    union {
        union {
            int64_t i;
            char *str;
        } literal;

        struct {
            Token op;
            struct AstNode *left;
            struct AstNode *right;
        } binary;

        ...
    };
} AstNode;
```

Tady pravě je využitá jedná z výhod C11, a konkretněji **anonymní union** - v některých případech nemá moc smysl uvádět jméno struktury ve struktuře, a tím pádem její členy jako by se vloží do rodičovské struktury, a zároveň kód pak bude čitelnější a kratší.

Způsob parsování

Parser je výhradně **rekurzivní a sestupný**. To znamená, že parsing probíha odzhora dolů, a využívá rekurzi. Každá funkce reprezentuje jeden z pravidel gramatiky.

Například, tato funkce parsuje binární operace:

```
AstNode *binary(Parser *self, AstNode *left) {
    ParseRule *op_rule = &rules[self->prev->kind];

    AstNode *node = astnode_new(AST_NODE_BINARY);
    node->binary.op = self->prev;
    node->binary.left = left;
    node->binary.right = expression(self, op_rule->prec);

    return node;
}
```

Přitom samotná funkce `binary()` se vyvolává ve funkci `expression()` (která parsuje jakykoliv výraz), takže je vidět, že se uplatňuje rekurze.

Prattův parser