

1 Pole

Jedna se o nejzákladnější a nejužitečnější datovou strukturu, se kterou se v programování setkáváme. V C se pole ovšem dají používat jen ve statických případech, kdy velikost můžeme stanovit ještě při kompilaci. Ovšem, i v případech, kdy to bude dostačující, tak budeme-li chtít předat pole do funkce, tak buď musíme definovat onu funkci pro konkrétní velikost pole, nebo využít ukazatele. První případ je použitelný jen zřídka, jelikož nepřináší abstrakci, kterou by jsme intuitivně chtěli, museli by jsme pro různé velikosti poli psát různé funkce.

To řeší předání pole přes pointer a, po případě, ukončením pole nějakým specifickým symbolem, nebo předáním doplňujícího parametru délky. V takovém to případě se však ztrácí jakákoliv výhoda vybraného datového typu, a vlastně i smysl onoho. Dostáváme méně explicitní kód, který je více nadolný na chyby, jelikož compiler time informaci, která se pojí jen k jedné proměnné, rozvádíme do dvou runtime.

Stanovíme tedy požadavky pro naše pole. Mělo by být využitelné ve funkcích bez ztráty identity a při tom být implicitním ukazatelem na počátek svých dat při přiřazení do pointeru. Navíc, rozšířit typ i na dynamické pole a dynamické pole variabilní délky.

1.1 Délka pole

Zavedeme získatelnou délku pole.

```
int [8] arr;  
arr.length; // vrati delku pole, tedy 8
```

Při předání pole do funkce se tedy předá ukazatel na data a jako skrytý parametr velikost pole. Pro případy, kdy není potřeba předávat velikost, se může použít pointer a implicitní přetypování.

1.2 Typy poli

1.2.1 Pole konstantní compile-time známé délky

Jedna se o pole analogické tomu, co je v C. Tedy

```
int [8] arr;
```

By vytvořilo pole o delce 8. Spočtená délka by se vždy dosazovala compile-time, realna proměnná pro její uchování by neexistovala.

1.2.2 Pole konstantní run-time známé délky

Jedna se o analogii alokace ukazatele v C, který by byl využíván jako pole.

```
int* arr = malloc(sizeof(int) * 8);
```

Tedy

```
int [const] arr = alloc int [8];
```

By alokovalo pole o delce 8 na heapu a vytvořilo by proměnnou pro uložení délky.

1.2.3 Array List

Protože se array list často využívá, bylo rozhodnuto ho implicitně přidat do jazyka v rámci poli. Jedna se pouze o dynamické pole s automatickou realokací při přidání prvku mimo rozsah. Tedy `int[] arr = alloc int[8]`; By alokovalo pole o delce 8 na heapu a vytvořilo by proměnnou pro uložení délky.

1.3 Práce s polem

Protože přinášíme rozdělení pole od ukazatele, tak bych s touto myšlenkou pokračoval dál a vnímal pole jako definici prvku stejných vlastností, jako něco, co určuje chování pro všechny jeho prvky v jednom místě. Vyjdeme-li z C, tak tato myšlenka je zde.

```
int arr[8];
```

Kde my vytvoříme kontejner pro 8 proměnných `arr[0] .. arr[7]` o datovém typu `int`, který specifikujeme jednou. Proměnná `arr` je však konstantní, jelikož není zřejmé, jestli např.

```
int arr1[8];
int arr2[8];
arr1 = arr2;
```

ma přiřadit všechny prvky `arr2` do `arr1`, nebo přepsat ukazatel `arr1` na ukazatel `arr2`. To plyne z toho, že vlastně pole jako takové v C by se dalo říct, že neexistuje. Jedna se fakticky vždy jen o ukazatel posypaný syntaktickým cukrem a v takovém to případě by, jaká koliv z předchozích operací, by mohla být vnímána jako hidden flow. My ovšem rozdělujeme mezi ukazatelem a polem na úrovni typu. Tedy můžeme rozdělit chování ukazatele a pole a prací s polem vnímat vskutku jako práci se všemi proměnnými naraz, aniž by jsme něčemu ublížili.

2 String

V C string literaly jsou pouze hezčí verze zapsané pole constantních charů. Vcelku, i když je to primitivní, tak zcela postačující. Problemem je zde 'absence' pole jako typu, jak již bylo zmínováno^[1]. Tedy vlastně se s každým stringem pracuje jako s pointrem. Jelikož my vnímáme pole jinak, tak můžeme rozvinout možnosti pole, aby nám umožňovaly ve výsledku lehčí práci i se stringy. Samotný typ pro string existovat nebude, ale bude jen podpora string literalu, který se při kompilaci rozloží na pole.

2.1 UTF-8

Bylo by vhodné rozšířit podporu literalu z ASCII na jiné kodování, aby byla jednoduchá cesta s případnou jednoduchou manipulací se složitějšími symboly. Jako takové kodování bych volil utf8, protože je kompatibilní s ascii, jeho blokem je byte, tedy není závislé na edianech a je velmi rozšířené.

Jako nejlepší možnost bych viděl zadání literalu v utf8 a compile-time vyhodnocení největší délky potřebné pro uložení jednoho symbolu, a následně konverzi na pole `intu` o patřičné velikosti, kde každý element bude samostatným symbolem zakódovaným v utf8.

2.2 Operace

Jako jediné konvenční operace nad stringy které by se měly integrovat do syntaxe, bych viděl concatinační a slice. Ostatní operace by už měly být obsaženy v standardní knihovně.

2.2.1 Spojení

Níc nového bych nevymyslel, a použil operator `..` jako v Lua.

```
int [] str1 = "Hello ";
int [] str1 = "World ";
int [] str3 = str1 .. " " .. str2;
```

Implementace by však nesměl obsahovat žádnou alokaci paměti, bylo by to zavádějící. Jelikož délky dynamických poli jsou jejich 'součástí', šlo by to využívat pro alokovaná pole. Ovšem, samotná, opět, nesmi obsahovat alokace, tedy případné výsledné pole by se muselo samostatně standardně alokovat prostředky jazyka.

```
u8[const] arrC3 = alloc [] : arrA .. arrB;
```

...

2.2.2 Slice

...

2.3 Namespace

Zručný nástroj k organizaci kódu. Umožňuje zhlukovat proměnné pod jedním společným názvem, který je rozlišitelný parsrem. Na rozdíl od použití identifikujících prefixu / postfixu v názvech je strukturním celkem z hlediska nástrojů operujících s kódem (např LSP). Umožňuje také při kompilaci hromadně pracovat s definovanými vevnitř proměnnými, a tedy se dá dobře využívat i pro import export části kódu (např python `import foo; import from foo x;`).

V našem případě by namespacem byl jednoduše pojmenovaný scope.

```
namespace Foo {
    x;
}
```

K přístupu by se použila syntaxe z C++ `Foo::x;`

2.4 Import

Nejhorší části C jsou hlavičkové soubory a s nimi související systém importu. Hlavní nevýhodou kterého je duplicita definic. Slouží však k dobrému úmyslů, k izolaci implementací a definici rozhraní.

My teda budeme chtít tuto myšlenku ponechat a rozvíět.

Základním celkem bude soubor, jelikož jeto to co ve vysledku předame překladači. Překladač dostane jen jeden vstupní soubor, který následně již za pomoci prostředku jazyka umožní načíst další soubory. Všechny importy však budou probíhat v rámci AST, každý soubor by tedy měl být samostatně parsovatelným celkem.

Prvně ošetříme možnost přímého importu souboru. Použijeme intuitivní syntaxi.

```
import filename;
```

Protože se v importovaném souboru mohou vyskytnout stejné názvy proměnných, chceme mít možnost zabalit ho do namespace.

```
import filename as namespace Foo;
```

To nám vytvoří namespace Foo a kořen rozparsovaného souboru filename se přidá jako jediný prvek do něj.

Syntakticky specifikujeme namespace, protože by jsme mohli využít onoho syntetického konstruktů k implementaci jiných způsobů zabalení souboru.

Např.

```
import filename as scope;  
import filename as fcn foo;
```

apod.

Dále, samozřejmě, budeme chtít umět vybrat patřičné namespace ze souboru (Popřípadě i identifikátory).

```
import from filename foo, boo as namespace Foo
```

V zásadě tohle nám umožní robustní import, a více prostředků potřebovat nebudeme. Zbývá zohlednit viditelnost jednotlivých identifikátorů.

Můžeme buď vycházet z toho, že vše je viditelné, a my omezujeme viditelnost, nebo naopak, vše je nepřístupné, a my rozšiřujeme přístup. Druhy přístup je víc prakticky, ale ztrácí na explicitnosti, protože, když importujeme soubor, tak intuitivně očekáváme, že se nám tam naimportuje všechno, než nic.

V celku, onen problem není tak podstatný, podstatnější je otázka viditelnosti vnořených importu. Tedy importuje li soubor, který importujeme, identifikátory z jiného souboru, budeme li je vidět také. Zřejmé je, že pokud jsou přístupné při importu, tak by měly být přístupné i pro další importy, jelikož jsou na stejné úrovni jako kód souboru, a nekladli jsme žádným způsobem omezení.

Tedy, navrhol bych umožnit omezit viditelnost importu, než omezovat viditelnost samostatných identifikátorů. Pak by jsme měli decentní explicitní možnost omezení viditelnosti symbolu, aniž by jsme to museli řešit poprvkově a navíc by jsme stále měli možnost vytvoření případného rozhraní z dostupných symbolu, které by se umísitili do jednoho souboru a zbyte by se importovali lokálně.

K označení lokálních importu bych použil slovo `local`

```
import filename as local namespace Foo
```

3 Function Overloading

I když se jedna o implicitní konstrukt, který skryva od čtenáře pravou identitu volané funkce, tak přináší, z mého hlediska, jednu zásadní věc, zjednotěná jména funkcí. Tedy, zamísto vepisování datového typu do jména funkce, můžeme jen uvést její činnost. To zjednodušuje vnímání samotného programu, jelikož při práci s vlastními datovými typy, které definují složité objekty, jména funkcí budou už znatelnou zátíží, oproti např. `maxi`, `maxf`, `maxu`, kde můžeme přibližně vydedukovat typ očekávané proměnné, se tak jednoduše nevystačíme. Navíc jména samotných funkcí s použitím postfixu/prefixu, které si zvolíme pro identifikaci, nebudou samostatnými celky z hlediska nástroje pracujících s kódem, tedy v základu samotným kompilátorem a např. LSP. Tedy nebude se moct nad nimi provádět žádná kontrola, tedy např. kontrola překlepu, stíženy refactoring, horší napověda, analýza atd..

Navíc, samotná abstrakce nad konkrétní volanou funkci pro čtenáře není nikterak zavadějící. Nebo spíš, je stejně zavadějící jako for loop, který za místo instrukce abstraktní instrukce for provádí skoky a sem a tam. Smysl čtenář získává ze samotného názvu funkce a vstupních proměnných, a vnímá konkrétní funkci jako černou skříňku. Tedy i když ona funkce dostává int, tak nemůže vědět, že ten int není hned první instrukci přetypován do floatu. Tedy jediné co to ovlivní je rychlost nalezení správné funkce při potřebě se podívat na její kod, což, bez užití LSP, bude zřejmě delší, ovšem, neřekl bych, že to není něco zavažného.

Tedy, opět, z mého hlediska, je lepší ho mít, než nemít. Zbývá rozhodnout, zda povolit implicitní overloading, tedy jestli

```
foo (int x);
```

nebo pro jiný datový typ musí být explicitní cast

```
foo ((int) 1.0);
```

Zde však dochází k zajímavému jevu. My explicitně vepisujeme datový typ, čímž identifikujeme funkci, ovšem o tom, jestli potřebná varianta existuje, se dozvíme buď z LSP, v tomto případě je cast jen z hlediska informace navíc (porovnáváme-li s implicitním overloadingem), nebo při kompilaci, což už je trochu pozdě. Tedy jediný k čemu to může sloužit je jako assert, kdy my víme, že chceme jít do konkrétní varianty oné funkce, a pokud neexistuje, tak dostat error. Ovšem to budeme muset specifikovat u každého volání overloaded funkce, což se přeci s tím, že chceme overloading hlavně z důvodu zjednodušené jmeňné stopy v kodu (nemluvě o tom, že vlastně máme tutéž informaci dva x v kodu, jednou při definici, po druhý při volání).

Bylo by tedy vhodné mít implicitní overloading, ale s opcí v jistých případech specifikovat konkrétní požadovaný datový typ. Zavedeme tedy příslušnou symboliku

```
foo ! ();
```

Využití prapodivného symbolu v tomto případě není zavadějící, jelikož očekávané intuitivní chování výrazu se nemění. Jedná se stále o function call, který nijak nemění výsledky volání ani jeho vstupy, z hlediska čtenáře je prakticky irelevantní.

3.1 Implementace

V C++ implementuji následovně bla bla bla ... https://en.cppreference.com/w/cpp/language/overload_resolution

My budeme postupovat obdobně.

Pro jmeno volané funkce najdeme všechny funkce se stejnými jmeny a v odpovídajícím scope. Uložíme do pole, kde v každém chlivečku bude struktura odkazující se na funkci a doplňující případné informace popisující schodu. Pro zatím, neuvažujeme-li polymorfismus, genericitu atd... si postačíme jen s jednou jedinou proměnnou typu int určující podobnost funkce našemu vzoru z volání.

Budeme procházet ono pole postupně funkci po funkci a buď je vyřazovat, nebo sestavovat skóre podobnosti. Nakonci vybereme funkci s největším skóre. Chyba nastane pokud budou dvě a více stejná maximální skóre, nebo nezbude žádná funkce se skórem.

3.2 Přístup jiných jazyků

V Odin je pouze explicitní, jelikož jazyk umožňuje definovat vnořené funkce ve funkcích, a tudíž rozlišení konkrétní funkce, která se má zavolat není triviální. <https://odin-lang.org/docs/overview/>

Zig nemá function overloading, ale podobného chování lze docílit při kompilaci za pomoci tzv. duck typing. <https://ziglang.org/documentation/master/>

4 Error handling

Vezmeme-li C, tak jazyk nenabízí přímý způsob spravy chyb. Chyby se mohou řešit např. návratovou hodnotou, nějakým specifickým stavem očekávané výstupní proměnné předané přes ukazatel (většinou NULL), speciální funkci, která vrací poslední chybu atd... V celku je to na programátorovi, aby vytvořil nějaký systém pro spravu chyb, a jestli vůbec. Pak, při práci s libovolným kódem je nutně číst komentáře k funkcím, externí dokumentaci apod. Zde opět narážíme na problém, kdy důležitá informace není součástí strukturních elementů kódu, ke kterým by měly různé nástroje přistup. Take

to postrada jednotnost, kde různé knihovny mohu řešit zpravu chyb vždy jinak, a ve výsledném programu se bude muset řešit zbytečný problem, jak s tím naložit. To vše nás ve výsledku vede k myšlence o přidání standartního systému pro spravu chyb v našem jazyce.

Podíváme-li se na jiné programovací jazyky a jejich metody řešení spravy chyb, tak dokažeme v zásadě vyčlenit dva přístupy.

Navratova hodnota Chyba je vracena jako navratova proměnná nebo její součást. Obvykle je to spojene s možností návratu několika proměnných, kde se vyděluje jedno, např poslední místo, pro případnou chybu (Odin), nebo je přímo speciální doplňující navratova hodnota vydělena jen pro chybu (Go). Nebo, třeba, se může vracet struktura obsahující jak případnou chybu, tak i navratovou hodnotu (Rust).

Tenhle přístup je přímočarý a explicitní a dává svobodu programátorovi jak a kde s chybou naložit. Zpracování chyby je pak přípmou součástí code-flow. Tedy chybový stav je prakticky jen další stav programu.

Try-Catch Využívá se systém tzv. exceptions, kde případně chybové místo je zabaleno do try bloku, a případná chyby odchycena v catch bloku. To umožňuje např. nezatěžovat kód správou chyb, a psát ho v try bloku tak, jako kdyby žádná chyba nastat nemohla, a následně jakoukoliv chybu zohlednit v catch bloku.

S try-catch se většinou pojí i tzv. throw mechanismus umožňující označit případné chyby, které může kód nějaké funkce vyvolat, a propagovat jejich ošetření do bloku, jež onu funkci volal.

- Jednotný datový typ.
- Umožnit vytvoření množin chyb, které by se mohly kompozičně skladat do nových množin. Např. můžeme vytvořit množinu chyb pro načtení souboru a množinu chyb pro zápis do souboru. Pak, budeme-li chtít vytvořit funkci, která čte a zapisuje do souboru, tak by jsme měli mít možnost spojit oně dvě množiny do jedné.
- Definice funkce musí specifikovat, které chyby při její volání mohou být vráceny.
- Umožnit jednoduchou propagaci chyby stakem funkcí dal. Tedy zjednodušit obdobný konstrukt `err = foo(); if err != null : return err ;`, který je

relativně frekventní.

<https://www.youtube.com/watch?v=uoIutDC5iBE>

4.1 Implementace

Protože nahlízet na chybu jako jen na další stav programu, i když, řekněme, speciální, je z mého hlediska přirozenější a implicitní přístup, tak se vydáme cestou navratové proměnné.

Jelikož používáme jen jednu navratovou proměnnou, chybu budeme chtít vracet samostatným kanálem. Ovšem, nebudeme chtít vnímat chybu jako přímo navratovou hodnotu, která je určena jen pro chybu, jak je tomu např. v Go. Protože pak musíme řešit po každé volání funkce dvě výstupní proměnné. To ve výsledku povede k vytvoření buď implicitních pravidel, nebo, k mnohomluvné (verbose) syntaxi.

Představme si to na následujícím příkladu v Go.

```
func foo() (int, error) {  
    return 42, nil;  
}
```

```
val1, err := foo();  
if err != nil { ... }
```

```
val2, err := foo();  
if err != nil { ... }
```

Symbol `:=` vyjadřuje, obdobně jako v Pascalu, definici s inicializací. Zde není zcela zřejmé, co se má dít, jelikož prvně provádíme definici `val1` a `err`, a následně, v témtýž scope provádíme definici `val2` a opět `err`. Samozřejmě, je to zohledněné pravidly jazyka, a kód je kompilovatelný, a nová definice `err` se neprovede, pouze `val2`. Ovšem, řekněme, dochází ke sporu syntaxe a semantiky, kde ze syntaktického hlediska se `err` chová jen jako druhá navratová hodnota, ovšem ze semantického se implicitně provádí 'vyjimky' v pravidlech, jen protože je to chybová hodnota. Navíc se to kompiluje přidáním kvalifikátoru. Budeme-li chtít označit `val1` jako `const` ale ne `err`, nebo naopak, budeme-li chtít mít jedno `embed` a druhý `const`, atd... To vše lze řešit na

ukor upovidané syntaxe, budeme-li chtít být explicitní, nebo přidáním implicitních pravidel. Proto se pokusíme najít jiné řešení, které by více sedělo naší vizi.

K návratu chyby využijeme tedy pravou stranu příkazu. To nám ponechá příkaz, nad kterým budeme chybu odchycovat, nezměnným, a tedy budeme moct jednoduše jak měnit samotný příkaz, tak i ošetření jeho chyb, jelikož syntaktický na sobě nebudou závislé. Navíc to nám může do budoucna umožnit odchycení chyby nejen z jednoho volání funkce, ale i z libovolného výrazu, který by mohl obsahovat několik volání funkcí.

Navrholval bych následující syntaxi.

```
int x = foo() catch err;
```

Kde se případná chyba uloží do proměnné `err`.

Zde bych stanovil, že nechceme zbytečně zesložitovat datový typ chyby přidáním různého implicitního chování, nebo různých druhů konstruktů pro tvorbu chyb. Chyba je vždy jen datového typu `error` a chová se vždy stejně. Tedy, můžeme v takovém to případě pominout samotnou definici `err`, jelikož je redundantní. Ošem, můžeme ji tam i mít.

```
error err;  
int x = foo() catch err;
```

4.1.1 Množiny chyb

Samotná chyba by měla být jednoduše identifikovatelná přes svoje jméno, aby ji bylo možné používat pro určení stavu. Např.

```
if err == ErrName : foo();
```

Chyby by měly být shlukovány do uživatelem definovaných skupin, které by pak sloužily pro určení chybového rozhraní funkcí. Skupiny by měly být shlukovatelné, jelikož funkce by měla mít možnost navracet i chyby užívaných funkcí, které mohou být definované samostatně, aniž by se pro ní redundantně definovaly nové chyby.

Tedy, řekněme, že budeme moct definovat jakysi množiny chyb, a jen je. Použijeme následující syntaxi.

```

error ErrorSetA {
    ErrorA;
    ErrorB;
};
error ErrorSetB {
    ErrorSetA;
    ErrorB;
};

```

Pak `ErrorSetA` je množina obsahující `ErrorA` a `ErrorB`, prázdné množiny, a `ErrorSetB` obsahuje množinu `ErrorSetA` a prázdnou množinu `ErrorB`. Libovolná s těchto množin je identifikovatelná svým jménem a může být přiřazena do datového typu `error`.

```

error err = ErrorSetB :: ErrorB;

```

K definici chybového rozhnutí funkce pak použijeme následující syntaxi.

```

fcn foo() using ErrorSetB => int {...}

```

funkce `foo` pak může vracet chyby definované v `ErrorSetB`, ale také i samotný `ErrorSetB`, což je nutné, abychom mohli využívat i případně prázdných množin definovaných vně jiných množin.

Protože oné množiny mají smysl jen při definici samotných funkcí a my neumožňujeme definovat funkci ve funkci, tak jejich definice uvnitř funkcí je zavádějící, a tudíž zakazána. A tedy můžeme vnímat oné množiny jako nadstavbu nad namespace pro chyby, a tedy k jejich diferencii používat stejný symbol `::`, jak již bylo naznačeno viz.[...].

Toto řešení je jednoduché a relativně všestranné. Umožňuje nám např. rozvíjet některou prázdnou množinu na plnohodnotnou, aniž by jsme rozbili kód, který onu množinu využíval. Ovšem, má jeden základní nedostatek – vracíme pouze stav. Tedy nemůžeme vrátit informaci o chybě. Teoreticky je to možné řešit přidáním počtu stavu, ovšem to zdaleka není praktické.

To nás omezuje jen při logování chyby, protože jinak my vždy popisujeme stav programu, který je nezbytný z hlediska jeho činnosti. Tudíž přidání v takovýchto případech chybového stavu je vlastně nezbytné (uvažujeme-li, že chceme tento stav mít jako chybový, obecně, samozřejmě můžeme ho řešit normální cestou). I tak máme možnosti, jak to zohlednit.

Pomocné proměnné V tomto případě využijeme stavu vstupních proměnných, buď již existujících, nebo nových, pomocných, k popisu chyby. Tedy např. máme-li následující funkci, která vyhledává v souboru slovo a vrací idx symbolu, kde se to slovo začíná vyskytovat

```
fcn find(u8[] fname, u8[] str) using IOErrorSet => int {...
```

tak můžeme i s chybou vrátit index, kde se chyba vyskytla.

Logování v místě vyskutu chyby Je zřejmé, že přímo v místě vyskutu chyby máme veškerou informaci k tomu, aby jsme ji popsali. Ovšem může nam chybět kontext, kdy pro nás bude důležitá informace z funkce, která nás volala. Tedy můžeme postupně logovat jen informaci dostupnou nam v daný moment a ve výsledku dostat podrobnou zprávu. Hlavním nedostatkem je však samotná nutnost logovat, protože pak musíme mít přístup k nějaké příslušné funkci, která by to dělala dle našich potřeb. Což je fakticky nerealizovatelné. Ovšem dalo by se to řešit za pomoci tzv. kontextu viz[], kde by všechny funkce mohly využívat standardního zápisu do stdout, ovšem by jsme pak z vně mohly definovat kontext, který by všechna tato volání přesměroval do naší zvolené funkce.

Ani jedna z možností není ideální, ale ve výsledku je to jen něco, co slouží jako doplnění systému. Něco, co je využíváno přímo při zpracování samotné chyby, a tedy neruší samotnou standartizaci, kterou jsme si kladli za cíl, protože popis samotné chyby už není obecně standartizovatelný, a tak či onak se jedná o konkrétní záležitost.

Pokud by jsme chtěli zobecnit náš model a rozšířit definici za pomoci struktury nebo unie, tak vlastně narazíme přitom se standartizací, protože zobecníme systém natolik, že bude moci být využíván i pro jiné věci, a také mnohá způsoby, tudíž vlastně naší postavený problém nevyřešíme, jen ho přesuneme jinde.

Jediný co by jsem mohli udělat, je povolit přiřazení chybám konkrétních hodnot, což by mohl být postačující kompromis. To umožní pak indexovat pole hodnotami chyb, což je ve výsledku velmi silný nástroj.

4.1.2 Navrat chyby

Jak bylo zmiňeno [], možnost v chybovém stavu vrátit i normalní hodnotu z funkce je zručná zaležitost. Navíc, je to dokonce nutná zaležitost, jelikož vnímame chybu jen jako další stav, a ne jako něco zvláštního.

Můžeme intuitivně zvolit následující syntaxi

```
return value , err ;
```

Kde value představuje proměnnou s navratovou hodnotou, a err navratovou chybu. Pak navrat value je nezměnný. Ale, musíme se zamyslet, co při navratu je chyby. Můžeme k tomu přistoupit tak, že vlastně takovy to případ existovat nebude, tedy vždy budeme muset vrátit i hodnotu. Tento způsob je pomimo všeho i zaručí, že proměnná do které se zapiše navratova hodnota nebudeme mít undefined hodnotu. I když je to skvělé chování, nemůžeme ho použít, protože máme-li být low level, tak musíme dát programátorovi i kontrolu. Nemůžeme jen tak zbytečně vnucovat instrukci. Tedy můžeme přidat symbol, např. __ definující přeskočení proměnné, a skončit s následujícím kódem

```
return __, err ;
```

4.1.3 Implementace v jiných jazycích

5 context

print, mem alloc, etc.

5.1 custom alloc