# 1 Pole

Jedna se o nejzákladnější a nejužitečnější datovou strukturu, se kterou se v programovani potkavame. V C se pole ovšem daji používat jen ve statických případech, kdy velikost můžeme stanovit ještě při kompilaci. Ovšem, i v připadech, kdy to bude dostačující, tak budeme li chtit předat pole do funkce, tak buď musime definovat onu funkci pro konkretni velikost pole, nebo využit ukazatele. První případ je použitelný jen zřídka, jelikož nepřináší abstrakci, kterou by jsme intuitivně chtěli, museli by jsme pro různé velikosti poli psat různé funkce.

To řeší předaní pole přes pointer a, po případě, ukončením pole nějakým specifickým symbolem, nebo předáním doplňujícího parametru délky. V takovém to připadě se však ztraci jaka koliv vyhoda vybraného datového typu, a vlastně i smysl onoho. Dostavame méně explicitní kod, ktery je více nadolný na chyby, jelikož compile time informaci, ktera se poji jen k jedne proměnné, rozvadime do dvou runtime.

Stanovíme tedy požadavky pro naše pole. Mělo by byt využitelné ve funkcich bez ztraty identity a při tom byt implicitním ukazatelem na počatek svých dat pří přiřazeni do pointeru. Navíc, rozšířít typ i na dynamické pole a dynamické pole variablní délky.

## 1.1 Délka pole

Zavedeme získatelnou délku pole.

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

Při předani pole do funce se tedy předa ukazatel na data a jako skryty parametr velikost pole. Pro připady, kdy neni potřeba předavat velikost, se muže použit pointer a implicitní přetypovaní.

## 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 proměnna pro jeji uchovani by neexistovala.

### 1.2.2 Pole konstantní run-time známé délky

Jedna se o analogii alokace ukazatele v C, ktery by byl využivany 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 využiva, bylo rozhodnuto ho implicitně přidat do jazyka v ramci poli. Jedna se pouze o dynamické pole s automatickou realokaci při přidani 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 Prace s polem

Protože přinašíme rozdělení pole od ukazatele, tak bych s touto myšlenkou pokračoval dal a vnímal pole jako definici prvku stejných vlastnosti, jako něco, co určuje chování pro všechny jeho prvky v jdnom mistě. 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, ktery specifikujeme jednou. Proměnná arr je však konstantní, jelikož neni 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 posypany syntaktickým cukrem a v takovém to připadě by, jaka koliv z předchozích operaci, by mohla byt vnímana jako hidden flow. My ovšem rozdělujeme mezi ukazatelem a polem na urovní typu. Tedy mužeme rozdělít chování ukazatele a pole a praci s polem vnímat vskutku jako praci se všemi proměnnými naraz, aniž by jsme něčemu ubližili.

# 2 String

V C string literaly jsou pouze hezčí verzí zapsani pole constantních charu. Vcelku, i když je to primitivní, tak zcela postačující. Problemem je zde 'absence' pole jako typu, jak již bylo zmiňovano[]. Tedy vlastně se s každym stringem pracuje jako s pointrem. Jelikož my vnímame pole jinak, tak mužeme rozvinout možnosti pole, aby nam umožňovaly ve vysledku lehčí praci i se stringy. Samotný typ pro string existovat nebude, ale bude jen podpora string literalu, ktery se při kompilaci rozloží na pole.

## 2.1 UTF-8

Bylo by vhodné rozšířít podporu literalu z ASCII na jiné kodovaní, aby byla jednoduchá cesta s připadnou jednoduchou manipulaci se složitějšími symboly. Jako takové kodovaní bych volil utf8, protože je kompatibilní s ascii, jeho blokem je byte, tedy neni zavísle na edianech a je velmi rozšířene.

Jako nejlepší možnost bych viděl zadani literalu v utf8 a compile-time vyhodnocení nevětší delky potřebné pro uložení jednoho symbolu, a nasledně konverzi na pole intu o patřičné velikosti, kde každy element bude samostatnym symbolem zakodovaným v utf8.

## 2.2 Operace

Jako jediné konvenční operace nad stringy ktere by se měly integrovat do syntaxe, bych viděl concatinaci a slice. Ostatní operace by už měly byt obsažené v standardní knihovně.

### 2.2.1 Spojení

Níc nového bych nevymyšlel, a použil operator .. jako v Lua.

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

Implementace by však nesměl obsahovat žadnou alokaci pamětí, bylo by to zavadějicí. Jelikož délky dynamickych poli jsou jejich 'současti', šlo by to využivat pro alokavana pole. Ovšem, samotná, opět, nesmi obsahovat alokace, tedy připadné vysledné pole by se muselo samostatně standardně alokavt prostředky jazyka.

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

...

### 2.2.2 Slice

...

## 2.3 Namespace

Zručný nástroj k organizací kodu. 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 rozvít.

Základním celkem bude soubor, jelikož jeto to co ve vysledku předame překladači. Překladač dostane jen jeden vstupní soubor, ktery nasledně již za pomoci prostředku jazyka umožňí 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 namespacu.

```
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 konstruktu k implementaci jiných zpusobu 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 zohlednít viditelnost jednotlivých identifikátorů.

Můžeme buď vycházet z toho, že vše je viditelńe, 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 p5istupné při importu, tak by měly být přístupně i pro další importy, jelikož jsou na stejné úrovní jako kód souboru, a nekladli jsme žádným způsobem omezení.

Tedy, navrhoval 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 stale měli možnost vytvoření připadného rozhraní z dostupných symbolu, ktere by se umisitili do jednoho souboru a zbyle by se importovali lokalně.

K označení lokalních importu bych použíl slovo local

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

# 3 Function Overloading

I když se jedna o implicitní konstrukt, ktery skryva od čtenáře pravou identitu volané funkce, tak přinaši, z mého hlediska, jednu zasadní věc, zjednošuená jmena funkcí. Tedy, zamisto vepisovani datového typu do jmena funkce, mužeme jen uvest jeji činnost. To zjednodušuje vnimani samotného programu, jelikož při praci s vlastnimi datovymi typy, ktere definuji složité objekty, jmena funkci budou už znatelnou zatíží, oproti např. maxi, maxf, maxu, kde mužeme přibližně vydedukovat typ očekavane proměnné, se tak jednoduše nevystačime. Navíc jmena samotných funkci s použitím postfixu/prefixu, které si zvolime pro identifikaci, nebudou samostatnými celky z hlediska nastroju pracujicich s kodem, tedy v zakladu samotným kompilatorem a např LSP. Tedy nebude se moct nad nimi provadět žadna kontorla, tedy např kontorla překlepu, stíženy refaktoring, horší napověda, analyza atd..

Navíc, samotná abstrakce nad konkretní volanou funkci pro četnáře není nikterak zavadějicí. Nebo spíš, je stejně zavadějicí jako for loop, ktery za misto instrukce abstraktní intrukce for provadí skoky a sem a tam. Smysl čtenář získava ze samotného nazvu funkce a vstupních proměnných, a vníma konkretní funkcí jako černou skříňku. Tedy i když ona funkce dostava int, tak nemuže vědět, že ten int neni hned první instrukci přetypovan do floatu. Tedy jediné co to ovlivní je rychlost nalezení spravné funkce při potřebě se podivát na jeji 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 nemit. Zbyva rozhodnout, zda povolit implicitni overloading, tedy jestli

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

nebo pro jiny datový typ musi byt explicitní cast

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

Zde však dochazi k zajimávemu jevu. My explicitně vepisujeme datový typ, čímž identifikujeme funkci, ovšem o tom, jestli potřebná varianta existuje, se dozvíme buď z LSP, v tomto připadě je cast jen z hlediska informace navíc (porovnavame li s implicitnim overloadingem), nebo při kompilaci, což už je trochu pozdě. Tedy jediny k čemu to muže sloužit je jako assert, kdy my víme, že chceme jit do konkretní varianty oné funkce, a pokud neexistuje, tak dostat error. Ovšem to budeme muset specifikovat u každého volani overloaded funkce, což se pře s tím, že chceme overloading hlavně z duvodu zjednodušené jmeńné stopy v kodu (nemluvě o tom, že vlastně mame tutež informaci dva x v kodu, jednou při definici, po druhy při volání).

Bylo by tedy vhodné mit implicitní overloading, ale s opci v jistých připadech specifikovat konkretní požadovaný datový typ. Zavedeme tedy přisloušnou symboliku

```
foo!();
```

Využití prapodivného symbolu v tomto připadě není zavadějicí, jelikož očekavané intuitivní chovaní vyrazu se nemění. Jedna se stale o function call, ktery nijak nemění vysledky volaní ani jeho vstupy, z hlediska čtenaře je prakticky irelavantní.

### 3.1 Implementace

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

My budeme postupovat obdobně.

Pro jmeno volané funkce najdeme všechny funckce se stejnymi jmeny a v odpovidajicím scope. Uložime do pole, kde v každem chlivečku bude struktura odkazujici se na funkci a doplňujici připadné informace popisujcí schodu. Pro zatím, neuvažujeme li polymorfismus, genericitu atd... si postačíme jen s jednou jedinou proměnnou typu int určujicí podobnost funcke našemu vzoru z volaní.

Budeme prochazet ono pole postupně funcki po funcki a buď je vyřazovat, nebo sestavovat skore podobnosti. Nakonci vybereme funkci s největším skore. Chyba nastane pokud budou dvě a vice stejná maximalní score, nebo nezbude žadna funkce se scorem.

### 3.2 Přistup jiných jazyku

V Odin je pouze explicitní, jelikož jazyk umožňuje definovat vnořené funkce ve funkcích, a tudíž rozlišení konkretní funkce, ktera se ma zavolat není trivialní. `https://odin-lang.org/docs/overview/`

Zig nema function overloading, ale podobného chování lze docilit při kompilaci za pomoci tzv. duck typing.`https://ziglang.org/documentation/master/`

## 4 Error handling

Vezmeme-li C, tak jazyk nenabizí přímý zpusob spravy chyb. Chyby se mohou řešít např. navratovou hodnotou, nějakým specifickým stavem očekavané vystupní promměnné předane přes ukazatel (většinou NULL), specialní funkci, ktera vraci poslední chybu atd... V celku je to na programatorovi, aby vytvořil nějaky system pro spravu chyb, a jestli vubec. Pak, při praci s libovolným kodem je nutne číst komentaře k funkcím, externí dokumentaci apod. Zde opět narážíme na problem, kdy duležita informace není současti strukturných elementu kodu, ke kterym by měly ruzné nastroje přístup. Take

to postrada jednotnost, kde různé knihovny mohu řešit zpravu chyb vždy jinak, a ve vysledném programu se bude muset řešít zbytečný problem, jak s tím naložit. To vše nas ve vysledku vede k myšlence o přídaní standartního systemu pro spravu chyb v našem jazyce.

Podívame-li se na jiné programovaci jazyky a jejich metody řešení spravy chyb, tak dokažeme v zasadě vyčlenít dva přistupy.

**Navratova hodnota** Chyba je vracena jako navratova proměnná nebo jeji součast. Obvykle je to spojene s možnosti navratu několika proměnných, kde se vyděluje jedno, např poslední misto, pro připadnou chybu (Odin), nebo je přímo specialní doplňujicí navratova hodnota vydělena jen pro chybu (Go). Nebo, třeba, se muže vracet struktura obsahujici jak připadnou chybu, tak i navratovou hodnotu (Rust).

Tenhle přístup je přímočaré a explicitní a dava svobodu programatorovi jak a kde s chybou naložit. Zpracovaní chyby je pak přípmou současti code-flow. Tedy chybovy stav je prakticky jen další stav programu.

**Try-Catch** Využiva se system tzv. exceptions, kde připadne chybové místo je zabalene do try bloku, a připadna chyby odchycena v catch bloku. To umožňuje např. nezatěžovát kod spravou chyb, a psat ho v try bloku tak, jako kdyby žadná chba nastat nemohla, a nasledně jakoukoliv chybu zohlednít v catch bloku.

S try-catch se většínou pojí i tzv. throw mechanizmus umožňujicí označit připadné chyby, které muže kod nějaké funkce vyvolat, a propagovat jejich ošetření do bloku, jež onu funkci volal.

- Jednotný datovy typ.
- Umožňit vytvoření množin chyb, ktere by se mohly kompozičně skladat do novyćh množín. Např mužeme vytvořit množinu chyb pro načtení souboru a množinu chyb pro zapis do souboru. Pak, budeme li chtit vytvořit funkci, ktera čte a zapisuje do souboru, tak by jsme měli mit možnost spojit oné dvě množiny do jedne.
- Definice funkce musí specifikovat, ktere chyby při jeji volaní mohou byt vracene.
- Umožnit jednoduchou propagací chyby stakem funkcí dal. Tedy zjednodušít obdobny konstrukt err = foo(); if err != null : return err;, ktery je

relativně frekventní.

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

## 4.1 Implementace

Protože nahlížet na chybu jako jen na další stav programu, i když, řekněme, specialní, je z mého hlediska přirozenější a implicitní přistup, tak se vydame cestou navratové proměnné.

Jelikož používame jen jednu navratovou proměnnou, chybu budeme chtit vracet samostatným kanálem. Ovšem, nebudeme chtit vnímat chybu jako přímo navratovou hodnotu, ktera je určena jen pro chybu, jak je tomu např. v Go. Protože pak musíme řešit po každe volaní funcke dvě vystupní proměnné. To ve vysledku povede k vytvoření buď implicitních pravidel, nebo, k mnohomluvné (verbose) syntaxi.

Představme si to na nasledujicím přikladu 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 inicializaci. Zde neni zcela zřejmé, co se ma dít, jelikož prvně provadíme definici val1 a err, a nasledně, v tymtyž scope provadíme definici val2 a opět err. Samozřejmě, je to zoohledněné pravidly jazyka, a kod je kompilovatelny, a nová definice err se neprovede, pouze val2. Ovšem, řekněme, dochází ke sporu syntaxe a semantiky, kde ze syntaktického hlediska se err chova jen jako druha navratova hodnota, ovšem ze semantického se implicitně provadí 'vyjimky' v pravidléch, jen protože je to chybova hodnota. Navíc se to kompilkuje přidaním kvalifkatoru. Budeme-li chtit označit val1 jako const ale ne err, nebo naopak, budeme li chtít mit jedno embed a druhý const, atd... To vše lze řešit na

ukor upovidané syntaxe, budeme-li chtit byt explicitní, nebo přidaním implicitních pravidel. Proto se pokusíme najit jiné řešení, ktere by více sedělo naší vizi.

K navratu chyby využijeme tedy pravou stranu příkazu. To nam ponecha příkaz, nad kterým budeme chybu odchycovat, nezměnným, a tedy budeme moct jednoduše jak měnít samotný přikaz, tak i ošetření jeho chyb, jelikož syntaktický na sobě nebudou zavíslé. Navíc to nam muže do budoucna umožnít odchycení chyby nejen z jednoho volaní funkce, ale i z libovolného vyrazu, ktery by mohl obsahovat několik volaní funkcí.

Navrholval bych nasledujíci syntaxi.

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

Kde se připadná chyba uloží do proměnné err.

Zde bych stanovíl, že nechceme zbytečně zesložiťovat datový typ chyby přidaním ruzného implicitního chování, nebo ruzných druhu konstruktu pro tvorbu chyb. Chyba je vždy jen datového typu error a chová se vždy stejně. Tedy, mužeme v takovém to připadě pominout samotnou definci err, jelikož je redundantní. Ošem, mužeme ji tam i mit.

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

### 4.1.1 Množiny chyb

Samotna chyba by měla byt jednoduše identifikovatelná přes svoje jmeno, aby ji bylo možné používát pro určení stavu. Např.

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

Chyby by měly byt shlukovane do uživatelem definovaných skupin, ktere by pak sloužili pro určení chybového rozhraní funkcí. Skupiny by měly byt shlukovatelné, jelikož funkce by měla mit možnost navracet i chyby uživaných funkci, ktere mohou byt definované samostatně, aniž by se pro ní redundantě definovaly nové chyby.

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

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

Pak ErrorSetA je množina obsahující ErrorA a ErrorB, prazdné množiny, a ErrorSetB obsahuje množinu ErrorSetA a prazdnou množinu ErrorB. Libovolná s těchto množin je identifikovatelná svým jmenem a muže byt přiřazena do datového typu error.

```
error err = ErrorSetB::ErrorB;
```

K definici chybového rozharní funkce pak použijeme nasledujicí syntaxi.

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

funkce foo pak muže vracet chyby definovane v ErrorSetB, ale take i samotný ErrorSetB, což je nutné, abychom mohli využivat i připadně prazdných množin definovaných vně jiných množin.

Protože oné množiny maji smysl jen při definici samotných funkci a my neumožňujeme definovat funkci ve funkci, tak jejich definice uvnitř funkcí je zavadějicí, a tudíž zakazana. A tedy mužeme vnímát oné množiny jako nadstavbu nad namespace pro chyby, a tedy k jejich diferenci používat stejný symbol ::, jak již bylo naznačeno viz.[...].

Toto řešení je jednoduché a relativně všestranné. Umožnuje nam např rozvít některou prazdnou množinu na plnohodnotnou, aniž by jsme rozbili kod, ktery onu množinu využival. Ovšem, ma jeden zakladní nedostatek – vracíme pouze stav. Tedy nemužeme vratít inforamci o chybě. Teoreticky je to možné řešit přidaním počtu stavu, ovšem to zdaleka není prakticke.

To nas omezuje jen při logovaní chyby, protože jinak my vždy popisujeme stav programu, ktery je nezbytny z hlediska jeho činnosti. Tudíž přidani v takových to připadech chybového stavu je vlastně nezbytné (uvažujeme-li, že chceme tento stav mit jako chybovy, obecně, samozřemě mužeme ho řešít normalní cestou). I tak mame možnosti, jak to zohlednít.

**Pomocné proměnné** V tomto připadě využijeme stavu vstupních proměnných, buď již existujicích, nebo novyćh, pomocných, k popisu chyby. Tedy např. mame li nasledujicí funkci, ktera vyhledava v souboru slovo a vraci idx symbolu, kde se to slovo začina vyskytovat

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

tak mužeme i s chybou vratit index, kde se chyba vyskytla.

**Logovaní v mistě vyskutu chyby** Je zřejmé, že přimo v mistě vyskytu chyby mame veškerou informaci k tomu, aby jsme ji popsali. Ovšem muže nam chybět kontext, kdy pro nas bude duležita informace z funkce, ktera nas volala. Tedy mužeme postupně logovnat jen informaci dostupnou nam v dany moment a ve vysledku dostat podbrobnou zpravu. Hlavním nedostatkem je však samotna nutnost logovat, protože pak musíme mit přistup k nějake přisloušné funkci, ktera by to dučlala dle našich potřeb. Což je fakticky nerealizovatelne. Ovšem dalo by se to řešít za pomoci tzv. kontextu viz[], kde by všechny funkce mohly využivat standartního zapisu do stdout, ovšem by jsme pak z vně mohly definovat kontext, ktery by všechna tato volaní přesměroval do nami zvolené funkce.

Ani jedna z možností není idelaní, ale ve vysledku je to jen něco, co slouží jako doplnění systemu. Něco, co je využiváno přímo při zpracovaní samotné chyby, a tedy neruší samotnou standartizaci, kterou jsme si kladli za cil, protože popis samotné chyby už není obecně standartiyovatelný, a tak čí onak se jedna o konkretní zaležitost.

Pokud by jsme chtěli zobecnit naš model a rozšiřit definici za pomoci struktury nebo unie, tak vlastně narazíme pření se standartizaci, protože zobecníme system natolik, že bude moct byt využívan i pro jiné věci, a take mnohy zpusoby, tudíž vlastně nami postaveny problem nevyřešíme, jen ho přesuneme jinam.

Jediny co by jsem mohli udělat, je povolit přiřazení chybam konkretních hodnot, což by mohl byt postačujicí kompromis. To umožní pak indexovat pole hodnotami chyb, což je ve vysledku velmí silný nastroj.

### 4.1.2 Navrat chyby

Jak bylo zmiňěno [], možnost v chybovém stavu vratít i normalní hodnotu z funkce je zručna zaležitost. Navíc, je to dokonce nutná zaležitost, jelikož vnímame chybu jen jako další stav, a ne jako něco zvlaštního.

Mužeme intuitivně zvolit nasledujicí 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. Mužeme k tomu přistoupit tak, že vlastně takovy to připad existovat nebude, tedy vždy budeme muset vratit i hodnotu. Tento zpusob je pomimo všeho i zaruči, že proměnna do ktere se zapiše navratova hodnota nebudeme mit undefined hodnotu. I když je to skvělé chování, nemužeme ho použit, protože mame li byt low level, tak musíme take dat programatorovi i kontrolu. Nemužeme jen tak zbytečně vnucovat instrukci. Tedy mužeme přidat symbol, např _ definujici přeskočení proměnné, a skončít s nasledujícím kodem

```
return _, err;
```

### 4.1.3 Implementace v jiných jazycích

# 5 context

print, mem alloc, etc.

## 5.1 custom alloc

# 6 Vestavená kompilcae C kodu

## 6.1 TCC

## 6.2 LLVM libclang

## 6.3 GCC

# 7 context