



Diplomová práce

Návrh jazyka odvozeného z C a implementace nástrojů pro překlad

Studijní program:

B0613A140005 – Informační technologie

Studijní obor:

Aplikovaná informatika

Autor práce:

Maxim Osolotkin

Vedoucí práce:

Ing. Lenka Koskova Třísková Ph.D.

Liberec 2025

Tento list nahradte
originálem zadání.

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

2. 4. 2025

Maxim Osolotkin

Návrh jazyka odvozeného z C a implementace nástrojů pro překlad

Abstrakt

Klíčová slova: programovací jazyk, překladač

Design of a C-derived language and compiler tools implementation

Abstract

Keywords: programming language, compiler

Poděkování

Obsah

Úvod	9
1 Kompilátor	10
1.1 Přechodná reprezentace	10
1.2 Konstrukce kompilátoru	12
1.2.1 Lexikální a syntaktická analýza	12
1.2.2 Validace a linkování AST	13
1.2.3 Vykreslení	13
1.3 Cross-Compilation	15
2 Gramatiky	16
2.1 Bezkontextová gramatika	17
3 Vývojové nástroje	19
3.1 Zvýraznění kódu	19
3.1.1 Dokumentace	20
3.2 Language server protocol	20
3.3 Debugger	21
3.3.1 Debug informace	21
3.4 Řešení	22
4 Návrh jazyka	23
4.1 Existující řešení	23
4.1.1 C++	24
4.1.2 D	24
4.1.3 Zig	25
4.1.4 Odin	26
4.2 Vstupní bod programu	27
4.3 Alokace	27
4.4 Komentáře	28
4.5 Pole	28
4.5.1 Délka pole	28
4.5.2 Typy poli	29
4.5.3 Práce s polem	30
4.6 Řetězce	31
4.6.1 UTF-8	32

4.6.2	Operace	33
4.7	Jmenný prostor	34
4.8	Systém importu	34
4.9	Přetěžování funkcí	35
4.9.1	Přístup jiných jazyků	37
4.10	Správa chyb	37
4.10.1	Definice požadavků	38
4.10.2	Implementace	38
4.11	Kontext	41
4.11.1	custom alloc	42
4.12	Compile-time exekuce	42
5	Implementace kompilátoru	44
5.1	Struktura	44
5.2	Uživatelské rozhraní	45
5.3	AST	46
5.4	Parsing	47
5.4.1	Práce s pamětí	48
5.4.2	Imports	48
5.4.3	Samotný parsing	49
5.5	Validace AST	50
5.6	Interpret	54
5.7	Generace C kódu	55
5.7.1	Souborová struktura	56
5.7.2	Globalní rozsah platnosti	56
5.7.3	Vykreslení pole	56
5.8	Vestavená kompilace C kódu	57
5.9	Správa chyb a logování	58
6	Závěr	60

Úvod

Dnes, v době, kdy člověk se spíš zeptá, umí-li to JavaScript, než, běží-li na tom Doom, C je stále C.

I když mně jazyk C imponuje, málokdy jsem se v něm našel dělat projekty. Ve většině případů jsem se uchýlil k používání C++, protože mi chyběly některé triviální moderní vlastnosti, které jsou dnes součástí mnoha jazyků (např. i pouhá namespace). Ovšem programování v C++ bylo vždy spojeno s utrpením. Tak jsem si položil otázku, zda existuje alternativa, jestli je tu něco, co by bylo jako C, ale mělo tu tak potřebnou špetku současnosti.

Odpovědí byl Odin a, popřípadě, Zig, které ve výsledku řešili můj problém, i když z části nepřímo. Ovšem, nebyl jsem v obou případech spokojen s přístupem k syntaxi, která, na rozdíl od C, šla cestou implicitnosti, ve stopách Go. Kdežto pro mě jednou z hlavních imponujících vlastností C byla i explicitní syntaxe, která i dělala dojem jazyka, kde to — co se přečte — se i vykoná.

Protože jsem ve výsledku nebyl úplně spokojen s existujícími řešeními a obecně mi přišlo, že je spíše řešena problematika náhrady C++ se zaměřením na bezpečnost, než na tvorbu jazyka, ve kterém by se dalo příjemně trávit čas při psaní vlastní aplikace, dospěl jsem k myšlence návrhu vlastního jazyka, a v důsledku psaní této práce.

Na úvod se dotknu teorie ohledně kompilátorů a programovacích jazyků. Dále představím možné nástroje/způsoby zlehčující práci s vlastním jazykem. A ve druhé polovině práce se budu věnovat samotnému návrhu jazyka, kde mimo zdůvodnění, proč je něco tak či onak, se budu odkazovat na jiné jazyky a diskutovat jejich přístup. Následně se dotknu i konkrétní implementace kompilátoru.

Nyní vás opustím a předám trpnému rodu.

1 Kompilátor

Kompilátorem nazveme program, který převádí vstupní text do výstupního textu zachovávající význam, kde oba texty jsou zapsané nějakým jazykem. Samotný proces převodu se nazývá kompilací nebo také překladem. V kontextu programovacích jazyků se jedná o převod zdrojového kódu konkrétního programovacího jazyka do jiného programovacího jazyka, nebo přímo strojového kódu.

Existence kompilátoru pro libovolný programovací jazyk je zásadní, protože z podstaty věci finálním cílem je dostat program reprezentující zdrojový kód běžící na nějakém stroji, či v nějakém virtuálním prostředí.

Za cíl se také může klást i návrh jazyka čistě pro zápis programů. Ovšem, pokud neexistuje nástroj pro překlad tohoto zápisu do jazyka, který ve výsledku je schopen být přeložen do spustitelné podoby, onen zápis nemá žádnou technickou relevanci.

Často tedy dochází k případům, kdy pojmy kompilátor a jazyk splynou nebo se zaměňují. Kdy se při použití názvu jazyka implicitně bere i na mysl konkrétní kompilátor, např. Go. Nebo kdy se naopak místo názvu jazyka používá název kompilátoru.

Protože kompilátor je jen program jako každý jiný, může být napsán v jakémkoliv již existujícím programovacím jazyce a zkompileován příslušným libovolným kompilátorem. Dokonce může být napsán v jazyce, který sám kompiluje a přeložen sam sebou. Takovýto jev se nazývá bootstrapping. To vše vede na problém o kuřeti a vejci. Zde ovšem máme jasné řešení, jelikož ve výsledku existuje stroj schopný vykonání nějakého souboru instrukcí. One instrukce vlastně tvoří jazyk, který je spustitelný, a tudíž se dá vnímat jako nejtriviálnější kompilátor pro daný stroj.

1.1 Přechodná reprezentace

Programovací jazyk slouží jako abstrakce semantiky programu a jeho skutečné podoby na konkrétním hardwaru a, po případě, operačním systému. Je zřejmé, že takto lze proložit chtěné množství vrstev abstrakcí před překladem do strojového kódu. Ovšem obecně má smysl jen jedna taková další abstrakce, kdy se jazyk přeloží nejprve do tzv. přechodné reprezentace, IR (intermediate representation), a až ona do kódu konkrétního hardwaru. Smyslem je vytvořit rozhraní pro výrobce hardwaru a jazyku. Část určená pro překlad do IR se označuje jako front-end a část převádějící IR do spustitelného kódu back-end.

Je nutno podotknout, že jak back-end, tak i front-end jsou samostatné celky, které jsou implementované pro problémy/potřeby, které chtějí řešit/naplnit. Proto jejich samotné implementace mohou také obsahovat svoje front-endy a back-endy. A tedy jak výrobce jazyků, tak i hardwaru, nemusí přímo implementovat práci IR, ale třeba využije nějakého rozhraní poskytnutého již existujícím obecným back-endem či front-endem.

Samotná IR může být reprezentována jak rozhraním a objekty či strukturami v programovacím jazyce, nebo přímo jako jazyk, tzv. mezijazyk, IL (intermediate language).

Dále se specifikuje pár ukázek IR s krátkým popisem a ukázkou reprezentace následující C funkce.

```
unsigned add1(unsigned a, unsigned b) {  
    return a+b;  
}
```

LLVM IR Forma, která se dá využívat napříč LLVM nástroji, hlavně tedy pro optimalizaci a kompilaci. Jedná se o jazyk, který je někde na pomezí C a assembleru. Může být jak v normální textové formě, nebo i přímo implementován v paměti.

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```

GCC GIMPLE Jedna z mezi frémů využívaných GCC, která je využívána při optimalizacích. Výrazy převádí do tříadresného formátu.

```
unsigned int add1(unsigned int a, unsigned int b) {  
    unsigned int _tmp;  
    _tmp = a + b;  
    return _tmp;  
}
```

Java bytecode Jedná se o instrukční sadu JVM (Java Virtual Machine). Jméno vyplývá z faktu, že každá instrukce je reprezentována jedním bytem. Bytecode je využíván JVM k JIT (viz []) kompilaci, lze ho tedy spustit, pokud existuje příslušné JVM.

```
.method public static add1(II)I  
.limit stack 2  
.limit locals 2  
iload_0  
iload_1  
iadd  
ireturn  
.end method
```

CIL Zkráceno z Common Intermediate Language. Jedná se o obdobu Java bytecode vyvinutou Microsoftem. Ke spuštění CIL je potřeba platforma, která podporuje nějakou implementaci tzv. Common Language Infrastructure, zkráceně CLI, jako je .NET.

```
.method uint32 add1(uint32 a, uint32 b) cil managed {  
    .maxstack 2  
    ldarg.0  
    ldarg.1  
    add  
    ret  
}
```

C I samotné Céčko může sloužit jako IR, i když nebylo přímo pro to navrženo. Jedná se o jazyk blízký k assembleru a využívaný v některých různých operačních systémech. Existuje pro něj jak velký výběr kompilátorů pro různé platformy, tak i jiných vývojových nástrojů.

1.2 Konstrukce kompilátoru

Samotný překlad se může rozdělit do pár základních kroků. Nejprve je provedena lexikální a syntaktická analýza a čirá sled textu je převedena na abstraktní reprezentaci. Následně je tato reprezentace zvalidována dle příslušných semantických pravidel. Validní reprezentace pak převedena do vybraného IR či přímo do spustitelné podoby. Viz obr[1].

Samozřejmě, překlad může obsahovat mnohem více kroků, např. po validaci může následovat optimalizace. Ovšem, vytknuté tři kroky jsou nezbytné pro jakýkoliv překlad.

1.2.1 Lexikální a syntaktická analýza

Cílem je převést zdrojový kód dle gramatiky jazyka do nějaké abstraktní datové struktury v paměti kompilátoru. Taková struktura je často reprezentována stromem, jelikož je to nej přirozenější vyjádření gramatiky, a nese ustálený název AST (abstract syntax tree).

Zde se nabízí zřejmá a 'dobrá' abstrakce lexikální a syntaktické analýzy. Kde modul lexikální analýzy se stará o zpracování vstupního textu a převádí slova na reprezentaci v paměti kompilátoru, která se nazývá token. Syntaktická analýza pak bude již se slovy pracovat jako s abstraktními celky. Lexikální část se často nazývá lexer a syntaktická parser.

Zároveň se smysluplná abstrakce nabízí i mezi samotnou gramatikou a lexémem-parsrem. Možnost vyjádření jazyka za pomoci jistého standardu gramatiky (viz.[2]) umožňuje i existenci příslušných nástrojů napomáhajících při generaci AST.

Mezi takové nástroje patří třeba YACC a ANTLR.

YACC Neboli Yet Another Compiler-Compiler. Nástroj umožňující parsing na úrovni gramatiky jazyka (pro bližší představení samotného formátu gramatiky viz. []). Vše probíhá v jakémsi dialektu C, kde se jednotlivým syntaktickým celkům dají přiřadit funkce, jež budou vykonány při jejich rozpoznání, jednotlivým za pomoci asociovaných pravidel. Jako lexer YACC využívá uživatelem definovanou funkci, standardně se využívá nástroj Lex. YACC ve výsledku generuje C kód (hlavně `yyparse` funkci), který se již používá v samotném kompilátoru.

ANTLR Neboli Another Tool for Language Recognition. Jedná se o nástroj umožňující generaci parseru z gramatiky. Kromě generace samotného parseru ANTLR vygeneruje i tzv. procházeče stromu, které umožní aplikaci vykonávat vlastní kód. Základním jazykem, pro který ANTLR generuje parser, je Java, ale umožňuje generaci i do jiných jazyků, jako C#, Python, Go atd.

1.2.2 Validace a linkování AST

Cílem je provést semantickou kontrolu AST. Kontrola se bude lišit od jazyka k jazyku v závislosti na striktnosti jeho pravidel. Může se zde provést ověření existence příslušných deklarací vyskytujících se proměnných v příslušných jmenných prostorech; kontrola datových typů proměnných a výrazů; nalezení vhodné funkce v případě function-overloadingu; a podobně. Kromě validace se zároveň vyskytujícím se symbolům spojí příslušné definice, je-li to třeba z hlediska navrženého AST. Tedy například uzlu reprezentujícímu proměnnou se přiřadí reference na její definici.

1.2.3 Vykreslení

Na konec se AST má vykreslit do finální podoby. Tedy, obecně by pro každý node měla existovat posloupnost instrukcí, které by to činily. Nejprirozenější způsob je existence funkce pro každý typ uzlu AST, kdy by se volala vždy příslušná funkce při procházení stromu. Ovšem, je to ve výsledku jen obyčejný program, takže implementace může být vždy přizpůsobena konkrétnímu problému.

Vykreslení lze rozdělit v závislosti na finálním produktu.

Kod Výsledkem je kód v jiném jazyce, tedy buď v IL, nebo přímo strojový kód. Zde buď kompilátor končí a očekává se, že výsledný kód se přeloží již jiným nástrojem do spustitelné podoby. Může to být i v podobě skriptu, který je pak součástí většího celku, jako je game engine. Nebo se může jednat i o tzv. transkripci, jako v případě TypeScriptu.

Interpretace Za místo získání nějakého výsledného kódu můžeme rovnou každý uzel interpretovat. Tedy, za místo napsání funkce, jejíž výstupem by byl text v jiném jazyce, lze v ní implementovat rovnou chování uzlu, jeho logiku. Takovéto kompilátory se z pravidla označují za interprety.

JIT I když se formálně jedná o první kategorii, tak samotná koncepce je významná a stojí za samostatnou zmínku. JIT stojí za Just In Time. Jedná se o způsob kompilace, kdy je generovaná IR reprezentace, která se pak předá programu

tzv. JIT kompilátoru, který už přeloží IR do konkrétního strojového kódu mašiny, na které běží. Důležitým bla bla bla..

1.3 Cross-Compilation

Někdy je vhodné přeložit program do strojového kódu jiného hardwaru, než na kterém běží kompilátor. Tomuto procesu se říká cross-compilation. Může to být v případech, kdy se program vyvíjí na vysoce výkonném stroji obsahujícím všechny potřebné nástroje pro rychlou a pohodlnou práci, a výsledný software má být určen jinému stroji neobsahujícím takovou infrastrukturu. Příčinou může být operační systém, nebo i samotný hardware stroje.

Také se jedná o případy, kdy se program kompiluje i pro jiné operační systémy, než na kterém je vyvíjen. Například, Doom byl vyvíjen na NeXT počítači s operačním systémem NeXTSTEP, zatímco byl spuštěn pod MSDOS.

Je zřejmé, že o cross-compilaci má smysl mluvit pouze v případě kompilace do strojového kódu. V jiných případech se jedná o kód, který je mezivýsledkem a jeho spuštění závisí na jiném nástroji, který sám musí být přeložen pro odpovídající stroj.

2 Gramatiky

Při návrhu programovacího jazyka hraje důležitou roli samotná syntaxe, která ho z velké části definuje. Syntaxe je totiž jakýmsi rozhraním mezi člověkem a jazykem, obzvlášť v případě programovacích jazyků, kde se v textových editorech či vývojových prostředích běžně různé části syntaxe různě zvýrazňují. Je tedy vhodné mít možnost ji nějakým způsobem formálně popsat, jak z teoretického hlediska, tak i z praktického, kdy definice gramatiky jazyka se může používat v různých nástrojích, např. jak již bylo zmíněno, syntaktické zvýrazňování.

K definici syntaxe jazyka slouží tzv. formální gramatika. Formální gramatiku můžeme definovat následovně:

Definice 2.1. Formální gramatika G je čtveřice (σ, V, S, P) , kde:

- σ je konečná neprázdná množina terminálních symbolů, tzv. terminálů.
- V je konečná neprázdná množina neterminálních symbolů, tzv. neterminálů.
- S je počáteční neterminál.
- P je konečná množina pravidel.

Terminály jsou dále nedělitelné symboly jazyka. Jsou to například klíčová slova nebo jednotlivá písmena sloužící pro definici proměnných. Neterminaly jsou pak jakési proměnné, které se dále dělí na další terminály nebo neterminaly. Neterminál může například představovat binární operátor, který pak bude definován jako množina již terminálních symbolů představujících jednotlivé binární operátory. Prázdný symbol se označuje jako ϵ .

Obecně pravidlo gramatiky můžeme vyjádřit jako zobrazení: ¹

$$\alpha \rightarrow \beta, \text{ kde } \alpha \in (\Sigma \cup V)^* V (\Sigma \cup V)^*, \text{ a } \beta \in (\Sigma \cup V)^*$$

Tedy vzorem je posloupnost terminálů a neterminálů obsahující alespoň jeden neterminál. Obrazem pak je libovolná posloupnost terminálů a neterminálů.

Gramatiky lze členit na základě striktnosti pravidel dle tzv. Chomského hierarchie.

Definice 2.2. Necht $G = (\sigma, V, S, P)$ je gramatika, pak:

¹Hvězdíčka (*) představuje symbol libovolného opakování výrazu, a to i žádného.

- G je gramatika typu 0 nebo take neomezená gramatika právě tehdy, když ...
- G je typu 1 nebo take kontextová gramatika právě tehdy, kde pro každé pravidlo $\alpha \rightarrow \beta$ z P platí $|\beta| \geq |\alpha|$ a zároveň pravidlo $S \rightarrow \epsilon$ se nevyskytuje na pravé straně.
- S je typu 2 nebo take bezkontextová gramatika právě tehdy, když pro každé pravidlo $\alpha \rightarrow \beta$ z P platí $|\alpha| = 1$. Neboli, že α je pouze neterminal.
- P je typu 3 nebo take regulární gramatika právě tehdy, když každé pravidlo z P je v jedné z forem:

$$A \rightarrow cB, A \rightarrow c, A \rightarrow \epsilon,$$

kde A, B jsou libovolné neterminaly a c je terminal.

Z hlediska programovacích jazyků prakticky se lze omezit na gramatiky bezkontextové.

2.1 Bezkontextová gramatika

Bezkontextovou gramatiku, kromě definice uvedené výše, lze také definovat z hlediska samotných pravidel, což bude názornější pro navazující text.

Definice 2.3. Gramatika je bezkontextová právě tehdy, když pro každé pravidlo z P platí buď

$$S \rightarrow \epsilon,$$

nebo

$$\alpha A \beta \rightarrow \alpha \gamma \beta,$$

kde

$$A \in N, \alpha, \beta \in (N \cup \Sigma \setminus \{S\})^* \text{ a } \gamma \in (N \cup \Sigma \setminus \{S\})^+$$

Například, pravidlo pro sestavení goto výrazu v jazyce C může být vyjádřeno ve volné formě třeba následovně

$$\text{goto} \rightarrow \text{'goto' identifier ';'}$$

Pravidlo definuje nonterminal `goto` jako možné slovo počínající goto, čířym textem následujícím nonterminalem `identifier`, který je definován v nějakém jiném pravidle, představujícím identifikátor. To vše je zakončeno terminálem představujícím symbol středníku.

Definuje-li se pak třeba `identifier` za pomoci regulárního výrazu následovně

$$\text{identifire} \rightarrow [\text{a-zA-Z}]^+$$

`goto` pravidlo bude třeba generovat slova jako

$$\text{goto FooLabel ;}$$

```
goto Me ;  
goto UnhandledException ;
```

Je zřejmé, že zápis pravidel může být různorodý. Pro sjednocení zápisu existují různé standardy. Může se vytknout několik relevantních a stručně předvést na příkladu s `goto` příkazem.

BNF zkraceno od Backus–Naur forma.

```
<goto-stmt> ::= "goto" <identifier> ";"  
<identifier> ::= <letter> <letters>  
<letters> ::= <letter> <letters> | \epsilon  
<letter> ::= "a" | "b" | ... | "Z"
```

EBNF rozěřená (Extended) Backus-Naur forma. Existuje několik verzí a má ISO Standard.

```
goto-stmt = "goto", identifier, ";";  
identifier = letter, { letter };  
letter = "a".."z" | "A".."Z";
```

YACC notace bližší seznámení s YACC'em může být naleznuto zde [\[\]](#).

```
goto_stmt : KW_GOTO identifier ';' { .. };
```

Složené závorky představují místo, kam se umísťuje C kód, který se má provést při parsingu oných elementů. Neterminaly z příslušných pravidel jsou přístupné za použití symbolu \$. Například \$ označuje proměnnou samotného pravidla, \$1 proměnnou prvního terminálu či neterminálu (`KW_GOTO` v případě pravidla `goto`), \$2 respektive druhého atd.

Definice `identifier` je pak součástí jiného programu zvaného Lex, na výstup kterého YACC spoléhá.

```
identifier : [A-Za-z]+
```

ANTLR notace bližší seznámení s ANTLR'em může být naleznuto zde [\[\]](#).

```
goto_stmt : 'goto' identifier ';' ;  
identifier : [a-zA-Z]+ ;
```

3 Vývojové nástroje

Po mimo samotného překladače se k práci s programovacím jazykem běžně využívají různé nástroje usnadňující práci.

Základem je samotný textový editor, bez kterého by nebylo možné samotný kód v celku psát. Samotné editory pak, většinou za pomoci pluginů, umožňují přidávat podporu různých jazyků. Mezi takové populární editory patří například VS Code nebo Vim/Nvim. Lze jich tedy využít jako platformu pro tvorbu jakéhosi IDE pro vlastní jazyk. Tento text se omezí pouze na VS Code a Nvim.

Pluginy, nebo také Extensions, se ve VS Code dají standardně psát za pomoci TypeScriptu či JavaScriptu. Jako v prohlížečích, je zde i možnost využití WebAssembly. Lze tedy využít i jiný jazyk, který by šel do WebAssembly zkompileovat, jako třeba Rust nebo C++. Ke komunikaci s editorem je zde VS Code API, které umožňuje přístup k elementům uživatelského rozhraní editoru, poslouchání různých eventů, přístup k debuggeru atd. Všechny pluginy se pak dají nahrát do jednotného oficiálního marketplace, kde budou dostupné uživatelům a umožní automatické aktualizace.

V případě Nvim je zde kromě klasických možností využití Vimscriptu, jak v případě Vim, dostupná možnost skriptování za pomoci integrovaného Lua script engine. Celé Vim API je pak dostupné skrz Lua. Lze tedy přímo přistupovat k bufferům a měnit rozhraní celého editoru. Pluginy jsou ve své podstatě jen zdrojové kódy, které jsou načtené v konfigu. Většinou za pomoci nějakého packer-manageru, který umožní načtení složky s pluginem jedním řádkem, a to i třeba z git repozitáře. Klasickým způsobem distribuce pluginů je pak git repozitář se samotným kódem pluginu, link na který uživatel předá packer-manageru. Takto uživatel bude moci i stáhnout updaty, jestli bude chtít.

3.1 Zvýraznění kódu

Za základní potřebnou vlastnost se může klást zvýraznění kódu, která je prakticky zřejmostí.

K definici vlastního zvýrazňování se v případě VS Code využívá TextMate, který umožňuje definovat vlastní gramatiku v JSON souboru za využití regulárních výrazů. V případě Vim se používá vlastní formát, který také umožňuje využití regex.

Oba tyto přístupy využívají tak či onak prohledávání a parsování zdrojového kódu

pro zvýraznění. Existuje však i jiný přístup, který je v praxi rychlejší a přesnější, a to za využití LSP. Většinou totiž i tak máme aktivní LSP, které nám zajišťuje např. doplňování slov, a tudíž už máme informaci o všech symbolech a jejich roli v jazyce. Oba vybrané editory mají vestavěnou podporu LSP.

3.1.1 Dokumentace

Po mimo zvýraznění kódu v editorech je někdy třeba mít možnost zvýraznění kódu ve statických dokumentech. Například jako dokumentace, která je nezbytná pro popis semantiky jazyka uživateli.

V takovémto případě lze využít například nástroje Shiki. Jedná se o JavaScript knihovnu, která využívá TextMate gramatiky k generaci zvýrazněného výstupu. V základu Shiki umí generovat výstup jako HTML. Klasické užití je pak napsání drobného skriptu v NodeJS, který by procházel HTML dokument a nahrazoval vybrané elementy, např. code, výstupem z Shiki. Pak výsledný HTML dokument neobsahuje žádný JS run-time kód. Obdobným způsobem je generována dokumentace obsažená v příloze.

V případě Vim syntaxe je zde možnost využití jeho samotného ke generování HTML z kódu. Je zde opět nutnost napsání nějakého skriptu, který by automaticky procházel HTML soubor a přepisoval ho.

```
vim -c 'syntax_on' -c 'T0html' -c 'wq' myfile.html
```

Bohužel, zde nejsou výrazné nástroje, které by umožnily využití Vim syntaxe pro generování zvýrazněného výstupu, jako v případě TextMate gramatiky. Pro využití v HTML dokumentech je zde jen opce využití vim.js, tedy portu Vim pro prohlížeče, který by v run-time mohl zvýrazňovat kód. Ovšem využití tohoto řešení jen pro zvýraznění kódu je zbytečně náročné.

3.2 Language server protocol

Language server protocol, zkráceně LSP. Jedná se o protokol využívaný pro komunikaci language serveru a klienta, kterým je třeba IDE nebo textový editor, kde language server předává informaci klientovi o textu z pohledu jazyka. Smyslem je nabídnout rozhraní mezi programem nabízejícím syntaktickou a semantickou informaci o kódu a vývojovým nástrojem.

V základu k implementaci lze použít část kódu ze samotné implementace kompilátoru, či dokonce celé moduly. Protože práce serveru je od části shodná až do fáze vykreslení. Ovšem, v případě kompilátoru je možnost ukončení kompilace při první chybě. V případě LSP by měl program umět kompilovat i neúplně správný syntaktický kód, a to i semanticky, a dávat výsledky o tom, co se podařilo převést do AST. Navíc, LSP by měl fungovat v reálném čase obnovujíc informaci o kódu po každém inputu uživatele. Tvorba LSP tedy není triviálním úkolem i za podmínky existence

kompilátoru, jelikož jak kompilátor, tak i LSP by měly fungovat co nejrychleji, ovšem jejich potřeby se protirečí.

bla bla bla

3.3 Debugger

Finalním nástrojem při tvorbě programů je debugger. Zde opět lze využít vybraných textových editorů jako platformy. Ovšem psaní vlastního debuggeru není zcela žádoucí, jelikož je to další aplikace, která se bude muset s jazykem vyvíjet a udržovat. Je výhodnější využít již existujících řešení skrze nějaký dostupný interface.

3.3.1 Debug informace

Debug informace je veškerá informace, která není obsažená ve spustitelném souboru, ale je napomocná debuggeru k propojení zdrojového kódu a konečných instrukcí. Debugger pak může umět například krokovat zkompileovaný program ve zdrojovém kódu, zobrazovat hodnoty proměnných atd.

Uvažujeme-li jazyk, který se bude kompilovat do strojového kódu, tak stačí mít program jako spustitelný soubor a k němu vygenerovanou debug informaci. První problém řeší samotný kompilátor, a tedy zbývá vyřešit otázku generace debug informace.

V zásadě existují dva hlavní formáty využívané moderními debuggery, a to PDB a DWARF.

PDB zkraceno z Program Database. Jedná se o soubory převážně využívané Microsoftem, například pro debugování ve Visual Studio. PDB vnitřně, pro definici samotných debug symbolů, využívá formátu CodeView. V rámci Windowsu existuje API, které umožňuje získání informací z PDB souboru bez znalostí formátu.

DWARF zkraceno z Debugging With Arbitrary Record Formats. Formát využívaný například GDB a LLDB. Převažně pro programy na Linux a macOS. Často se používá v rámci ELF souborů. Je standardně vestavěn do spustitelného souboru.

Samou přímočarou možností je vlastnoruční generace těchto souborů. Naštěstí některé backendy umožňují generaci oněch symbolů.

V případě LLVM IR lze třeba definovat podrobnější informace o původním kódu za pomoci maker `#dbg_value`, `#dbg_declare` a `#dbg_assign`. Může to vypadat následovně

```
%i.addr = alloca i32, align 4
#dbg_declare(ptr %i.addr, !1, !DIExpression(), !2)
; ...
!1 = !DILocalVariable(name: "i", ...) ; int i
!2 = !DILocation(...)
```

Kde první řádek představuje deklaraci proměnné `i` typu `int32_t`. Následující pak přidává oně deklaraci metadata a na dalších řádcích se některá metadata specifikují. Lze to použít jak pro generaci PDB, tak i DWARF.

Nebo, například při použití C jako IL, lze využít vybraného kompilátoru umožňujícího generaci potřebného formátu. Pro mapování zdrojového kódu na C kód pak lze využít direktivy `#line`, která umožňuje specifikovat číslo řádku a název souboru. Direktiva však funguje jen na bezprostředně následující řádek kódu, což lze řešit generací kódu obecně bez nových řádků a přidáváním je vždy s použitím oné direktivy.

3.4 Řešení

Pokud je možné generovat debug informaci se spustitelným souborem, lze využít libovolného již existujícího debuggeru podporujícího formát oné informace.

Protože VS-Code má standardně implementované rozhraní pro debuggery, lze vytvořit vlastní konfiguraci pro již existující debugger plugin, kde se zamění příkaz kompilace. A po případě se to dá zabalit i do samostatné extension.

Nvim nemá standardní interface pro debugger, takže v případě každého debugger pluginu je konfigurace individuální, jestli je vůbec v konkrétním případě dostupná. Vždy ale lze udělat fork ...

4 Návrh jazyka

Prvně bych vytvořil nějakou představu o vizi jazyka a motivaci. Protože v zásadě cílem je přijít s moderní obdobou C, tak by bylo k věci jim i začít.

V jazyce C se mi v zásadě líbí jeho jednoduchost, co se napíše, to v podstatě i platí. Neprovádí se v pozadí tučt instrukcí — neobsahuje skrytý tok řízení. Má explicitní syntaxi, která se dá číst. Nabízí možnost přímé a volné práce s pamětí, která je ve většině jazyku potlačena kvůli bezpečnosti, ale dodává tu špetku hravosti, kdy se problém vyřeší chytrou, nebo i hloupou, interpretací paměti.

Je to jazyk, ve kterém je zajímavé programovat, i když ne vždy je vhodným nástrojem pro produkční řešení. Má ale pár zásadních nedostatků, které mně většinu odradí od jeho využití. Například: samotné sestavení programu je příšerné, neustálá duplicita informace v deklaracích a oddělených definicích, problémy s duplicitními jmény při použití knihoven, makra atd.

Vycházejíc z toho bych tedy viděl proceduralní systemový general purpose jazyk umožňující robustní a explicitní compile-time vyhodnocení. Jazyk, který by umožňovat jednoduchou a neomezenou manipulaci s pamětí. Byl čitelný sam o sobě i na úkor osvědčeným postupům. Interpretace kódu po přečtení by měla co nejvíce odpovídat skutečnosti, například: deklarace proměnné by neměla být ve vchozím případě konstatní, protože po přečtení kódu, který nespecifikuje vlastností deklarce je přirozenější se domnívat, že žádných vlastnotí nenabyva, než že jsou nějaké standardní skryté. Syntaxe by měla být srozumitelná intuitivně i bez znalostí gramatiky. Tedy, v zásadě, akce spíš interpretovaný slovy než symboly.

Jazyk by neměl přímo plnit nějaký cíl, nebo být určen speciální sféře. Jedná se o tvorbu nástroje pro moje vlastní užití, jazyk, ve kterém bych mohl psát vlastní projekty. A tedy i pro jedince, které by měli stejný náhled na věc.

4.1 Existující řešení

Existují jazyky v tom či jiném smyslu řešící muj problem. Některé z mého hlediska jsou vskutku skvělé a zajímavé. Ovšem, nenašel jsem se v nich na tolik, abych pak něco nevyčítal. Přijde mi tedy vhodným je představit a vytknout věci, které jsou odlišné od moji představy.

4.1.1 C++

besprostřední následník C, který je i často s ním nerozlučně spojen jako C/C++. C++ ponechává v základu C s drobnými změnami a staví na tom v základu za pomoci standardní knihovny. C++ tedy dědí i špatné vlastnosti C, jako jsou například makra a system importu. Nové vlastnosti často nejsou částí jazyka, ale objekty právě standardní knihovny. Mezi takové patří i základní věci jako pole.

Jazyk obsahuje mnoho různorodých konceptů umožňujících řešit problémy paradigmaticky různými způsoby. I když se to může vnímat jako výhoda, a člověk si může vybrat nějakou podmnožinu vlastností, která mu vyhovuje, tak jen zřídka veškerý kód je psán jedním člověkem. Například při práci s knihovnou, která řeší problémy objektově se musí potýkat člověk nepíšící kód objektově. To vede k velmi nekonzistentnímu kódu. I samotná standardní knihovna, která implementuje hodně zásadních vlastností jazyka využívá metaprogramování a objekty.

Syntaktický je pak jazyk příšerný, protože po mimo míchání samotných přístupů k programování se míchá i C a C++ kód. To zesložituje vnímání a čitelnost kódu, protože není moc jasné co se má dít, a jak C++ objekty nakládají s původními datovými typy a jak je interpretují.

Ukázka syntaxe – vypočet

```
#include <memory>
class widget {
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000); // lifetime automatically tied to enclosing scope
                      // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

4.1.2 D

Jazyk syntaktický blízký C, zjednodušuje koncepty C++ a zbavuje se přímé návaznosti na C. Tedy importování system je řešen za pomoci modulu, podobně jako v Java. Žádná makra, spoleh na compile-time exekuci a genericitu zjednodušit koncepty C++ a zbavit Zjednodušený Syntaktický blízký jazyk k C.

Ovšem, je to objektový jazyk s hidden-flow elementy (např try-catch), který jde spíš ve stopách C++ než C. Navíc obsahuje garbage collector, který sice lze zakázat, ale některé vnitřní operace ho stále budou používat.

Ukázka syntaxe – paralelní inicializace poli

```
void main() {
    import std.datetime.stopwatch : benchmark;
    import std.math, std.parallelism, std.stdio;

    auto logs = new double[100_000];
    auto bm = benchmark!({
        foreach (i, ref elem; logs)
            elem = log(1.0 + i);
    }, {
        foreach (i, ref elem; logs.parallel)
            elem = log(1.0 + i);
    })(100); // number of executions of each tested function
    writefln("Linear_init: %s msecs", bm[0].total! "msecs");
    writefln("Parallel_init: %s msecs", bm[1].total! "msecs");
}
```

4.1.3 Zig

Jedná se o relativně moderní, jednoduchý, procedurální jazyk jdoucí ve stopách C. Jazyk staví na compile-time exekuci mataprogramování nabízející široké spektrum možností jejího využití. Neobsahuje skrytý řídicí tok programu. Práce s pamětí je manuální. Překladač obsahuje varianty sestavení, které umožňují provádět doplňující bezpečnostní kontroly jak při kompilaci tak i v run-timmu. Navíc je možné sestavovat program cross-kompilací.

Ve své podstatě se jedná o velmi blízko jazyk svým představám až na některé drobné věci jako například, že ukazatel je standardně nenulovatelný. Hlavní výtku mám k syntaxi, která není moc explicitní a využívá až moc symbolů.

Ukázka syntaxe – parsing celých čísel

```
const std = @import("std");
const parseInt = std.fmt.parseInt;

test "parse_integers" {
    const input = "123_67_89,99";
    const ally = std.testing.allocator;

    var list = std.ArrayList(u32).init(ally);
    // Ensure the list is freed at scope exit.
    // Try commenting out this line!
    defer list.deinit();

    var it = std.mem.tokenizeAny(u8, input, "_", ",");
    while (it.next()) |num| {
        const n = try parseInt(u32, num, 10);
        try list.append(n);
    }

    const expected = [_]u32{ 123, 67, 89, 99 };

    for (expected, list.items) |exp, actual| {
```

```

        try std.testing.expectEqual(exp, actual);
    }
}

```

4.1.4 Odin

Obdobně jak Zig se jedná o moderní analog C. Proceduralní jazyk s jednoduchou a minimalisticou syntaxí. Nabízí pár zajímavých jako třeba kontext viz^[1]. a vestavené aritmetické programování s poli, a k tomu i maticový typ, který umožňuje třeba násobení matic, matic a poli a pod.

Jako nedostatek bychom vytknuli absenci explicitní compile-time exekuce, která je fakticky možná jen při definici konstant. Syntakticky jazyk je relativně implicitní a podobá se jazyku Go. Syntaxe mi přijde více intuitivní než v případě Zigu.

Ukazka syntaxe – programování s poli

```

package main
import "core:fmt"

main :: proc() {
    Vector3 :: distinct [3]f32
    a := Vector3{1, 2, 3}
    b := Vector3{5, 6, 7}
    c := (a * b)/2 + 1
    d := c.x + c.y + c.z
    fmt.printf("%.1f\n", d) // 22.0

    cross :: proc(a, b: Vector3) -> Vector3 {
        i := a.yzx * b.zxy
        j := a.zxy * b.yzx
        return i - j
    }

    cross_explicit :: proc(a, b: Vector3) -> Vector3 {
        i := swizzle(a, 1, 2, 0) * swizzle(b, 2, 0, 1)
        j := swizzle(a, 2, 0, 1) * swizzle(b, 1, 2, 0)
        return i - j
    }

    blah :: proc(a: Vector3) -> f32 {
        return a.x + a.y + a.z
    }

    x := cross(a, b)
    fmt.println(x)
    fmt.println(blah(x))
}

```

4.2 Vstupní bod programu

Obvykle vstupním bodem programu ve vyšším programovacím jazyce je nějaká tzv. main funkce. Taková funkce může mít za úkol předání vstupních argumentů programu a oddělení globálního scope.

Samotný koncept mi přijde obskurtním:

- Prvně, main funkce zvyšuje indenci kódu a zesložituje strukturu programu bez možnosti se tomu vyhnout. Když, například, uživatel bude chtít začít v lokálním scope, protože je to to, co se mu líbí na main funkci, tak to může udělat přímo za pomoci odpovídajícího syntaktického konstruktu. Dokonce, když to udělá, tak jasně dá čtenáři znát svou myšlenku.
- A za druhý, ruší chápání pořadí vykonání instrukcí. Instrukce se totiž běžně mohou objevit i v globálním scope. Ovšem, protože z main funkce nelze nijak skočit do globálního scope, ale stále se jedná o místo, kde by měl program začít své konání, tak není jasné jak, a jestli vůbec, se provedou globální instrukce.

Jako vstupní bod programu jsem tedy zvolil počátek souboru, obdobně jako v Pythonu, nebo, když mám vybírat z C-like jazyku, jak v HolyC.

4.3 Alokace

Dynamickou alokaci bych nevnímal jako funkci, operator, nebo výraz, ale jako samostatný celek, který by sloužil alternativou při přiřazení. Tedy, že by přiřazení buď bylo alokací, nebo výřezem. Smyslem je vždy zaručit, že dynamicky alocovaná paměť bude vždy přiřazena proměnné.

Syntaxi jsem zvolil následující

```
int^ ptr = alloc 8; // alokuje 8 bytu
int^ ptr = alloc int[8]; // alokuje 8 * sizeof int
```

Při alokaci při deklaraci lze vynachát datový typ na pravé straně, jestli má být schodný s tím na levé. Formálně je to možné, protože alloc je alternativní rvalue, oproti výrazu new v C++ či D, který by se měl řídit pravidly výrazu.

Následující řadky by tedy vyjadřovali to same.

```
int^ ptr = alloc int[8];
int^ ptr = alloc [8];
```

Navíc bych umožnil přímou inicializaci za pomoci symbolu :, který se využívá v jazyce jako počátek příkazu. Následující kód inicializuje alokuje velikost odpovídající int a a bude inicializována na 1.

```
int^ ptr = alloc : 1;
```

4.4 Komentáře

V C, nebo například i v C++ a D nelze vnořovat blokové komentáře `/**/`. Není to vyznáčným nedostatek, ale stále nepříjemný, pokud se komentuje něco, co už obsahuje komentář. Navíc se je to nekonzistentní s řádkovými komentáři, které se mohou vnořovat.

Volil bych následující syntaxi.

```
\{  
    \{ comment \}  
\}
```

4.5 Pole

Jedna se o nejzákladnější a nejužitečnější datovou strukturu vyskytující se v programování. V C se ovšem pole dají používat jen ve statických případech, kdy velikost lze stanovit ještě při kompilaci. Ovšem, i v případech, kde je to dostačující, tak při využití pole jako argumentu funkce se funkce buď musí definovat pro konkrétní velikost pole, anebo obecně pro ukazatel.

První případ je použitelný jen zřídka, jelikož nepřináší abstrakci, která se intuitivně pojí s vybraným datovým typem. Muselá by se vytvořit pro různé velikosti poli samostatná funkce. To se může řešit předáním pole přes pointer a buď 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 konceptuální smysl onoho. Kod je ve výsledku méně explicitní, a navíc více nadolný na chyby, jelikož compile-time informaci, která se pojí jen k jedné proměnné, se rozvádí do dvou run-time.

Stanovil bych tedy některé základní požadavky pro 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.

4.5.1 Délka pole

K získání délky pole jsem zavedl následující syntaxi.

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

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

4.5.2 Typy poli

Po mimo klasického rozdělení poli na statická a dynamická, bych chtěl umožnit jejich dělení v závislosti na variabilitě délky. To by umožnilo vytvářet funkcím více specifická rozhraní pro práci s poli. Například by `int[const]` by vyznačovalo konstantní délku a funkce, která by neměla potřeby měnit velikost vstupního pole by pak mohla přijmout jak statické, tak i dynamické pole.

Navíc bych chtěl integrovat array list do jazyka v rámci poli, jelikož je to často využívaná struktura. Array list bych viděl jako automaticky rozšiřovatelné pole tak, aby vždy šlo zapsat na zvolený index.

Pole konstantní compile-time známé délky

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

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

by vytvořilo pole o délce 8. Spočtená délka by se vždy dosazovala compile-time, reálná proměnná by se pro její uchování negenerovala.

Pole konstantní run-time známé délky

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

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

Tedy

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

by alokovalo pole o délce 8 na heapu a vygenerovalo by příslušnou proměnnou pro uložení délky někde v paměti programu. Pole by, samozřejmě, nešlo realokovat, jelikož délka pole je obecně run-time známa, a tedy není možnosti ověřit při kompilaci její neměnnost.

Pole variabilní run-time známé délky

Analogie využití ukazatele jako pole v C.

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

Tedy

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

by alokovalo pole o délce 8 na heapu a vygenerovalo by příslušnou proměnnou pro uložení délky někde v paměti programu. Pole by šlo realokovat.

Array List

Šel by vytvořit následovně

```
int[] arr;
```

nebo se specifickou počáteční delkou, v tomto případě 8.

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

Volba kvalifikatoru

Lze také vnímat dynamické pole za výchozí, a ne array-list. Pak by se využil kvalifikator při inicializaci array-listu, pole variabilní run-time známé délky by se inicializovalo bez kvalifikatoru. Takovým kvalifikátorem by mohl být třeba `auton` od slova `autonomus`.

Tato varianta se protířečí s základní představou o jazyce. Na druhou stranu se ale zbavuje implicitní alokace, protože ona varianta se již musí konkrétně zvolit.

Příklady jiných jazyků

C++ ponechává vše jak v C. Řeší vše za pomoci kontejneru definovaných v `std` knihovně. Nema však analog dynamického pole, ale nabízí `std::span`, který může sloužit jako interface pro souvislý kontejner, nemůže však sloužit přímo jako kontejner pro data.

```
std::array<int, 8> arr;  
std::vector<> arr;
```

D obohacuje jak statické, tak i dynamické pole o délku.

```
int[8] arr;  
int[] arr = new int[8];  
// v obou případech délka dostupná jako  
arr.length;
```

Odin má k dispozici statická pole, slice, které se využívají obdobně jako `std::span`, ale umožňují alokaci, a tzv. dynamická pole, které je analogem array listu pro jehož deklaraci se využívá kvalifikator na místě délky. Všechny mají dostupnou funkci `len` vracející jejich délku a dynamické pole může využít funkce `cap` k získání reálné alokované délky.

```
arr: [5]int // static  
arr: make([]int, 8) // slice  
arr: [dynamic]int  
len(arr);  
cap(arr);
```

4.5.3 Práce s polem

Protože cílem je vytknout identitu pole oprotí ukazateli, tak bych zduraznil rozdíl mezi nimi i z pohledu práce s daty. Pole je na rozdíl od ukazatele kus paměti obsahující stejně elementy za sebou. Tedy bych pole vnímal jako proměnnou definující stejné prvky. Akce prováděné se samotnou proměnnou jako takovou (bez indexace), by se týkaly všech prvků v poli.

I když C formálně rozlišuje mezi ukazatelem a polem, v praxi často dochází k nejasnostem ohledně jejich identity. Například při následné deklaraci pole

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

se vytvoří kontejner pro osm proměnných `arr[0]` až `arr[7]` o datovém typu `int`, který se specifikuje jen jednou. Použije-li se kvalifikátor, tak se taky aplikuje na všechny prvky.

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

Aritmetické operace však nejsou aplikované na všechny prvky, ale chovají se k proměnné jako k ukazateli.

```
int* ptr = arr + 1;
```

V důsledku dvojakosti následující přiřazení není dovoleno

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

jelikož pak není jasné, zda se jedná o přiřazení všech prvků `arr2` do `arr1`, nebo přepsání ukazatele `arr1` na ukazatel `arr2`. Což není z přetčení kódu zjevné, jelikož standardně se využívá kvalifikátoru `const` k zakázání přiřazení do proměnné.

Navíc, syntaxe řešící onen problém již v jazyce je, což přináší ještě špetku zmatku.

```
const int* const ptr;
```

Jasně rozdělení mezi ukazatelem a polem na úrovni datových typů umožňuje eliminovat podobné nejednoznačnosti. Stanovil bych, že kvalifikatory jsou vždy vztaženy ke všem proměnným v poli, že přiřazení pole do pole je definováno jako přiřazení jednotlivých prvků poli, že libovolné operace jsou vztaženy vždy na všechny prvky pole. Vyjimkou by bylo přiřazení pole do ukazatele, kde se jedná o přetypování, a operace zřetězení (viz []), protože to postrádá z definice smysl.

Tedy, například by pak bylo možné inicializovat všechny prvky pole na 0 následně.

```
int[8] arr1 = 0;
```

Nebo sečíst dva pole

```
int[3] arr1 = [ 1, 2, 3 ];
int[3] arr2 = arr1 + arr1; // [ 2, 4, 6]
```

a podobně.

4.6 Řetězce

V C řetězcové literály jsou pouze hezčí variantou zapsání pole konstantních znaků. Ve své podstatě, i když je to primitivní, tak zcela postačující. Problemem je zde 'absence' identity pole jako typu, jak již bylo zmíněno viz. []. Tedy vlastně se s každým řetězcem pracuje jako s ukazatelem.

Jelikož já definuji pole jinak, tak lze rozvinout možnosti pole, aby umožňovali ve výsledku lehčí práci i s řetězci. Samotný datový typ pro řetězec existovat nebude, ale bude jen vestavěná podpora řetězcových literalu, které se při kompilaci rozloží na pole.

4.6.1 UTF-8

Bylo by vhodné rozšířit podporu literalu z ASCII na jiné kodování, které by umožnilo jednoduchou manipulaci se složitějšími symboly. Jako takové kodování bych volil UTF-8, protože je kompatibilní s ASCII a blokem je byte, tedy není závislé na edianech a je velmi rozšířené.

Protože symboly v UTF-8 jsou variabilní délky, víděl bych jako nejlepší možnost compile-time vyhodnocení největší delky potřebné pro uložení jednoho symbolu příslušného literalu a následnou konverzi na pole integralních hodnot o patřičné velikosti, kde každý element bude samostatným symbolem zakodovaným v UTF-8.

```
int16 str = "čau";
str[0]; \\ 'č'
str[1]; \\ 'a'
str[1]; \\ 'u'
```

To umožní pracovat se symboly samostatně, využívat všechny výhody pole a mít pro ASCII text stejně velké pole jako v C. Tedy, například, levá strana může definovat libovolnou velikost k uložení symbolu:

```
int32 str = "čau";
```

Aby bylo možné pracovat s řetězcovými literály jako s prostým kusem paměti, přidal bych možnost definovat tzv. surový (raw) řetězcový literál následujícím způsobem:

```
int^ str = "Hello"R;
```

Příklady jiných jazyků

V jazyce D jsou řetězcové literály standardně ve formátu UTF-8 jako neměnné (immutable) pole znaků, ale pomocí postfixu `u` řetězcového literálu je lze převést na `wchar` (UTF-16) nebo `dchar` (UTF-32).

```
"hello"w;
r"ab\n" // Wysiwyg ("what you see is what you get") quoted strings can be d
```

V Odinu jsou řetězce také ve formátu UTF-8. Navíc obsahuje koncept tzv. `rune`, který umožňuje pracovat s jednotlivými symboly v řetězcích, kde `rune` reprezentuje Unicode kódový bod. Dále nabízí speciální datový typ `cstring`, který slouží k reprezentaci řetězců ukončených nulovým znakem kompatibilních s jazykem C.

```
str := "čau"
for r in str { fmt.print(r, '.') }
// vypíše "č .a .u ."
```


V Zig jsou řetězcové literály ve skutečnosti pouze ukončené nulou bajtová pole, která lze zapsat v kodu pomocí UTF-8 symbolu. Další rozšíření práce je dostupné prostřednictvím standardní knihovny.

```
const arr = "čau";
print("{d}\n", .{arr[0]}); // první byte 'č': 196
print("{d}\n", .{arr[2]}); // byte reprezentující a: 97
```

4.6.2 Operace

Jako jediné konvenční operace nad řetězcy, které by se měly integrovat do syntaxe, bych viděl zřetězení a výběr podřetězce. Ostatní operace by už měly být obsažené v standardní knihovně.

Zřetězení

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

```
u8[] str1 = "Hello";
u8[] str1 = "World";
u8[] str3 = str1 .. " " .. str2;
```

Jelikož délka libovolného pole je získatelná, lze to zobecnit na jakýkoliv typ pole. Je však důležité, aby implementace neobsahovala žádnou alokaci paměti, bylo by to zavádějící. Tedy případné výsledné pole by se muselo samostatně standardně alokovat prostředky jazyka.

Pro dynamické pole by to mohlo vypadat následovně

```
u8[const] str3 = alloc [] : str1 .. " " .. str2;
```

Výběr podřetězce

S podřetězce nebo výřezy polí se lze setkat v různých jazycích v různých podobách. Obvykle se jedná o reprezentaci libovolné souvislé části pole, která sama o sobě neobsahuje data, ale odkazuje na původní pole. Tedy v C by se slice dal definovat třeba následujícím způsobem

```
struct Slice {
    int* dataPtr;
    int len;
};
```

kde `dataPtr` by ukazoval na nějaký element v původním poli, a `len` by specifikovalo délku. Odin a Zig například implementují výřez (slice) právě jako pointer na data a délku.

Odin, Zig a D například vnímají výřezy jako datové typy a například je využívají jako nějaký interface pro dynamická pole.

Já bych nevnímal výřez jako samostatný datový typ, ale spíš jako operaci nad polem. Výřezy by tedy byli datového typu pole, protože ve své podstatě je to to same. ...

4.7 Jmenný prostor

Jedná se o jednoduchý, ale zručný nástroj pro organizaci kódu. Umožňuje zhlukovat proměnné pod jedním společným názvem, který je rozlišitelný překladačem. Na rozdíl od použití identifikujících prefixů nebo postfixů v názvech je z hlediska nástrojů pracujících s kódem (např. LSP) strukturním celkem.

Umožňuje také při kompilaci hromadně pracovat s vnitřně definovanými proměnnými, čehož se dá efektivně využívat i pro import a export částí kódů. Například v Pythonu:

```
import foo;
from foo import x;
```

To, mimo jiné, také umožňuje řešit kolize názvů při importování knihoven (pokud nejste C++). Pokud by se jmenné prostory pouze importovaly, tak by si stále nesli svůj předem definovaný název a s tím i možnost kolizi. Proto je vhodné například umožnit pojmenovování jmenných prostorů ze strany importujícího. Opět, například jak v Pythonu:

```
import foo as boo;
```

Jmenné prostory bych viděl jednoduše jako pojmenovaný scope.

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

Pro přístup k prvkům bych využil syntaxe z C++.

```
Foo::x;
```

4.8 Systém importu

Hlavičkové soubory a s nimi spojený systém importu bych kvalifikoval jako nejhorší část C. Jejich hlavní nevýhodou je duplicita definic. Slouží však k dobrému úmyslu – izolaci implementace a definici rozhraní. Cílem tedy bude zachovávat tuto myšlenku, ale vyhnout se jak preprocesoru, tak i duplicitě.

Základním celkem bude soubor, jelikož jeto to co se ve výsledku předá 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.

Intuitivně se nabízí možnost přímého importu souboru následující syntaxí:

```
import filename;
```

Ověšem, této možností bych se vzdal. Přejde mi, že by jen vybízela k “nesprávnému” přístupu při importování viz [Jmenné prostory], a nepřenašelo nic, co by nešlo řešit jinak.

Zavedl jsem tedy variantu, která by vždy zabalovala soubor ze strany uživatele importu.

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

To by vytvořilo jmenný prostor `Foo` a překopirovalo kořen rozparsovaného souboru `filename` do něj.

Syntakticky se speciálně specifikuje `namespace`, protože jsem se rozhodl onen konstrukt využít k implementaci jiných způsobu zabalení souboru. Například:

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

Dále tento konstrukt lze rozšířit a zavést import jen vybraných symbolů ze souboru.

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

V zásadě tohle umožní robustní import, a více prostředků není potřeba. Zbývá zohlednit viditelnost jednotlivých identifikátorů.

Lze buď vycházet z toho, že vše je viditelné, a omezuje se viditelnost, nebo naopak, vše je nepřístupné, a rozšiřuje se přístup. Druhy přístup je víc prakticky, ale ztrácí na explicitnosti, protože, když importujeme soubor, tak intuitivně očekáváme, že se nám tam naimportuje všechno (nebo alespoň něco), než nic.

Podstatnější je však otázka viditelnosti vnořených importů. Tedy, importuje-li soubor identifikátory z jiného souboru, budou-li viditelné 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 nekladlá se žádná omezení.

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

K označení lokálních importů bych použil slovo `local`

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

Samotná klasifikace proměnných dle viditelnosti by se mohla kdyžtak řešit za pomoci direktiv. Například:

```
#private  
fcn foo();
```

4.9 Přetěžování funkcí

I když se jedná o implicitní konstrukt, který skryva od čtenáře pravou identitu volané funkce, tak přináší, z mého hlediska, jednu zásadní věc, zjednošuená jména funkcí.

Tedy, zamísto třeba vepisování datového typu do jména funkce, k rozlišení funkce, lze 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`, `lmaxf`, `lmaxu`, kde lze přibližně vydedukovat typ očekávané proměnné.

Navíc jména samotných funkcí s použitím identifikujících postfixu/prefixu, nebudou samostatnými celky z hlediska nastrojení pracujících s kódem, tedy v základu samotným kompilátorem a např. LSP. I když by to šlo řešit za pomoci jmenných prostorů, tak to jen zhoršuje jmennou stopu.

Samotná abstrakce nad konkrétní volanou funkcí není pro čtenáře nijak zavádějící. Nebo spíše, je zavádějící stejně jako smyčka `for`, která místo abstraktní instrukce `for` provádí skoky a sem a tam. Smysl čtenář získává ze samotného názvu funkce a vstupních proměnných, přičemž samotná konkrétní funkce je pro něho jako černoá skříňka. I když ona funkce dostává `int`, tak nelze vědět, jestli ten `int` není hned první instrukcí přetypován na `float`. Tedy jediné co to ovlivní je rychlost nalezení správné funkce při potřebě se podívat na její kód, což, bez užití LSP, bude zřejmě delší. Nicméně, neřekl bych, že se jedná o něco závažného.

Z mého hlediska, je lepší ho mít, než nemít. Zbyva tedy rozhodnout, zda povolit implicitní přetěžování, jestli může platit

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

nebo pro jiný datový typ musí být explicitní přetypování:

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

Explicitním uvedením datového typu lze identifikovat volanou funkci. Ovšem, existuje-li potřebná varianta se dá dozvědět při psaní kódu jen z LSP. V takovém to případě je explicitní přetypování z hlediska informace nadbytečné (v porovnání s implicitním přetěžováním). Alternativně až po kompilaci.

Tedy, faktické využití explicitního přetěžování je obdobné assertu, kdy je vyžadována konkrétní varianta funkce, a při její absenci se očekává chyba při kompilaci. Nicméně, přetypování by se muselo specifikovat u každého volání přetížené funkce, což se pře s hlavním cílem – zjednodušení jmenné stopy. A to nemluvě o tom, že vlastně tatež informace existuje dvakrát: jednou při definici a podruhé při volání.

Bylo by tedy vhodné mít implicitní přetěžování, ale s opcí v jistých případech vyžadovat přesnou kontrolu datových typů při vyhledávání funkce. Zvolil jsem následující symboliku:

```
foo!();
```

Využití prapodivného symbolu v tomto případě není zavádějící, jelikož očekávané intuitivní chování výrazu se nemění. Jedná se stále o function call, který nijak nemění

vysledky volání ani jeho vstupy, z hlediska čtenáře je prakticky irelevantní.

4.9.1 Přístup jiných jazyku

Odin obsahuje pouze explicitní přetěžování, jelikož jazyk umožňuje definovat vnořené funkce ve funkcích, a tudíž rozlišení konkrétní funkce, která se má zavolat, není trivialní.

Zig nemá přetěžování funkcí, ale podobného chování lze docílit při kompilaci za pomoci tzv. duck typing a metaprogramování.

```
fn add(comptime T: type, a: T, b: T) T {  
    return a + b;  
}  
  
const result = add(i32, 1, 2);  
const resultFloat = add(f32, 1.0, 2.0);
```

D a C++ mají implicitní přetěžování funkcí.

4.10 Správa chyb

Uvažuje-li se C, tak jazyk nenabízí přímý způsob správy chyb. Chyby lze řešit například návratovou hodnotou, specifickým stavem očekávané výstupní proměnné předané přes ukazatel (často NULL), speciální funkcí vracející poslední chybu apod. V zásadě je to na programátorovi, aby vytvořil nějaký systém pro správu chyb, a jestli vůbec.

Při práci s libovolným kódem je pak nutné číst komentáře k funkcím, externí dokumentaci apod. To opět vede na problém, kdy důležitá informace není součástí strukturních elementů kódu, ke kterým by měly různé nástroje přístup. Navíc to postrádá jednotnost, jelikož různé knihovny mohou řešit správu chyb vždy odlišně. Ve výsledném programu se tak bude muset řešit zbytečný problém – jak s tím naložit.

To vše mně ve výsledku vede k myšlence o přidání standardního systému pro správu chyb.

Z metod řešení jiných programovacích jazyků standardizovanou správu chyb lze v zásadě vyčlenit dva přístupy.

Návratová hodnota Chyba je vrácena jako návratová proměnná nebo její součást.

Obvykle je to spojeno s možností návratu několika proměnných, kde se vyděluje jedno, např. poslední místo, pro případnou chybu (Odin), nebo je přímo speciální doplňující návratová hodnota vydělena jen pro chybu (Go). Take se třeba může vracet struktura obsahující jak případnou chybu, tak i návratovou hodnotu (Rust).

Tenhle přístup je přímočarý a explicitní a dává svobodu programátorovi jak a kde s chybou naložit. Zpracování chyby je pak přímou součástí toku programu.

Tedy chybový stav je prakticky jen další stav programu.

Try-Catch Využívá se systém tzv. vyjímek, kde případně chybové místo je zabalené do `try` bloku, a případná chyba odchycena do `catch` bloku. To umožňuje například nezatěžovat kód správou chyb a psát ho v `try` bloku tak, jako kdyby žádná chyba nastat nemohla. Odchycené chyby se následně zohlední v `catch` bloku.

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

4.10.1 Definice požadavků

Neprve bych si definoval požadavky chybového systému:

- Jednotný datový typ.
- Chyby by bylo možné seskupovat do skupin, které by se mohly kompozičně skládat. Například existují-li samostatné skupiny chyb pro čtení a zápis do souboru, tak by mělo být možné spojit do společné skupiny.
- Definice funkce by měla specifikovat chyby, které mohou být při jejím volání vráceny.
- Umožnit jednoduchou propagaci chyby zásobníkem funkcí dal. Tedy zjednodušit obdobný často se vyskytující konstrukt:

```
err = foo(); if err != null : return err;
```

4.10.2 Implementace

Protože nahlížet na chybu jako jen na další stav programu, i když, řekněme, speciální, je z mého hlediska přirozenější a implicitní přístup neobsahující skryté tok řízení, tak jsem zvolil cestu návratové proměnné.

Jelikož funkce mají k dispozici jen jednu návratovou proměnnou, chyba se bude vracet samostatným kanálem. Nicméně, nechtěl bych vnímat chybu jako přímo návratovou hodnotu určenou jen pro chybu, jak je tomu např. v Go. Protože pak se pro každé volání funkce musí řešit dvě výstupní proměnné. To ve výsledku povede k vytvoření buď implicitních pravidel, nebo k rozvláčné syntaxi.

K bližší předstě uvedu následující příklad v Go. Symbol `:=` vyjadřuje, obdobně jako v Pascalu, deklaraci s inicializací.

```
func foo() (int, error) {  
    return 42, nil;  
}  
  
val1, err := foo();  
if err != nil { ... }
```

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

Intuitivně zde není zcela zřejmé, co se má dít. Prvně se provádí definice `val1` a `err`, načež se ve stejném rozsahu platnosti provádí definice `val2` a opět `err`.

Samozřejmě, je to zohledněné pravidly jazyka, kod je kompilovatelný a nová definice `err` se neprovede. Ovšem, dochází ke sporu syntaxe a semantiky, kde ze syntaktického hlediska se `err` chová jen jako druhá navratová hodnota, ale ze semantického se implicitně provádí 'vyjimky' v pravidlech, jen protože se jedná o chybovou hodnotu.

Navíc se to kompiluje přidáním kvalifikátoru. Bude-li se například chtít označit `val1` jako `const` ale ne `err`, nebo naopak, bude-li se chtít mít jedno `embed` a druhý `const`. To lze řešit na úkor upovídané syntaxe, bude-li se chtít zachovávat explicitnost, nebo přidáním implicitních pravidel. A tedy obdobné řešení mi nevychovuje.

K návratu chyby bych využil pravé strany příkazu. To umožní syntakticky oddělit samotný příkaz a ošetření chyby. Navíc to může do budoucna umožnit odchycení chyby nejen z jednoho volání funkce, ale i z libovolného výrazu obsahujícího i několikero volání funkcí.

Volil jsem následující syntaxi

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

kde by `error` reprezentoval datový typ chyby a případná chyba by se v tomto případě uložila do proměnné `err`.

Zde bych stanovil, že nechci zbytečně obohacovat datový typ chyby o implicitní chování, nebo konstrukty pro tvorbu chyb. Chyba by byla vždy datového typu `error` a chovála by se vždy stejně.

Kuriozně se lze v takovém to případě dopustit jedné výjimky – pominutí samotné definice chyby před odchycením – jelikož je redundantní, místo odchytu totiž může jen pracovat s datovým typem `error`.

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

Protože by to však bylo jak zavadějící, tak bych chybu onu chybu definoval jen lokálně pro následující rozsah platnosti, kde by mohlo být umístěné její spracování. Samotný rozsah platnosti by se nelišil chováním od klasických.

```
error err;
int x = foo() catch err; // odchycuje globalně
x = foo() catch err {
    // odchycená chyba použitelná jen zde
    print(err);
}
print(err) // chyba s prvního volání
```

Množiny chyb

Samotná chyba by měla být jednoduše identifikovatelná přes své jméno, aby ji bylo možné používat pro určení stavu. Například:

```
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 jakési množiny chyb, a jen je. Použijeme následující syntaxi.

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

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

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

K definici chybového rozhraní funkce se pak použije následující syntaxi.

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

Funkce `foo` pak může vracet chyby definované v `ErrorSetB`.

Protože oné množiny mají smysl jen při definici samotných funkcí a definice funkce ve funkci není umožněná, tak jejich definice uvnitř funkcí je zavádějící, a tudíž zakázána. 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 například rozvíjet některou prázdnou množinu na plnohodnotnou, aniž by se rozbil kód využívající onu množinu. Ovšem, má jeden základní nedostatek – vracíme pouze stav. Tedy nemůžeme vrátit informaci o chybě. Teoreticky je to řešitelné přidáním nových stavů, ovšem to zdaleka není praktické.

Prakticky to omezuje jen při logování chyby, protože jinak vždy popisujeme stav programu, který je nezbytný z hlediska jeho činnosti. Tudíž přidání v takovýchto případech chybového stavu je vlastně nezbytné (uvažuje-li se, že tento stav je vhodné vnímat jako chybový, obecně to lze řešit normální cestou).

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, která se kladla za cíl, protože popis samotné chyby už není obecně standartizovatelný, a tak čím dál tím se jedná o konkrétní záležitost.

Pokud by se navržený model zobecnil definici chyby za pomoci struktury nebo unie, tak vlastně dojde k rozporu se standartizací. Systém se totiž zobecní natolik, že bude moci být využíván i pro jiné věci, a mnohé způsoby. A tudíž se vlastně postaví problém, který nevyřeší, jen se přesuneme jinde.

Mohl bych 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.

Navrat chyby

Možnost v chybovém stavu vrátit i normální hodnotu z funkce je zručná záležitost. Může posloužit jako například doplňující informace o chybovém stavu. Navíc, je to dokonce nutná záležitost, jelikož vnímáme chybu jen jako další stav, a ne jako něco zvláštního.

Volil bych následující intuitivní syntaxi:

```
return value, err;
```

Kde `value` představuje proměnnou s návratovou hodnotou, a `err` návratovou chybu.

Pak návrat jen hodnoty je nezměnný, ale otázkou je návrat jen chyby. Lze k tomu přistoupit tak, že vlastně takovýto případ existovat nebude, tedy vždy spolu s chybou se vrátí i hodnota. To také zaručí, že proměnná, do které se запиše návratová hodnota, nebude mít neurčenou hodnotu. I když je to skvělé chování, nelze ho použít. Má-li být jazyk nízkourovňový, musí také dát programátorovi i kontrolu. Nelze jen tak zbytečně vnucovat instrukci. Tedy, přidal bych symbol, např. `_` definující přeskočení proměnné, a skončil s následujícím kódem:

```
return _, err;
```

Implementace v jiných jazycích

4.11 Kontext

Koncept, který umožňuje měnit chování kódu dle svých potřeb a tím i znovuvyžít kód pro konkrétní potřeby. Ovšem, když je kontext explicitní, tedy je předáván jako parametr funkce. Pak je to jen na konkrétní knihovně nabídnout-li vůbec kontext a když ano, tak jaký.

Implicitní kontext, který integrován do jazyka pak umožňuje řešit onen problém. Obzvláště je to výhodné v jazycích s manuální kontrolou paměti, kde za pomoci kontextu se dá řešit problém použitím vlastních alokátorů. Takovými jazyky jsou třeba Odin a Jai.

`print, mem alloc, etc.`

4.11.1 custom alloc

4.12 Compile-time exekuce

V rámci optimalizace kompilatory mohou provádět vypočty některých výrazů pokud je dostatek informace. Tedy například:

```
int x = 5 + 3 * 9 - 2;
```

Výraz definující `x` se může předpočítat a za běhu programu se nebude muset nic vypočítávat.

Ovšem, to vše je prováděno implicitně. Podstatnou možností je ale mít kontrolu nad compile-time exekucí ze stranky jazyka. Koncepce je zpravidla obsažená v nějaké formě v nízkourovnových jazycích, jako například C++, D, Odin, Zig, Rust, atd.

K deklaraci compile-time proměnné bych využil klasifikátoru `embed` od slova `embedded` (vestavený), protože slovo odraží smysl, a vzniká obdobně jako `const`. Navíc má stejnou delku, což by při následných deklaracích vypadalo dobře:

```
const int x;  
embed int y;
```

Implementačně by proměnná neexistovala, ale vždy by se využívala její spočtená hodnota.

Takový to jednoduchý klasifikátor pak umožní provádět velmi složité compile-time vypočty. Protože ze své podstaty `embed` specifikuje, že proměnná má být spočtená při kompilaci, fakticky tedy jako koliv nebyla pravá strana výrazu, kompilátor se ji pokusí vypočíst, a buď uspět, nebo ukončit kompilaci s chybou.

what should happen at compile-time, does happen at compile-time.

Lze se tedy například pokusit vypočíst hodnotu libovolné funkce při kompilaci přičemž funkce se nemusí speciálně předeclareovat jako compile-time jako v C++:

```
fn add(i32 x, i32 y) -> i32 { .. }  
embed int ans = add(4, 2);
```

Obdobně to funguje v D, kde se da využít enumeratoru k definici compile-time proměnné a spočíst pravou stranu:

```
int add(int a, int b) { .. }  
enum ans = add(4, 2);
```

Nebo třeba v Zig parametry funkce `comptime` umožňuje parciálně `comptime-run-time` funkce.

```
fn add(a: u32, b: u32) u32 { .. }  
const ans = comptime add(4, 2);
```

V C++

```
constexpr int add(int a, int b) { .. }  
constexpr int ans = add(4, 2);
```

5 Implementace kompilátoru

K implementaci kompilátoru jsem použil jazyk C++ využívající standartu C++20. Jako cílovou a vyvojovou platformu jsem zvolil Windows, kde jsem využil Visual Studio předkladače (cl) a Visual Studio 2022 k ladění. Jako IL jsem vybral jazyk C, protože jsem s ním a s nástroji pro práci s ním dobře znám. Pro překládání generovaného C kodu jsem zvolil Tiny C Compiler (TCC), a konkrétně jeho knihovnu libtcc, která umožňuje vestavenou kompilaci C kodu.

Jedná se o konzolovou aplikaci, která pracuje se soubory na disku. Tedy, závislost na operačním systému není tak velká a std knihovna z větší části pokryla rozdíly Windows a Unixu. Po mimo rozdíly v systémových funkcích, jako třeba získání cesty k spustitelnému souboru, která nejde získat skrz `std::filesystem`, se hlavně liší knihovný a cesty potřebné pro run-time libtcc. Protože se mi podařilo program úspěšně zkompileovat pod Ubuntu 22.04.5.LTS, port pro Unix by tedy měl být jednoduše realizovatelný.

5.1 Struktura

Struktura aplikace je následující. Nejprve je zpracován uživatelský vstup ve formě argumentu z příkazové řádky. Ten je převeden do konfigurace modulu Compiler. Modul Compiler je reprezentován jmenným prostorem, a tedy je statický. Modul provádí v zásadě čtyři věci postupně. Nejprve běží parser, který inicializuje a zaplní AST. Dále běží validator, který již provede semantickou kontrolu AST. Následně je kód přeložen instancí modulu Translator. V závislosti na uživatelském vstupu výstup překladače může být převeden do binární podoby a popřípadě hned spuštěn.

Cílem bylo dospět k robustní implementaci, která by dokázala vytvořit základnu pro další, řekněme, ideální, implementaci kompilátoru, který by byl specifický pro daný jazyk. Tedy, hlavním úkolem bylo přijít s AST, které by se dalo využít jak pro překlad do libovolného IL, tak i které by bylo možné využít ve vestaveném interpretu.

Zatím se za cíl nekladlo rychlost řešení, ale spíše předložit důkaz konceptu pro navržený jazyk.

5.2 Uživatelské rozhraní

Pro komunikaci s uživatelem se využívá argumentu příkazové řádky. Uživatel má v základu tři hlavní příkazy následující jménem vstupního souboru, na který mají být aplikovány:

build Příkaz sestaví program do spustitelné podoby.

run Příkaz sestaví program do spustitelné podoby a bezprostředně ho spustí v terminálu.

translate Příkaz pouze přeloží do vybraného IL.

Následně mohou být uvedené doplňující možnosti dostupné pro každý příkaz:

-ol (Output Language) Vybere, který IL použít, prozatím jen C.

-of (Output File) Jméno vystupního spustitelného souboru bez rozšíření.

-od (Output Directory) Určí složku, do které se uloží vystupní přeložené IL soubory.

-gd (Generate Debug information) Jestli se má vygenerovat debug informace.

-h (Help) Vypíše nápovědu ohledně uživatelského rozhraní.

-b (Batch/Bash) Říká překladač, že je spouštěn ne přímo uživatelem, ale za pomoci skriptu.

Samotná implementace samotného rozhraní byla trivialní, pouze za pomoci for smyčky a funkce strcmp. Žádné hashování se neprovádělo. Jedinou zajímavou částí byla implementace příkazu run.

Příkaz run potřeboval umožnit spustit zkompileovaný program po kompilaci. Lze třeba použít std::system, ovšem, v takovém to případě nelze ukončit překladač a dále používat stejný terminal pro komunikaci se spuštěným programem. To mi nevyhovovalo.

Na POSIX systémech lze využít funkcí fork a exec z <unistd.h>, jejichž po sobě jdoucí volání nahradí běžící process za zvolený. Windows však nenabízí obodbu funkce fork, k dispozici má pouze funkci CreateProcessA, která se dá využít k vytvoření processu s konkrétním nastavením. Ovšem, nelze, nebo alespoň se mi nepodařilo, docílit obdobného chování jak u fork a exec. I když samotný process po ukončení hlavního programu dále komunikuje s terminálem, samotný terminal se po ukončení původního processu vždy vypíše prompt.

Tedy, na Windows, pro docílení ideláního chování, kdy spuštěný program používá stejný terminal a na venek se to tváří jako stále jeden program lze jen za použití třetího programu, který by sloužil prostředníkem a nejprve spustil kompilátor a následně, v závislosti na výstupu onoho, zkompileovaný program. Takový to program jsem implementoval jako bat skript.

5.3 AST

Centrum programu je statické AST jehož uzel se reprezentuje za pomoci struktury `SyntaxNode`. Kořen stromu je statickým prvkem `SyntaxNode`. `SyntaxNode` obsahuje i další statické prvky, které odkazují na kontejnery s cachovanou informací, která by byla dostupná přímo, aniž by se musel procházet strom. Jedná se například o odkazy na proměnné, definice, funkce atd. Samotná definice pak vypadá obdobně:

```
struct SyntaxNode {
    static Scope* root;
    ...
    NodeType type;
    Scope* scope;
    Location* loc;
    ...
};
```

Každý uzel tedy ve své podstatě nese informaci o svém typu, rozsahu platnosti a lokaci ve zdrojovém kódu.

Každý konkrétní uzel pak dědí `SyntaxNode`, obdobně třeba vypadá uzel reprezentující while smyčku:

```
struct WhileLoop : SyntaxNode {
    Scope* bodyScope;
    Variable* expression;

    WhileLoop() : SyntaxNode(NT_WHILE_LOOP) {};
};
```

I přesto, že `SyntaxNode` představuje uzel stromu, tak neslouží k vyjádření všech syntaktických prvků. Samostatnou reprezentaci mají výrazy. Je to dané tím, že vnitřně výrazy představují vyjádření hodnoty proměnné. Tedy v AST je vždy výraz zastoupen proměnnou.

Obecný výraz je definován velmi jednoduše:

```
struct Expression {
    ExpressionType type;
};
```

Pak konkrétní výraz vypadá například následovně:

```
struct OperatorExpression : Expression {
    OperatorEnum operType;
};

struct BinaryExpression : OperatorExpression {
    BinaryExpression() { type = EXT_BINARY; };

    Variable* operandA;
    Variable* operandB;
};
```

Vyjadření hodnoty proměnné je pak představeno následovně:

```
struct Operand : SyntaxNode {
    VariableDefinition* def;

    Value cvalue; // c as compiler
    Value ivalue; // i as interpreter

    std::vector<Value> istack;

    Expression* expression;
    ...
}
```

Jedná se o vyjadření obecného operandu výrazu, proměnná se pak jen definuje jako pojmenovaný operand. Hodnota je tedy buď vyjadřena odkazem na definici, přímo hodnotou, nebo výrazem. Samotná hodnota se využívá i pro určení datového typu operandu a vypadá následovně:

```
struct Value {
    DataTypeEnum dtypeEnum;
    int hasValue = 0;
    union {
        int32_t      i32;
        int64_t      i64;
        ...
        void*         any;
    };
};
```

Jak lze usoudit z atributů ivalue a istack v definici operandu, abstrakce hodnoty ve formě Value je použita kvůli interpreteru, který s touto strukturou pracuje. Podrobněji oné atributy budou představené v příslušné sekci [].

5.4 Parsing

Rozhodl jsem se nejit cestou generatoru jako je YACC, protože, ze zkušeností, bych stejně musel napsat veškerý kod pro sestavení stromu, neměl bych jednoduchý způsob vypisování chyb libovolným způsobem, měl bych omezeny přístup k buffrum souboru, neměl bych kontrolu nad paměti atd. Navíc bych nevyužil výhod při prototypování syntaxe, protože mám konkrétní typ jazyka, který chci implementovat, tedy bych jen mohl implementovat nutnou abstrakci pro zaměnu syntaktických celku.

Při implementaci parsru jsem se rozhodl nepoužít abstrakci ve tváři lexru, jelikož mně jen zajímalo jaky to je, protože ve většině publikaci se lexer používá a chtěl jsem si udělat názor jinou implementací.

5.4.1 Prace s pamětí

Obecně jsem se rozhodl zatím neřešit dealokaci AST, protože překladač při chybě končí svojí činnost, a tedy bude rychlejší to přenechat OS. Ovšem, protože v budoucnu se nejspíš bude derivovat z překladače implementace LSP, tak by je nutné, aby veškerá paměť, která je alokovaná a není součástí výsledného AST nebo jiných statických částí, byla dealokována.

Každý soubor se vždy načte celý do paměti a je obsažen v ní až do samotného ukončení programu. Je to nutné, protože v libovolný okamžik je nutné moct uživateli vypsat nějaký relevantní kus kódu. Protože samotný nejpreve je sestaveno celé AST a až následně se provádí validace, tak se nemůžou soubory postupně dealokovat.

Abych toho využil, rozhodl jsem se ne alokovat paměť pro textové řetězce, ale používat vždy příslušného ukazatele do příslušného souboru.

5.4.2 Importy

Zvnějšku parser je jen jedná funkce přijímající na vstup jméno souboru, který oná vníma jako vstupní soubor programu. Vnitřně globálně parsing řeší dvě funkce, `parseFile` a `processImport`, které za pomoci struktury `ImportNode` řeší parsing všech dalších souborů a spojení jich do výsledného AST.

Každý soubor se parsuje do předem dodané instance `Scope`. Při parsování souboru se případné importy zařazují postupně do příslušného uzlu importovacího stromu reprezentovaného strukturou `ImportNode`. Importovací strom slouží k zachování vztahu souborů a kontrole cirkulárních importů.

```
struct ImportNode {
    FileId* fileId;
    Scope* fileScope;
    ImportStatement* import;
    ImportNode* parent;
    std::vector<ImportNode*> children;
};
```

Aby se předešlo parsování znovu zpracování téhož souboru, každý rozparsovaný soubor se označil unikátním id. Id bylo definováno následovně:

```
struct FileId {
    uint64_t size;
    std::filesystem::file_time_type time;
}
```

Jednalo se tedy o reprezentaci souboru jeho velikosti a časem vytvoření. Takovýto přístup umožňoval rychle porovnávat různé soubory. Jinak by třeba šlo použít absolutní cesty k souboru, ovšem to mi přišlo nešikovné, protože obecně cesta může být všelijak dlouhá. Každý soubor byl pak ukládán do mapy a před parsingem se ověřovalo, jestli již existuje nebo ne.

Algoritmický by se pak funkce `processImport` dala popsát následovně:


```

processImport(parentNode) :
    for node in parentNode :
        fpath = getRealFilePath(node.fname)
        node.fid = genFileId
        file = find(parsedFiles, node.fid);
        if (file) :
            node.fscope = file.scope
        else :
            parseFile(fpath, node)
            insert(parsedFiles, node.fid, node.fscope)
        if not doesImportExistsInPath(parentNode, node) :
            error exit;
        insertToAST(node);
    for node in parentNode :
        processImport(node)

```

Je nutné podotknout, že importovat symboly dopředu Scope je relativně drahé, protože se prvky v každém array-listu budou muset posunout. V případě třeba funkci nezáleží na pořadí, protože neosahují sam o sobě spustitelný kód a jejich definice nemusí předcházet použití. Tedy je lze prostě zařadit nakonec. V případech, kdy to nejde, krom zřejmé změny kontejneru na třeba list, lze alespoň posunout prvky jen jednou za soubor udělav posun až po zpracování všech importu.

5.4.3 Samotný parsing

Parser jsem implementoval procedurálně, kde každá funkce relativně odpovídala syntaktickému celku, který má za úkol rozparsovat. Každá taková funkce dostává pointer na buffer s textem a pointer na Location, které definuje lokaci, tedy hladvně index, který slouží jako přímý index do bufferu a číslo řádku.

Ovšem prakticky to není tak hezké, protože rozhodnutí, kterou větví gramatiky se vydat se rozhodují vždy individualně a využívajíc napřímo bufferu textu. Kdežto v případě lexru bych mohl využít tokenu a buď indexace pole, nebo switch-casu v každém případě, což by bylo víc čitelné.

Klíčová slova se vnítně namapovala na integralní hodnoty, což vytvořilo abstrakci mezi samotnými celky, které reprezentují a slovy jako takovými. Navíc to umožnilo použití stejných funkcí pro zpracování různých množin slov. Například direktiv kompilatora. Pro zpracování klíčových slov byl v zásadě použit jeden velký switch-case, protože mám rad switch-case. V budoucnu, ovšem, by ho bylo dobré nahradit indexací pole, kde každý index by ukazoval na samostatnou funkci.

Operatory se reprezentovaly jako `uint32_t`, což imeožovalo délku operatoru na 4 symboly (po případě by šlo rozšířit na 8 za použití `uint64_t`), ale umožnilo indexaci. Omezení na malý počet symbolu pro operator není faktický omezující, protože většínou operatory jsou reprezentovány matematickými symbolami nebo krátkými slovy. Výhoda je však vyznačná. Look-up operatoru mohl být proveden následovně:

```

OperatorEnum findBinaryOperator(uint32_t word) {
    switch (word) {

```

```

    case operators[OP_ADDITION].word : return OP_ADDITION;
    ...
    case operators[OP_SHIFT_LEFT].word : return OP_SHIFT_LEFT;
    default : return OP_NONE;
}
}

```

5.5 Validace AST

Úkolem bylo jak provést semantickou kontrolu AST, tak i obohatit AST o potřebnou semantickou informaci. Ve výsledku by AST mělo být bezchybné a plně odpovídat navrženým podmínkám. Tedy, ukazatel vždy směřuje na platné místo v paměti, které lze jednoznačně identifikovat podle daných pravidel, jinak má hodnotu NULL; vždy je použita správná hodnota enumerátoru; všechny doplňující příznaky jsou správně nastaveny; atd. Další moduly pak již nemusí řešit validnost a mohou slepě důvěřovat AST.

K validaci se využívalo cachovaných odkazu na konkrétní typy uzlu. To bylo prováděno lokálně v rámci rozsáhu platností a globálně pro celé AST. Tedy se nemusel procházet celý strom, ale mohlo se přímo sekvenčně přistupovat k většině důležitým prvkům.

Během validace se prováděly následující akce:

- propojení definic uživatelských datových typů s jejich konkrétními užitími.
- vzájemné propojení chybových množin
- propojení chybových množin užitých ve funkcích s jejich definicí
- validace definic uživatelských datových typů
- propojení proměnných s definicemi
- propojení a validace *goto* příkazu
- validace, že každá funkce má globální platnost
- propojení volání funkcí s definicemi
- výpočet všech compile-time proměnných
- výpočet, nebo převedení na výraz, délek polí
- kontrola argumentů volání funkcí
- kontrola *return* příkazu
- kontrola správnosti přiřazení a alokací
- kontrola inicializací

- kontrola podmíněných výrazu a switch-casu
- kontrola obecných výrazu

Data-flow

Důležitou otázkou bylo řešení správného přiřazení proměnných k definicím. V zásadě definice se dají členit do dvou kategorií. V jednom případě definice je dostupná pro proměnnou nachází-li se v rozsahu její platnosti. Ve druhém případě navíc musí platit, že definice předchází proměnnou.

První případ se dá řešit jednoduše, stačí aby struktura definujících rozsahů platnosti vždy měla kontejner pro uložení odkazu na vyskytované v ní definice. Vhodným kontejnerem je hash mapa, protože ta zároveň umožní zachovat identitu názvu už při parsingu. Pak stačí v případě každé proměnné rekurzivně procházet rozsahy platnosti směrem ke kořenu a prohledávat kontejner na předmět odpovídající definice. Pseudokodem by to šlo reprezentovat následovně.

```
findDefinition(var)
    scope = var.scope
    while(scope) :
        def = find(scope.defs, var.name)
        if (def) return def
        scope = scope.scope
    return null
```

Druhý případ už byl o něco složitější, jelikož zaleželo na pořadí. Zřejmou možností by bylo postupovat obdobně, ovšem k uložení dat využít array-list. Pak v každém rozsahu platnosti se zachová pořadí. Jak lze ale chápat sekvenční prohledávání kontejneru textových řetězců není zrovna rychlé. Navíc, by obdobný kontejner musel obsahovat nejen definice, ale i uzly reprezentující rozsahy platností pro zachování informace o pořadí mezi přechody z dítěte do rodiče.

Pokud by se situace zjednodušila a omezilo se jen na jeden soubor, tak řešit by to šlo na úrovni parseru. Opět by se použilo hash mapy a každá proměnná by se testovala jako v prvním případě přímo při parsingu. V takovém to případě by v mapách neexistovali ještě definice, které by byly definovány za proměnnou. Ovšem, protože obecně chci mít možnost manipulovat se stromem na úrovni AST, tak nemůžu zaručit, že se pořadí uzlu po parsingu jednoho konkrétního souboru nezmění. Navíc importy samotné narušují standardní tok definic Foo::Boo

Rozhodl jsem se tedy postoupit jinak a zavést nový atribut v SyntaxNode, který by definoval pořadí definic a ostatních potřebných elementů v rámci Scope a vybral jsem pro něj ne moc vhodný název *parentIdx*. Smyslem bylo přiřadit při parsingu každému potřebnému uzlu index z pohledu jeho rodiče a při validaci při výběru kandidata z hash mapy rozhodnout se v závislosti na pořadí indexu.

Zde lze vidět schematickou ukázkou již reálné funkce upravenou pro ilustrační cíle:

```
Variable* findDefinition(Scope* scope, Variable* const var) {
    int idx = var->parentIdx;
```

```

while (scope) {
    SyntaxNode* node = find(scope->defSearch, var);

    if (node) {
        if (node->parentIdx < idx
            && node->type == NT_VARIABLE) {
            return node;
        }
    }

    idx = scope->parentIdx;
    scope = scope->scope;
}

return NULL;
}

```

I když se to může zdát zavadějící, protože se při manipulaci s AST se uzly budou muset posouvat, tak je zde pár vlastností, které je nutné si uvědomit:

- Indexy jsou vždy relativní vůči Scope, a tedy se posune jen dílčí část.
- Indexovat se musí jen specifické typy uzly, nemusí se procházet tolik prvku.
- Indexovat lze i do zaporných hodnot, jelikož se jedná pouze o nástroj k rozhodnutí o pořadí.
- Indexy se mohou duplikovat.

Například, u funkcí a jmenných prostoru nezáleží na pořadí, každá funkce tedy může vždy mít index 0. Při importu rozsahu platností nebo jmenného prostoru se žádná již existující proměnná nemůže odazat na jejich vnitřní proměnné a není nutno nic měnit. Pokud se importem přidává definice na začátek souboru, což je zcelá běžné, tak se lze podívat na index stavajícího prvního prvku a použít index menší. A podobně...

Tedy, pokud se importy omezí jen na počátek souboru, tak bych řekl, že lze vždy docílit $O(1)$ aktualizace indexu.

Přetěžování funkcí

Klasicky přiřazení volání funkce ke správné definici se ničím neliší od přiřazení proměnné definice, postačí jen jméno. V případě přetěžování funkcí už roli hrají i datové typy a počet vstupních argumentu. V případě implicitního přetěžování již nejde identifikovat funkci striktně na rovnosti datových typu, viz příklad:

```

int foo(int x, float y);
int foo(float x, float y);

foo(1, 2);

```

Jak lze vidět, datové typy argumentu volání funkce foo striktně nesedí ani jedné definici, ovšem intuitivní je, že by se měla přiřadit první definice. Bude tak i například

v případě C++.

K určení nejvhodnější funkce jsem tedy použil jakéhosi score, které sestavovalo v závislosti na potřebě přetypování a datových typech, které se přetypování účastnili. Score bylo definováno jako int viz.:

```
struct FunctionScore {
    Function* fcn;
    int score;
};

std::vector<FunctionScore> fCandidates;
```

Což pro moje účely bylo dostačující. Hodnoty score jsem rozdělil do čtyř kadegorii, v závislosti na kategorii se pak přičetla příslušná hodnota do skore. Kategorie a zároveň i hodnoty byli vyjádřeny enumerátorem následovně:

```
enum Score {
    FOS_IMPLICIT_CAST,
    FOS_SAME_SUBTYPE_SIZE_DECREASE,
    FOS_SAME_SUBTYPE_NO_SIZE_DECREASE,
    FOS_EXACT_MATCH,
};
```

Kde nejvýše hodnocenou kategorií je případ přesné shody. Dale následuje případ přetypování do stejného podtypu (například i32 do i64) bez snížení přesnosti. Na což navazuje obobný případ akorat se snížením přesnosti (například i64 do i32). A nejmeně hodnocenými jsou pak ostatní jiné implicitní přetypování.

Zde by šlo obohatit model o více kategorií a rozlišovat zda se jedná o přechod od neznamenkového typz do znamenkového a naopak, nebo rozlišovat na kolik se zvětšila nebo zmenšila přesnost při přechodu z jednoho typu na druhý. Ovšem, takové to chování jsem prozatím shledal jako zavadějící.

Tedy, pro každé volání funkce se nejprve v jeho rozsahu platností naleznou funkce se schodnými jmény. One funkce se uloží do kontejneru fCandidates. Každá kandidátní funkce se projde a spočíta se její skore v závislosti na volání. Na konec se vybere funkce s největším skore. Chyba nastane pokud budou dvě a více stejná maximalní score, nebo nezbude žádná funkce se scorem.

Vypočet datových typu a výrazu

Nejprve jsem chtěl provádět vypočty datových typu spolu s evaluací výrazu v jedné funkci, která by se pokusila spočítat každý uzel výrazu a při neúspěchu by jen vyjádřila datový typ. Při procházení výrazu jsem ovšem vždy byl ve stavu, kdy se mohla očekávat neplatná hodnota, která měla být zohledněna. Dospěl jsem tedy k tomu, že by bylo vhodnější ty funkce rozdělit i když mají fakticky stejnou logiku.

5.6 Interpret

Protože compile-time evaluace je základem jazyka a musí být umožněno vyhodnocovat prakticky všechno, rozhodl jsem to řešit implementací vestaveného interpreteru. U kolem interpreta je možnost jak evaluace proměnných a funkcí, tak i libovolného operatora.

Jak bylo zmíněno, interpreter využívá speciálního atributu `ivalue` v `Operand`. Atribut se využívá pro ukládání mezivýsledku interpreta po spočtení každého uzlu. Postupně se tak spočítá hodnota finalního, vstupního, uzlu a hodnota se přepíše do `cvalue`. Ovšem, to nestačí, protože se také musí řešit evaluace funkcí.

Evaluace operatoru

Samotným jádrem jsou dílčí funkce umožňující evaluaci konkrétních operatorů. Funkce jsou zabalené do unie, která umožňuje sjednotit různé typy funkcí:

```
union OperatorFunction {
    void (*binary) (Value*, Value*);
    void (*unary) (Value*);
};
```

Funkce v poli jsou seřazeny seřazené do skupin dle datového typu, kde každá skupina je seřazená dle operatoru. Pořadí určuje vždy příslušný enumerator. Obdobně třeba vypadá funkce pro použití operatora, kde, lze vidět způsob indexace pole `operatorFunctions`:

```
int applyOperator(OperatorEnum oper, Value* vl) {
    OperatorFunction fcn =
        operatorFunctions[vl->dtypeEnum*OPERATORS_COUNT+oper];
    fcn.unary(vl);
    return Err::OK;
}
```

Funkce a rekurze

V případě funkce se spočtená hodnota zapíše do `ivalue` každé vstupní proměnné. Funkce je pak teoreticky spočetná, ovšem zde se již musí pracovat ne s uzly výrazu, ale s libovolnými uzly AST. Každý uzel tedy musí mít svou definovanou logiku jeho výpočtu. Obdobně vypadá například vykonání podmíněného výrazu:

```
int execBranch(Branch* node) {
    for (int i = 0; i < node->expressions.size(); i++){
        evaluate(node->expressions[i]);
        if (readValue(node->expressions[i])>i32) {
            return execScope(node->scopes[i]);
        }
    }

    if (node->scopes.size() > node->expressions.size()) {
        return execScope(node->scopes[node->scopes.size() - 1]);
    }
}
```

```

        return Err::OK;
    }

```

Protože funkce může obsahovat další volání funkcí, nemohl jsem si vystačit jen s `ivalue`, a tak jsem přidal ještě atribut `istack`. Jak plyne z názvu, `istack` měl umožnit ukládat hodnoty všech ostatních volání, každá funkce pak měla atribut `istackIdx` reprezentující index, který by měl být použit pro získání hodnot pro její kontext.

Funkce ovšem mohla volat i samu sebe. Musel jsem tedy k atributu `istackIdx` zavést atribut `icnt`, který by počítal každé takové volání a těm samým sloužil jako identifikace kontextu. U každé `Value` se pak nastavoval atribut `hasValue` na odpovídající index, a tak mohla být provedena kontrola, jestli hodnota byla spočtena v nynějším kontextu. Pokud hodnota nebyla spočtena v daném kontextu, tak se hodnota překopírovala do `cvalue`, v opačném případě se hodnota kopírovala zpět do `ivalue`.

I když to může znít složitě, fakticky se jednalo o překopírování všech hodnot do `cvalue`, což lze udělat před voláním znovu sama sebe. Ve funkci se normálně pracovalo s `ivalue` a při návratu se hodnoty překopírovaly zpět. Akorta se to provádělo na místě výpočtu, a tedy se kopírovali jen potřebné proměnné.

Protože ve výsledku by se musel po mimo hlavní části udržívat také všechny uzly v interpretu a neustále je měnit při změnách či refaktoringu, tak jsem se rozhodl se jim moc nezabývat dokud projekt je v aktivním vývoji. A tedy implementováno je pár uzlů, které slouží k demonstraci funkčnosti řešení. Udržované jsou jen aritmetické funkce, které se vnitřně využívají třeba k výpočtu delek polí.

5.7 Generace C kódu

Překladač AST byl řešen na rozdíl od ostatních modulu abstraktně. Má totiž smysl umět překládat strom do různých IR, na rozdíl třeba od parsru, kterých nemá mít smysl několikero, jelikož pracuje vždy s jedním konkrétním jazykem.

Překladačem je vlastně instance následující jednoduché struktury, která definuje interface:

```

struct Translator {
    FILE* mainFile;
    int debugInfo;

    void (*init)                (char* const dirName);
    void (*printNode)           (FILE* file, int level,
                                SyntaxNode* node, Variable* lvalue);
    void (*printExpression)     (FILE* file, int level,
                                Expression* node, Variable* lvalue);
    void (*printForeignCode)    ();
    void (*exit)                ();
};

```

Zasadními funkcemi jsou `printNode` a `printExpression`.

5.7.1 Souborová struktura

Generovat výsledný kód jsem se rozhodl sekvenčně přímo do souboru. Proto jsem potřeboval několik souborů, abych mohl generovat různé části AST do různých souborů a ve výsledku je spojit v potřebném pořadí.

Použil jsem následující soubory:

- main.c, byl použit jako hlavní soubor, kde byli přidány ostatní soubory. Zapisoval se kód, který bylo možné dát do main funkce.
- functions.h zapisovaly se tam definice funkcí, aby mohly být dostupné napříč všemi ostatními funkcemi
- functions.c zapisovaly se samotné definice funkcí
- typedefs.h zapisovaly se definice struktur a unii.
- variables.h zapisovaly se deklarace definice proměnných, především globálních.
- foreign_code.c, veškerý kód cizích jazyků, viz[]

pořadí....

5.7.2 Globální rozsah platnosti

Protože na rozdíl od C je vstupním bodem počátek souboru, tak i všechny proměnné v něm obsažené by měly být globálními a v případě definic jsem nemohl je prostě napsat do mainu, ale musel jsem zapsat deklaraci zapsat do souboru variables.h.

Problematickou částí byla definice statických poli, kde

5.7.3 Vykreslení pole

Nejspíš jedinou netriviální věcí při vykreslení bylo vykreslení výrazu obsahujících konkatinaci poli bez využití alokaci.

Implementaci jsem řešil generací for smyček. Myšlenka byla následující, v nezávislosti kde se vyskytuje operátor konkatinace z pohledu stromové struktury, každý takový výraz vždy rozšiřuje výsledné pole a tedy se chová, řekněme, lineárně a každý usek výrazu mezi operatory lze tedy reprezentovat for smyčkou.

Podívejme se na příklad pro ilustraci.

```
ans = "A" .. (1 + "BB" .. ("C" + 3)) .. "DD"
```

Je-li na levé straně pole řádné velikosti (zajisti to je úkol validátora), tak lze postupně procházet výraz a zapisovat výraz do for smyčky, kde na levé straně je ans s určitým odsazením a na pravou postupně zapisujeme výraz. Smyčku uzavřeme pokud narazíme na operátor konkatinace. Pak posuneme offset levé strany o délku dílčího pole, které jsme připojovali a opakujeme.

Tedy, nejprve se nastaví výchozí odsazení na 0 a délka procházení na velikost prvního pole:

```
off0 = 0;
len0 = 1;
for i to len : ans[off0 + i] = "A"[i];
```

Až narazíme na konkatinaci, posuneme odsazení levé strany, nastavíme délku a pokračujeme novou smyčkou:

```
off1 = len0;
len1 = 2;
for i to len1 : ans[off1 + i] = 1 + "BB"[i];
```

Opakujeme v dalším kroku:

```
off2 = len1;
len2 = 2;
for i to len2 : ans[off2 + i] = "C"[i] + 3;
```

Opakujeme v posledním kroku:

```
off3 = len2;
len3 = 2;
for i to len3 : ans[off3 + i] = "DD"[i];
```

Výsledný výraz je schodný s původním a dá se generovat sekvenčně. I když ukazano na statickým příkladem, dá se zobecnit na dynamicky.

Prakticky se generace realizuje retrospektivně. Tedy, nejprve je rekurzivně procházen výraz než se narazí na operator konkatinace, načež se vykreslí původní část. To umožňuje získat délku pole před vykreslením aniž by délka byla uložena napříč všemi uzly ve výrazu.

5.8 Vestavená kompilace C kodu

Protože jsem chtěl, aby kompilátor obsahoval konvenční možnost generace spustitelného souboru, tak jsem dospěl k integraci TCC kompilátoru. Obecně bych mohl použít třeba `std::system("gcc build my code pls")`, tak jsem jak nechtěl být závislý na něčem, co už má uživatel předinstalováno.

Opce by bylo buď distribuovat gcc, nebo i jiný překladač, spolu s kompilátorem, což není šikovný, protože program pak nejde distribuovat jako zdrojový kód. Lepší cestou by bylo integrovat kompilátor do kodu přes příslušnou knihovnu.

GCC

LLVM

TCC jsem vybral proto, že je to malý kus softwaru, který vypadal vhodně pro potřebu vestavení, jelikož by nezabral moc místa.

Protože jsem nenašel moc rozumnou informaci o správném použití knihovny pro mojí potřeby, tak popíšu použitý postup trochu detailněji.

Samotná instalace knihovny je standardní, tcc obsahuje include a libtcc složky, které se musí předat compilatoru jako include path a lib path respektivě a jeden dll soubor. Knihovně potřebné pro samotnou kompilaci C kódu v run-timu jsou pak dostupné ve složce lib.

Nejprve je nutné vytvořit instanci TCCState, předat potřebné cmd argumenty a nastavit výstupní program. Záleží na pořadí!

```
TCCState *state = tcc_new();
tcc_set_options(state, "-ggdb");
tcc_set_output_type(state, TCC_OUTPUT_EXE);

tcc_add_library_path(state, libPath.c_str());
tcc_add_include_path(state, tccIncPath.c_str());

tcc_add_library(state, "gdi32");
..
tcc_add_library(state, "msvcrt");

tcc_add_file(state, "main.c");

tcc_output_file(state, Compiler::outFile);
```

Většina funkcí vrací chybový stav zohlednění kterého se v ukázce pominulo. K uvolnění resursu alokovaných tcc_new se použije funkce tcc_delete.

Ke kompilaci C kódu

5.9 Správa chyb a logování

Protože chyby a varování jsou fakticky jediným komunikačním prostředkem kompilatoru s uživatelem a jejich kvalita je zcela zásadní, potřeboval jsem si vytvořit jednotný a robustní systém chyb a definovat pravidla jednotná pravidla pro správu a logování chyb.

Aby se docílilo jednoznačnosti, definoval jsem si pro sebe pravidlo – chyba musí být logovaná přímo ve funkci vyskytu. Jinak se musí řešit jestli použitá funkce vracející chybu už provedla logování nebo ne. Navíc to umožňuje klasickou propagaci chyby až do mainu, kde se v každé funkci, buď díky lenosti, nebo nerozhodností, se chyba neošetří a jen se předá dál.

Problemem při logování v místě chyby může být ne vždy úplná znalost kontextu, ovšem, protože postupně parsujeme AST, tak lze případně retrospektivně podrobnější informaci získat. Navíc, vnější funkce může v případě nutnosti provést svůj doplňující log, pokud to z jejího kontextu přijde vhodné.

Chybu jsem tedy reprezentoval jednoduše jako negativní integralní hodnotu, kde bezchybový explicitní bezchybový stav je 0. Každá funkce by tedy měla mít návratový typ `int` a vracet případnou chybu. Pro každou chybu jsem definoval take standartní chybovou hlášku `printf` syntaxi. Hlášky byli umístěné v pole a indexovatelné absolutní hodnotou příslušné chyby.

Logovací system se skládá z pár funkcí a statických hodnot pod namespacem `Logger`. Za pomoci bitových hodnot lze filtrovat typy hlášek, například potlačit informace a varování. Logovací funkce pak umožňovly po mimo hezčího vypisu samotné hlášky take vypsát konkrétní místo ve zdrojovém kodě v řádkovém formátě a podtrhnout nutnou část viz obr[].

Protože někdy chyba funkce nemusí známenat kompletní chybu parsingu, někdy se může parser vydat jednou cestou, zjistit, že to nejdě rozparsovát a zkusit jinou cestu, tak je nutné umět se vyhnout logování. Pro takový to případ je v rámci `Loggeru` dostupná proměnná `mute`, kde každé vlákno může nastavit vlastní bit v závislosti na svém id. Prozatím ale funguje jen jako `bool` hodnotá, protože multithreading v aplikaci nebyl řešen.

Ovšem, protože chyba samotného parsru vlastně vede ke konci kompilace, tak generace chybové zpravy může byt obecně složitá, a tedy i když řešení skrz samotný `Logger` je čisté, tak není optimalní. Lepší by bylo předávat potřebnou informaci skrz vstupní proměnné funkce, v ideale zavést nějaký context a předavat obecně potřebné proměnné skrz něho.

6 Závěr