FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

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 nahraďte
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

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

Kompilatorem 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 take 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 zkompilován 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 frémy využívaných GCC, která je využívaná 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 sleď 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[].

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 nejpřirozenější 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.[]) 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 interpretery.

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 neprazdná množina terminálních symbolů, tzv. terminálů.
- V je konečná neprazdná množina neterminálních symbolů, tzv. neterminálů.
- S je počáteční neterminal.
- 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. Neterminal 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]

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

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

[1] Hvězdička (*) 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 $\alpha \to \beta$ z P platí $|\beta| \geq |\alpha|$ a zároveň pravidlo $S \to \epsilon$ se nevyskytuje na pravé straně.
- S je typu 2 nebo take bezkontextová gramatika právě tehdy, když pro každé pravidlo $\alpha \to \beta$ z P platí $|\alpha| = 1$. Neboli, že α je pouze neterminal.
- P je typu 3 nebo take regulární gramatika právě tehdy, když každe pravidlo z P je v jedné z forem:
$$A \to cB, A \to c, A \to \epsilon,$$
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
$$S \to \varepsilon,$$
nebo
$$\alpha A \beta \to \alpha \gamma \beta,$$
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

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

Pravidlo definuje nonterminal `goto` jako možné slovo počínající goto, čírym 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

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

`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 rozëířená (Extended) Backus-Naur forma. Existuje několik verzí a ma ISO Standard.

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

YACC notace bližší seznámení s YACC'em muže byt naleznuto zde [].

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

Složené závorky představují místo, kam se umisťuje C kód, který se má provést při parsingu oných elementů. Neterminaly z příslušných pravidel jsou přístupná za použití symbolu $. Například $ označuje proměnnou samotného pravidla, $1 proměnnou prvního terminala či neterminala (`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 muže byt 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 zkompilovat, 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 enginu. 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 moct 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 'TOhtml' -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 protiřečí.

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 zkompilovaný 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 oné 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 zasadě cílem je přijit s moderní obdobou C, tak by bylo k věci jim i začít.

V jazyce C se mi v zasadě líbí jeho jednoduchost, co se napiše, to v podstatě i platí. Neprovádí se v pozadí tucet instrukcí — neobsahuje skrytý tok řízení. Má explicitní syntaxi, která se dá číst. Nabízí možnost příme a volné práce s pamětí, ktera je ve většině jazyku potlačená kvuli bezpečností, ale dodává tu špetku hravostí, kdy se problém vyřeší chytrou, nebo i hloupou, interpretaci 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 většíniu 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 umožňujicí 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 odpovidat skutečnosti, například: deklarace proměnné by neměla byt ve vychozím připadě konstatní, protože po přečtení kodu, ktery nespecifikuje vlastností deklarce je přirozenější se domnivát, že žadných vlastnotí nenabyva, než že jsou nějaké standardní skryté. Syntaxe by měla byt srozumitelná intuitivně i bez znalostí gramatiky. Tedy, v zasadě, akce spiš interpretovaný slovy než symboly.

Jazyk by neměl přímo plnít nějaky cíl, nebo byt určen specialní sfeře. Jedná se o tvorbu nastroje pro moje vlastní užití, jazyk, ve kterem 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 čí jiném smyslu řešicí muj problem. Některé z mého hlediska jsou vskutku skvělé a zajímavé. Ovšem, nenašel jsem se v ních na tolik, abych pak něco nevyčítal. Přijde mi tedy vhodným je představít a vytknout věcí, které jsou odlišné od moji představy.

4.1.1 C++

besprostřední naslednik C, ktery je i často s ním nerozlučně poejn jako C/C++. C++ ponechavá v zakladu C s drobnými změnami a staví na tom v zakladu za pomocí standardní knihovný. C++ tedy dědí i špátné vlastností C, jako jsou například makra a system importu. Nové vlastností často nejsou častí jazyka, ale objekty právě standardní knihovny. Mezi takové patří i základní věci jako pole.

Jazyk obsahuje mnoho různorodých konceptu umožňujících řešit problemy paradigmalně různými způsoby. I když se to může vnímat jako výhoda, a člověk si může vybrát nějakou podmnožinu vlastností, která mu vyhovuje, tak jen zřídka veškerý kod 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 metaprogramovaní a objekty.

Syntaktický je pak jazyk příšerný, protože po mimo míchání samotných přístupu k programování se míchá i C a C++ kód. To zesložiťuje vnímání a čitelnost kodu, protože nebí moc jasné co se má dít, a jak C++ objekty nakldaaji s puvodnimi datovými typy a jak je interpretuji.

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 importovaní system je řešen za pomoci modulu, podobně jako v Java. Žádná makra, spoleh na compile-time exekucí a genericitu jednodušit 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

Jedna se o relativně moderní, jednoduchý, procedurální jazyk jdoucí ve stopách C. Jazyk staví na compile-time exekucí mataprogramování nabízejíc široké spektrum možností jejího využití. Neobsahuje skrytý řidicí tok programu. Práce s pamětí je manualní. Překladač obsahuje varianty sestavení, které umožňují provádět doplňujicí bezpečnostní kontorly jak při kompilaci tak i v run-timmu. Navíc je možné sestavovát program cross-kopilaci.

Ve své podstatě se jedná o velmi blizky jazyk mym představám až ná některé drobné věci jako například, že ukazatel je standardně nenulovatelný. Hlavní vytku mám k syntaxi, která není moc explicitní a využívá až moc symbolu.

Ukazka 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 syntaxi. Nabizí pár zajimavých jako třeba kontext viz[]. a vestavené aritmetické programování s poli, a k tomu i maticový typ, ktery umožňuje třeba nasobení matic, matic a poli a pod.

Jako nedostatek bycho vytknul absenci explicitní compile-time exekuce, která je fakticky možná jen při definici konstant. Syntakticky jazyk je relativně implicitni a podobá se jazyku Go. Syntaxe mi přijde vice 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 funckce. Taková funkce muže mit za ukol předání vstupních argumentu programu a oddělení globalního scope.

Samotný koncept mi přijde obskurdním:

- Prvně, main funkce zvyšuje indencí kodu a zesložiťuje strukturu programu bez možnosti se tomu vyhnout. Když, například, uživatel bude chtít začít v lokalním scope, protože je to to, co se mu líbí na main funkci, tak to muže udělat přímo za pomocí odpovidajicího syntaktického kostruktu. Dokonce, když to uděla, tak jasně da čtenáří znat svou myšlenku.
- A za druhý, ruší chápaní pořadí vykonání instrukcí. Instrukce se totíž běžne mohou objevít i v gloablním scope. Ovšem, protože z main funkce nelze nijak skočít do globalního scope, ale stale se jedna o misto, kde by měl program začít své konání, tak není jasné jak, a jestli vubec, se provedou globalní instrukce.

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

4.3 Alokace

Dynamickou alokaci bych nevnímal jako funkcí, operator, nebo výraz, ale jako samostatný celek, ktery by sloužíl alternativou při přiřazení. Tedy, že by přířazení buď bylo alokaci, nebo výrezem. Smyslem je vždy zaručit, že dynamicky alocovaná paměť bude vždy přířazena proměnné.

Syntaxi jsem zvolil nasledujicí

```
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 byt schodný s tím na levé. Formalně je to možné, protože `alloc` je alternativní `rvalue`, oprotí výrazu `new` v C++ či D, ktreý by se měl řidít pravidly vyrazu.

Nasledujicí řadky by tedy vyjadřovali to same.

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

Navíc bych umožnil přimou inicializací za pomoci symbolu `:`, ktery se využívá v jazyce jako počatek přikazu. Nasledujicí kod inicalizuje alokuje velikost odpovidajicí int a a bude inicilizována na `1`.

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

4.4 Komentáře

V C, nebo například i v C++ a D nelze vnořovat blokové komentáře `/**/`. Není to vyznáčný nedostatek, ale stalé nepřijemny, pokud se komentuje něco, co už obsahuje komentář. Navíc se je to nekonzistentní s řadkovými komentáří, ktere se mohou vnořovat.

Volil bych nasledující syntaxi.

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

4.5 Pole

Jedna se o nejzákladnější a nejužitečnější datovou strukturu vyskytujicí se v programovani. V C se ovšem pole daji používat jen ve statických případech, kdy velikost lze stanovit ještě při kompilaci. Ovšem, i v připadech, kde je to dostačující, tak při využíti pole jako argumentu funkce se funkce buď musi definovat pro konkretni velikost pole, anebo obecně pro ukazatel.

První případ je použitelný jen zřídka, jelikož nepřináší abstrakci, která se intuitivně pojí s vybránym datovým typem. Muselá by se vytvořít pro různé velikosti poli samostatná funkce. To se muže řeší předaním pole přes pointer a buď ukončením pole nějakým specifickým symbolem, nebo předáním doplňujícího parametru délky.

V takovém to připadě se však ztraci jaka koliv vyhoda vybraného datového typu, a vlastně i konceptualní smysl onoho. Kod je ve vysledku méně explicitní, a navíc více nadolný na chyby, jelikož compile-time informaci, ktera se poji jen k jedne proměnné, se rozvádí do dvou run-time.

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

4.5.1 Délka pole

K získaní delky pole jsem zavedl nasledujicí syntaxi.

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

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

4.5.2 Typy poli

Po mimo klasického rozdělení poli na statická a dynamická, bych chtěl umožnít jejich dělení v zavislostí na variabilnosti délky. To by umožnílo vytvařet funkcím vice 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 mohlá přijmout jak statické, tak i dynamické pole.

Navíc bych chtěl integrovat array list do jazyka v ramci poli, jelikož je to často využivaná struktura. Array list bych viděl jako automaticky rozšířovatelné 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 delce 8. Spočtená délka by se vždy dosazovala compile-time, realná proměnna by se pro jeji uchovaní negenerovala.

Pole konstantní run-time známé délky

Jedna se o analogii alokace konstantního ukazatele v C, ktery by byl využivan jako pole.

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

Tedy

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

by alokovalo pole o delce 8 na heapu a vygenerovalo by přislušnou proměnnou pro uložení délky někde v pamětí programu. Pole by, samozřejmě, nešlo realokovat, jelikož delka pole je obecně run-time znama, a tedy není možnosti ověřit při kompilaci jeji 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 delce 8 na heapu a vygenerovalo by přislušnou proměnnou pro uložení délky někde v pamětí programu. Pole by šlo realokovat.

Array List

Šel by vytvořít nasledovně

```
int[] arr;
```

nebo se specifickou počateční delkou, v tomto připadě 8.

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

Volba kvalifikatoru

Lze take vnímat dynamické pole za výchozí, a ne array-list. Pak by se využíl kvalifikator při inicilaizací array-listu, pole variabilní run-time známé délky by se inicializovalo bez kvalifikatoru. Takovým kvalifikatorem by mohl byt třeba `auton` od slova autonomus.

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

Příklady jiných jazyku

C++ ponecháva vše jak v C. Řeší vše za pomoci konteineru definovaných v `std` knihovně. Nema však analog dynamického pole, ale nabizí std::span, který muže sloužít jako interface pro souvislý kontejner, nemuže 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 připadech delka dostupná jako
arr.length;
```

Odin ma k dispozici statická pole, slicy, ktere se využivají obdobně jako std::span, ale umožňují alokaci, a tzv dynamicka pole, ktere je analogem array listu pro jehož deklaraci se využívá kvalifikatoru na mistě délky. Všechny maji dostupnou funkcí `len` vracejicí jejich delku a dynamické pole muže využit funkce `cap` k získaní realné 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 zdurazníl rozdíl mezí nimi i z pohledu prace s daty. Pole je na rozdíl od ukazatele kus pamětí obsahujicí stejně elementy za sebou.Tedy bych pole vnímal jako proměnnou definujicí stejné prvky. Akce provaděne se samotnou proměnnou jako takovou (bez indexace), by se tykalí všech prvku v poli.

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

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

se vytvoří kontejner pro osum proměnných `arr[0]` až `arr[7]` o datovém typu `int`, který se specifikuje jen jednou. Použíje-li se kvalisifikator, 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 dusledku dvojakosti nasledující přířazení není dovoleno

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

jelikož pak není jasny zda se jedna o přiřazení všech prvku `arr2` do `arr1`, nebo přepsaní ukazatele `arr1` na ukazatel `arr2`. Což není z přečtení kodu zjevné, jelikož standardně se využivá kvalifisikatoru const k zakazaní přiřazení do proměnné.

Navíc, syntaxe řešící onen problem již v jazyce je, což přinaší jetě špetku zmatku.

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

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

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

```
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 zapsaní 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 zmiňěno viz.[]. Tedy vlastně se s každym řetězcem pracuje jako s ukazatelem.

Jelikož já definují pole jinak, tak lze rozvinout možnosti pole, aby umožňovali ve vysledku lehčí praci 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 rozšířít podporu literalu z ASCII na jiné kodovaní, které by umožnílo jednoduchou manipulaci se složitějšími symboly. Jako takové kodovaní bych volil UTF-8, protože je kompatibilní s ASCII a blokem je byte, tedy neni zavísle na edianech a je velmi rozšířene.

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

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

To umožní pracovát se symboly samostatně, využívát všechny vyhody pole a mit pro ASCII text stejně velké pole jako v C. Tedy, například, leva strana muže 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 jazyku

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, '.') }
// vypiše "č .a .u ."
```

V Zig jsou řetězcové literály ve skutečnosti pouze ukončené nulou bajtová pole, která lze zapsat v kodu pomoci UTF-8 symbolu. Další rozšíření prace je dostupné prostřednictím standardní knihovnu.

```
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, ktere by se měly integrovat do syntaxe, bych viděl zřetězení a vyběr podřetězce. Ostatní operace by už měly byt obsažené v standardní knihovně.

Zřetězení

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

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

Jelikož delká libovolného pole je získatelná, lze to zobecnít na jaky koliv typ pole. Je však duležité, aby implementace neobsahovalá žadnou alokaci pamětí, bylo by to zavadějicí. Tedy připadné vysledné pole by se muselo samostatně standartně alokovat prostředky jazyka.

Pro dynamické pole by to mohlo vypadadat nasledovně

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

Vyběr podřetězce

S podřetězce nebo výřezy polí se lze setkat v různých jazycích v ruzných 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 nasledujicím zpusobem

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

kde `dataPtr` by ukazoval na nějaký element v puvodním poli, a `len` by specifikovalo délku. Odin a Zig například implementují vyřez (slice) pravě jako pointer na data a delku.

Odin, Zig a D například vnímají vyřezy jako datové typy a například je využivají jako nějaky interface pro dynamická pole.

Já bych nevnímal vyřez jako samostatný datový typ, ale spiš jako operací nad polem. Vyřezy 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 organizací kodu. 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 definovany název a s tím i možnst kolizi. Proto je vhodné například umožnít pojmenovování jmenných prostoru ze strany importujicího. Opět, například jak v Pythonu:

```
import foo as boo;
```

Jmenné prostory bych viděl jednoduše jako pojmenovaný scope.

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

Pro přístup k prvkum bych využíl syntaxe z C++.

```
Foo::x;
```

4.8 Systém importu

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

Základním celkem bude soubor, jelikož jeto to co se ve vysledku předá překladači. Překladač dostane jen jeden vstupní soubor, ktery nasledně 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.

Intitivně se nabizí možnost přímého importu souboru nasledující syntaxi:

```
import filename;
```

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

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

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

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

Syntakticky se specialně specifikuje `namespace`, protože jsem se rozhodl onen konstrukt využít k implementaci jiných zpusobu zabalení souboru. Napřiklad:

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

Dále tento konstrukt lze rozšířít a zavest import jen vybraných symbolu ze souboru.

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

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

Lze buď vycházet z toho, že vše je viditelńe, 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 importu. Tedy, importuje-li soubor identifikátory z jiného souboru, budou-li viditelné také. Zřejmé je, že pokud jsou přistupné při importu, tak by měly být přístupně i pro další importy, jelikož jsou na stejné úrovní jako kód souboru, a nekladlá se žádná omezení.

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

K označení lokalních importu bych použíl slovo `local`

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

Samotná klasifikace proměnných dle viditelnosti by se mohlá kdyžtak řešít za pomoci direktiv. Napřiklad:

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

4.9 Přetěžování funkc

I když se jedna o implicitní konstrukt, který skryva od čtenáře pravou identitu volané funkce, tak přinaši, z mého hlediska, jednu zasadní věc, zjednošuená jmena funkcí.

Tedy, zamisto třeba vepisovani datového typu do jmena funkce, k rozlíšení funkce, lze jen uvest jeji činnost.

To zjednodušuje vnimani samotného programu, jelikož při praci s vlastnimi datovymi typy, ktere definuji složité objekty, jmena funkci budou už znatelnou zatíží, oproti např. `maxi`, `lmaxf`, `lmaxu`, kde lze přibližně vydedukovat typ očekavane proměnné.

Navíc jmena samotných funkci s použitím identifikujicích postfixu/prefixu, nebudou samostatnými celky z hlediska nastroju pracujicich s kodem, tedy v zakladu samotným kompilatorem 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. NNebo spíše, je zavádějící stejně jako smyčka `for`, která místo abstraktní intrukce `for` provádí skoky a sem a tam. Smysl čtenář získáva ze samotného názvu funkce a vstupních proměnných, přičemž samotná konkretní funkcí je pro něho jako černoá skříňka. I když ona funkce dostava `int`, tak nelze vědět, jestli ten `int` neni hned první instrukci přetypovan na `float`. Tedy jediné co to ovlivní je rychlost nalezení spravné funkce při potřebě se podivát na jeji kod, což, bez užití LSP, bude zřejmě delší. Nicmeně, neřekl bych, že se jedná o něco zavažného.

Z mého hlediska, je lepší ho mít, než nemit. Zbyva tedy rozhodnout, zda povolit implicitni přetěžování, jestli muže platít

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

nebo pro jiny datový typ musi byt explicitní přetypování:

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

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

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 jmeńné stopy. A to nemluvě o tom, že vlastně tatež informace existuje dvakrát: jednou při definici a podruhé při volání.

Bylo by tedy vhodné mit implicitní přetěžování, ale s opci v jistých připadech vyžadat přesnou kontrolu datových typu při vyhledavaní funkce. Zvolil jsem nasledujicí symboliku:

```
foo!();
```

Využití prapodivného symbolu v tomto připadě není zavadějicí, jelikož očekavané intuitivní chovaní vyrazu se nemění. Jedna se stale o function call, ktery nijak nemění

vysledky volaní ani jeho vstupy, z hlediska čtenaře je prakticky irelavantní.

4.9.1 Přistup jiných jazyku

Odin obsahuje pouze explicitní přetěžování, jelikož jazyk umožňuje definovat vnořené funkce ve funkcích, a tudíž rozlišení konkretní funkce, ktera se ma zavolat, není trivialní.

Zig nema přetěžování funkcí, ale podobného chování lze docilit 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++ maji implicitní přetěžování funkcí.

4.10 Správá chyb

Uvažuje-li se C, tak jazyk nenabizí 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 nějaky system pro správu chyb, a jestli vubec.

Při práci s libovolným kodem je pak nutne číst komentaře k funkcím, externí dokumentaci apod. To opět vede na problem, kdy důležita informace není současti strukturních elementu kodu, ke kterým by měly různé nastroje přístup. Navíc to postrada jednotnost, jelikož různé knihovny mohu řešit správu chyb vždy odlišně. Ve vysledném programu se tak bude muset řešít zbytečný problem – jak s tím naložit.

To vše mně ve vysledku vede k myšlence o přídaní standartního systemu pro spravu chyb.

Z metod řešení jiných programovacich jazyku standartizovánou spravu chyb lze v zasadě vyčlenít dva přistupy.

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

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

Tedy chybovy stav je prakticky jen další stav programu.

Try-Catch Využiva se system tzv. vyjímek, kde připadne chybové místo je zabalene do `try` bloku, a připadna chyby odchycena do `catch` bloku. To umožňuje například nezatěžovát kod spravou chyb a psat ho v `try` bloku tak, jako kdyby žadná chyba nastat nemohla. Odchycené chybá se nasledně 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 požadavku

Neprve bych si definoval požadavky na chybový system:

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

4.10.2 Implementace

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

Jelikož funkce mají k dispozicí jen jednu navratovou proměnnou, chyba se bude vracet samostatným kanálem. Ovšem, nechtěl bych vnímat chybu jako přímo navratovou hodnotu určenou jen pro chybu, jak je tomu např. v Go. Protože pak se pro každé volaní funcke musí řešít dvě vystupní proměnné. To ve vysledku povede k vytvoření buď implicitních pravidel, nebo k mnohomluvné (verbose) syntaxi.

K bližší předstě uvedu nasledujicí příklad v Go, symbol `:=` vyjadřuje, obdobně jako v Pascalu, definici s inicializaci.

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

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

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

Zde neni zcela zřejmé, co se ma dít. Prvně provadíme definici `val1` a `err`, a nasledně, v tymtyž scope provadíme definici `val2` a opět `err`.

Samozřejmě, je to zoohledněné pravidly jazyka, kod je kompilovatelny a nová definice `err` se neprovede. Ovšem, dochazí ke sporu syntaxe a semantiky, kde ze syntaktického hlediska se `err` chova jen jako druhá navratova hodnota, ale ze semantického se implicitně provadí 'vyjimky' v pravidléch, jen protože je to chybova hodnota.

Navíc se to kompilkuje přidaním kvalifkatoru. Bude-li se chtit označit `val1` jako `const` ale ne `err`, nebo naopak, budeme li chtít mit jedno `embed` a druhý `const`, atd. To vše lze řešit na ukor upovidané syntaxe, budeme-li chtit byt explicitní, nebo přidaním implicitních pravidel. Proto se pokusím najit jiné řešení.

K navratu chyby bych tedy využil pravé strany příkazu. To umožní oddělít syntaktický samotný přikaz a ošetření chyby. Navíc to muže do budoucna umožnít odchycení chyby nejen z jednoho volaní funkce, ale i z libovolného vyrazu, ktery by mohl obsahovat několik volaní funkcí.

Navrholval bych nasledujíci syntaxi.

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

Kde se připadná chyba uloží do proměnné err.

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

Kuriozně se lze v takovém to připadě dopustít jedné vyjimky – pominutí samotné defince chyby před odchycením – jelikož je redundantní, místo odchytu totíž muže přiřázovat jen pracovát s datovým typem error.

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

Množiny chyb

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

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

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

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

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

Pak `ErrorSetA` je množina obsahujicí prazdné množiny `ErrorA` a `ErrorB` a `ErrorSetB` obsahuje množinu `ErrorSetA` a prazdnou množinu `ErrorB`. Libovolná s těchto množin by měla byt identifikovatelná svým jmenem a byt přiřazena do datového typu error.

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

K definici chybového rozharní funkce se pak použije nasledujicí syntaxi.

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

Funkce `foo` pak muže vracet chyby definovane v `ErrorSetB`.

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

Toto řešení je jednoduché a relativně všestranné. Umožnuje například rozvít některou prazdnou množinu na plnohodnotnou, aniž by se rozbil kod využivající onu množinu. Ovšem, ma jeden zakladní nedostatek – vracíme pouze stav. Tedy nemužeme vratít inforamci o chybě. Teoreticky je to řešitelné přidaním nových stavu, ovšem to zdaleka není prakticke.

Prakticky to omezuje jen při logovaní chyby, protože jinak vždy popisujeme stav programu, ktery je nezbytny z hlediska jeho činnosti. Tudíž přidani v takových to připadech chybového stavu je vlastně nezbytné (uvažuje-li se, že tento stav je vhodné vnímat jako chybovy, obecně to lze řešít normalní cestou).

Ve vysledku je to jen něco, co slouží jako doplnění systemu. Něco, co je využiváno přímo při zpracovaní samotné chyby, a tedy neruší samotnou standartizaci, která se kladla za cil, protože popis samotné chyby už není obecně standartiyovatelný, a tak čí onak se jedna o konkretní zaležitost.

Pokud by se navrhnutý model zobecnil definici chyby za pomoci struktury nebo unie, tak vlastně dojde k rozporu se standartizaci. System se totíž zobecní natolik, že bude moct byt využívan i pro jiné věci, a mnohy zpusoby. A tudíž se vlastně postaveny problem nevyřeší, jen se přesuneme jinam.

Mohl bych povolit přiřazení chybam konkretních hodnot, což by mohl byt postačujicí kompromis. To umožní pak indexovat pole hodnotami chyb, což je ve vysledku velmí silný nastroj.

Navrat chyby

Možnost v chybovém stavu vratít i normalní hodnotu z funkce je zručna zaležitost. Muže posložít jako napřiklad doplňujicí informace o chybovém stavu. Navíc, je to dokonce nutná zaležitost, jelikož vnímame chybu jen jako další stav, a ne jako něco zvlaštního.

Volil bych nasledujicí intuitivní syntaxi:

```
return value, err;
```

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

Pak navrat jen hodnoty je nezměnný, ale otázkou je navrat jen chyby. Lze k tomu přistoupit tak, že vlastně takovy to připad existovat nebude, tedy vždy spolu s chybou se vratit i hodnota. To take zaruči, že proměnna, do ktere se zapiše navratova hodnota, nebudeme mit neurčenou hodnotu. I když je to skvělé chování, nelze ho použit. Ma-li byt jazyk nízkoúrovňový, musí take dat programatorovi i kontrolu. Nelze jen tak zbytečně vnucovat instrukci. Tedy, přidal bych symbol, např _ definujici přeskočení proměnné, a skončíl s nasledujícím kodem:

```
return _, err;
```

Implementace v jiných jazycích

4.11 Kontext

Koncept, ktery umožňujicí měnít chování kodu dle svých potřeba a tím i znovvyuživat kod pro konkretní potřeby. Ovšem, když je kontext explicitní, tedy je předaván jako parametr funkce. Pak je to jen na konkretní knihovně nabidne li vubec kontext a když ano, tak jaky.

Implicitní kontext, ktery integrován do jazyka pak umožňuje řešit onen problem. Obzvlášť je to výhodne v jazykach s manualní kontrolou paměti, kde za pomoci kontextu se dá řešit problem použití vlastních alkoatoru. Takovými jazyky jsou třeba Odin a Jai.

print, mem alloc, etc.

4.11.1 custom alloc

4.12 Compile-time exekuce

V ramci optimalizace kompilatory mohou provádět provadět vypočty některých výrázu pokud je dostatek informace. Tedy například:

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

Výraz definující x se muže předpočitat a za běhu programu se nebude muset níc vypočitavat.

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

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

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

Implementačně by proměnna neexistovala, ale vždy by se využivala jeji spočtená hodnota.

Takový to jednoduchý klasifikator pak umožní provadět velmí složíte compile-time vypočty. Protože ze své podstaty embed specifikuje, že proměnná má byt spočtená při kompilaci, fakticky tedy jaka koliv nebyla pravá strana výrazu, kompilator se ji pokusí vypočíst, a buď uspět, nebo ukončit kompilací 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 specialně předeklarovat jako compile-time jako v C++:

```
fcn 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 spočist pravou stranu:

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

Nebo třeba v Zig parametry funkce comptime umožňuje parcialně comptime-runtime 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 kompilatoru

K implementaci kompilatoru jsem použil jazyk C++ využivajíc standartu C++20. Jako cilovou a vyvojovou platformu jsem zvolil Windows, kde jsem využil Visual Studio předkladače (cl) a Visual Studio 2022 k ladění. Jako IL jsem vybrál jazk C, protože jsem s ním a s nastroji pro pro práci s ním dobře znám. Pro překlád generováného C kodu jsem zvolil Tiny C Compiler (TCC), a konkretně jeho knihovnu libtcc, ktera umožňuje vestavenou kompilaci C kodu.

Jedná se o konzolovou aplikaci, která prácuje se soubory na disku. Tedy, zavislot na operačním systemu není tak velka a std knihovná z větší častí pokryla rozdily Windows a Unixu. Po mimo rozdílu v systemových funkcích, jako třeba získání cesty k spustítelnému souboru, která nejde získat skrz std::filesystem, se hlavně líší knihovný a cesty potřebné pro run-time libtcc. Protože se mi podařílo program uspěšně zkompilovat pod Ubuntu 22.04.5.LTS, port pro Unix by tedy měl byt jednoduše realizovatelný.

5.1 Strukturá

Strukturá aplikace je nasledujicí. Nejprve je zpracován uživatelský vstup ve formě argumentu z přikazové řádky. Ten je převeden do konfigurace modulu Compiler. Modul Compiler je reprezentován jmenným prostorem, a tedy je staticky. Modul provádí v zasadě čtyří věci postupně. Nejprve běži parser, který inicializuje a zaplní AST. Dále běži validator, ktery již provede semantickou kontrolu AST. Nasledně je kod přeložen instanci modulu Translator. V zavislotí na uživatelském vstupu vystup překladače muže byt převeden do binarní podoby a popřipadě hned spuštěn.

Cílem bylo dospět k robustné implementaci, ktera by dokazala vytvořit zakladnu pro dalnější, řekněme, idealní, implementaci kompilatoru, ktery by byl specificky pro danny jazyk. Tedy, hlavním ukolem bylo přijit s AST, které by se dalo využit jak pro překlad do libovolného IL, tak i které by bylo možné využít ve věstaveném interpreturu.

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

5.2 Uživatelské rozhrání

Pro komunikaci s uživatelem se využívá argumentu přikazové řádky. Uživatel má v zakladu tři hlavní přikazy nasledujicí jménem vstupního souboru, na který maji byt aplikovane:

build Přikaz sestaví program do spustitelné podoby.

run Přikaz sestaví prográm do spustitelné podoby a bezprostředně ho spustí v terminalu.

translate Přikaz pouze přeloží do vybraného IL.

Nasledně mohou byt uvedené doplňujicí možností dostupné pro každy přikaz:

-ol (Output Language) Vybere, ktery 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 informáce.

-h (Help) Vypiše napovědu ohledně uživatelského rozhrání.

-b (Batch/Bash) Říka překladačí, že je spouštěn ne přímo uživatelem, ale za pomocí skriptu.

Samotná implementace samotného rozhrání byla trivialní, pouze za pomoci for smyčky a funkce strcmp. Žádné hashování se neprovadělo. Jedinou zajimavou časti byla implementace přikazu run.

Přikaz run potřeboval umožnít spustít zkompilovaný program po kompilaci. Lze třeba použit std::system, ovšem, v takovém to připadě nelze ukončit překladač a dalé používat stejný terminal pro komunikaci se spuštěným programeme. To mi nevyhovovalo.

Na POSIX systemech lze využít funkcí fork a exec z <unistd.h>, jejichž po sobě jdoucí volání nahradí běžicí process za zvolený. Windows však nenabizí obodbu funkce fork, k dispozicí má pouze funkcí CreateProcessA, která se dá využít k vytvoření processu s konkretním nastavením. Ovšem, nelze, nebo alespoň se mi nepodařilo, docilit obdobného chování jak u fork a exec. I když samotný process pro ukončení hlavního programu dale komunikuje s terminalem, samotný terminal se po ukončení puvodního processu vždy vypiše prompt.

Tedy, na Windows, pro docílení idelaního chování, kdy spuštěný program používá stejný terminal a na venek se to tváři jako stále jden program lze jen za použití třetího programu, ktery by sloužíl prostředníkem a nejprve spustil kompilator a nasledně, v zavislostí na vystupu onoho, zkompilovaný program. Takový to program jsem implementoval jako bat skript.

5.3 AST

Centerm 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 informaci, ktera 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;
    ...
};
```

Každy uzel teda ve své podstatě nese informaci o svém typu, rozsahu platností a lokaci ve zdrojovém kodě.

Každy konkretní uzel pak dědí SyntaxNode, obdobně třeba vypadá uzel reprezentujicí while smyčku:

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

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

I přesto, že SyntaxNode představuje uzel stromu, tak neslouži k vyjadření všech syntaktických prvku. Samostatnou reprezentaci maji výrazy. Je to dané tím, že vnítřně vyrazy představují vyjadření hodnoty proměnné. Tedy v AST je vždy vyraz zastoupen proměnnou.

Obecný výraz je definovan velmi jednoduše:

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

Pak konkretní výraz vypadá například nasledovně:

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

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

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

Vyjadření hodnoty proměnné je pak představeno nasledovně:

```
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 vyjadřená odkazem na definici, přímo hodnotou, nebo výrazem. Samotná hodnota se využívá i pro určení datového typu operandu a vypada nasledovně:

```
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 použitá kvůli interpreteru, který s touto strukturou pracuje. Podrobnějc oné atributy budou představené v přislušné 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 zpusob vypisování chyb libovolným zpusobem, měl bych omezeny přistup k buffrum souboru, neměl bych kontrolu nad paměti atd. Navíc bych nevyužíl vyhod při prototypování syntaxe, protože mam konktretní typ jazyka, který chcí implementovát, tedy bych jen mohl implementovát nutnou abstrakci pro zaměnu syntaktických celku.

Při implementaci parsru jsem se rozhodl nepoužit abstrakci ve tvaři lexru, jelikož mně jen zajimalo jaky to je, protože ve většíně publikaci se lexer používá a chtěl jsem si udělat nazor jinou implementaci.

5.4.1 Prace s paměti

Obecně jsem se rozhodl zatím neřešít dealokaci AST, protože překladač při chybě končí svojí činnost, a tedy bude rychlejší to přenechát OS. Ovšem, protože v budoucnu se nejspiš bude derivovát z překladače implementace LSP, tak by je nutné, aby veškera paměť, která je alokovaná a není součástí vysledného AST nebo jiných statických částí, byla dealokovana.

Každy soubor se vždy načte cely do pamětí a je obsažen v ní až do samotného ukončení programu. Je to nutné, protože v libovolný okamžík je nutné moct uživatelí vypsát nějaky relevantní kus kodu. Protože samotný nejpreve je sestaveno celé AST a až nasledně se provádí validace, tak se nemužou soubory postupně dealokovát.

Abych toho využíl, rozhodl jsem se ne alokovat paměť pro textové řetězce, ale používát vždy přislušného ukazatele do přislušného souboru.

5.4.2 Importy

Zvnějšku parser je jen jedná funkce přijimajicí na vstup jméno souboru, ktery oná vníma jako vstupní soubor programu. Vnítřně globalně parsing řeší dvě funkce, parseFile a processImport, které za pomoci structury ImportNode řeší parsíng všech dálších souboru a spojení jich do vysledného AST.

Každy soubor se parsuje do předem dodané instance Scope. Při parsování souboru se připadné importy zařazují postupně do přislušného uzlu importovácího stromu reprezentováného strukturou ImportNode. Importovácí strom slouží k zachování vztahu souboru a kontrole cirkulárních importu.

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

Aby se předešlo parsování znovu zpracování téhož souboru, každy rozparsovaný soubor se označíl unikatním id. Id bylo definovano nasledovně:

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

Jednalo se tedy o reprezentací souboru jeho velikosti a časem vytvoření. Takový to přístup umožňoval rychle porovnovát ruzné soubory. Jinak by třeba šlo použít absolutní cesty k souboru, ovšem to mi přišlo nešikovné, protože obecně cesta muže byt všelijak dlouhá. Každy soubor byl pak ukladán do mapy a před parsingem se ověřoválo, jestli již existuje nebo ne.

Algoritmitický by se pak funkce processImport dala popsát nasledovně:

```
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 nezaleží na pořadí, protože neosahují sam o sobě spustitelný kod a jejich definice nemusí předcházet použití. Tedy je lze prostě zařadít nakonec. V připadech, kdy to nejde, krom zřejmé změny kontejneru na třeba list, lze alespoň posunout prvky jen jednou za soubor udělav posun až po zpracování všech imporu.

5.4.3 Samotný parsing

Parser jsem implementoval procedruralně, kde každa funkce relativně odpovidala syntaktickému celku, ktery má za ukol rozparsovat. Každa taková funkce dostává pointer na buffer s textem a pointer na Location, které definuje lokaci, tedy hladvně index, ktery slouží jako přimy index do bufferu a číslo řadku.

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

Kličová slová se vnítřně namapovála na integralní hodnoty, což vytvořílo abstrakcí mezí samotnymi celky, které reprezentují a slový jako takovými. Navíc to umožnílo použití stejných funkcí pro zpracování ruzných množín slov. Napřiklad direktiv kompilatora. Pro zpracování kličových slov byl v zasadě použít jeden velky switch-case, protože mám rad switch-case. V budoucnu, ovšem, by ho bylo dobré nahradit indexaci pole, kde každy index by ukazoval na samostatnou funkci.

Operatory se reprezentovaly jako uint32_t, což imezoválo délku operatoru na 4 symboly (po připadě by šlo rozšířít na 8 za použití uint64_t), ale umožnílo indexaci. Omezení na malý počet symbolu pro operator není faktický omezujicí, protože větši-nou operatory jsou reprezentovany matematickymi symbolami nebo kratkymi slovy. Vyhoda je však vyznačná. Look-up operatoru mohl byt proveden nasledovně:

```
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 obohadit AST o potřebnou semantickou informaci. Ve vysledku by AST mělo byt bezchybné a plně odpovidat navrženým podminkam. 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 mužou slepě duvěřovát AST.

K validací se využívalo cachovaných odkazu na konkretní typy uzlu. To bylo prováděno lokalně v ramci rozsáhu platností a globalně pro celé AST. Tedy se nemusel procházet cely strom, ale mohlo se přímo sekvenčně přistupovát k většíně duležitym prvkům.

Během validace se provaděly nasledujicí akce:

- propojení definic uživetlem definovaných datových typu s jejich konkretnimi užiti.
- vzajemné propojení chybových množín
- propojení chybových množín užitých ve funkcích s jejich definicemí
- validace definic uživatelských datových typu
- propjení proměnných s definicemi
- propojení a validace *goto* příkazu
- validace, že každá funkce má globální platnost
- propojení volání funkcí s definicemi
- vypočet všech compile-time proměnných
- vypočet, nebo převedení na výraz, délek poli
- kontrola argumentu volaní funkcí
- kontrola *return* příkazu
- kontrola správností přířazení a alokaci
- kontrola inicializaci

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

Data-flow

Duležitou otázkou bylo řešení správného přířazení proměnných k definicícm. V zasadě definice se daji členit do dvou kategorii. V jednom případě definice je dostupná pro proměnnou nachází-li se v rozsahu její platnosti. Ve druhém případě navíc musí platít, že definice předchází proměnnou.

První připad se dá řešít jednoduše, stačí aby strukturá definujicí rozsáh platností vždy měla kontejner pro uložení odkazu na vyskytované v ní definice. Vhodným kontejnerem je hash mapa, prtože ta zaroveň umožní zachovát identitu nazvu už při parsingu. Pak stačí v případě každé proměnné rekurzivně procházet rozsahý platnosti směrem ke kořenu a prohledavát kontejner na přemět odpovidajicí definice. Pseudokodem by to šlo reprezentovát nasledovně.

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

Druhý připad už byl o něco složitější, jelikož zaleželo na pořádí. Zřejmou možností by bylo postupovat obdobně, ovšem k uložení dat využít array-list. Pak v každem rozsahu platnosti se zachová pořádí. Jak lze ale chápat sekvenční prohledavání kontejneru textových řetězcu není zrovná rychlé. Navíc, by obdobný kontejner musel obsahovát nejen definice, ale i uzly reprezentujicí rozsahy platností pro zachování informace o pořádí mezí přechody z dítěté do rodíče.

Pokud by se situace zjednodušila a omezilo se jen na jeden soubor, tak řešít by to šlo na urovní parseru. Opět by se použilo hash mapy a každá proměnná by se testovála jako v prvním případě přímo při parsingu. V takovém to případě by v mapach neexistovali ještě definice, které by byli definovane za proměnnou. Ovšem, protože obecně chci mit možnost manipulovat se stromem na urovní AST, tak nemužu zaručít, že se pořadí uzlu po parsingu jednoho konkretního souboru nezmění. Navíc importy samotné narušují standardní tok definic Foo::Boo

Rozhodl jsem se tedy postoupit jinak a zavest nový atribut v SyntaxNode, ktery by definoval pořádí definic a ostatních potřebných elementu v ramcí Scope a vybral jsem pro něj ne moc vhodný nazev *parentIdx*. Smyslem bylo přiřadít při parsingu každemu potřebnemu uzlu index z pohledu jeho rodíče a při validaci při vyběru kandidata z hash mapy rozhodnout se v zavislostí na pořadí indexu.

Zde lze viděť schematickou ukázku již realné 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 muže zdat zavadějicí, protože se při manipulací s AST se uzly budou muset posouvát, tak je zde pár vlastností, které je nutné si uvědomí:

- Indexy jsou vždy relativní vučí Scope, a tedy se posune jen dilčí čast.
- Indexovát se musí jen specifické typy uzly, nemusí se procházet tolík prvku.
- Indexovát lze i do zaporných hodnot, jelikož se jedná pouze o nastroj k rozhodnutí o pořádí.
- Indexy se mužou duplikovát.

Například, u funkcí a jmenných prostoru nezaleží na pořádí, každa funkce tedy muže vždy mit index 0. Při importu rozsahu platností nebo jménného prostoru se žádná již existujicí proměnná nemuže odazat na jejich vnitřní proměnné a není nutno nic měnit. Pokud se importem přidavá definice na začatek souboru, což je zcelá běžné, tak se lze podívát na index stavajicího prvního prvku a použít index menší. A podobně...

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

Přetěžování funkci

Klasicky přiřazení volaní funkce ke správné definici se ničím neliší od přiřazení proměnné definice, postačí jen jmeno. V připadě přetěžování funkcí už roli hrají i datové typy a počet vstupních argumentu. V připadě 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 přiřadít první definice. Bude tak i například

v případě C++.

K určení nejvhodnější funkce jsem tedy použíl jakehosi score, ktere sestavovalo v zavislosti 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 zavislostí na kategorii se pak přičetla přislušná hodnota do skore. Kategorie a zaroveň i hodnoty byli vyjadřeny enumeratorem nasledovně:

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

Kde nejvyše hodnocenou kategorii je připad přesné shody. Dale nasleduje připad přetypování do stejného podtypu (napřiklad i32 do i64) bez snižení přesnosti. Na což navazuje obobný připad akorat se sníženim přesnosti (napřiklad i64 do i32). A nejmeně hodnocenymi jsou pak ostatní jiné implicitní přetypovaní.

Zde by šlo obohatit model o vice kategorii a rozlišovát zda se jedná o přechod od neznamenkového typz do znamenkového a naopak, nebo rozlišovát na kolik se zvětšíla nebo zmenšíla přesnost při přechodu z jednoho typu na druhy. Ovšem, takové to chování jsem prozatím shledal jako zavadějicí.

Tedy, pro každé volaní funkce se nejprvé v jeho rozsahu platností naleznou funkce se schodnymi jmény. One funkce se uloží do kontejneru fCandidates. Každa kandidatní funkce se projde a spočíta se jeji skore v zavislostí na volaní. Na konec se vybere funkce s největším skore. Chyba nastane pokud budou dvě a vice stejná maximalní score, nebo nezbude žadna funkce se scorem.

Vypočet datových typu a výrazu

Nejprve jsem chtěl provadět vypočty datových typu spolu s evaluaci výrazu v jedné funkci, ktera by se pokusila spočíst každy uzel vyrazu a při neuspěchu by jen vyjadřila datový typ. Při procházení výrazu jsem ovšem vždy byl ve stavu, kdy se mohlá očekavát neplatná hodnota, která měla byt zohledněna. Dospěl jsem tedy k tomu, že by bylo vhodnější ty funkce rozdělít i když mají fakticky stejnou logiku.

5.6 Interpret

Protože compile-time evaluace je zakladem jazyka a musí byt umožněno vyhodnocovát prakticky všechno, rozhodl jsem to řešít implementaci vestaveného interpreteru. Ukolem interpreta je možnost jak evaluace proměnných a funkcí, tak i libovolného operatora.

Jak bylo zminěno, interpreter využíva specialního atributu ivalue v Operand. Atribut se využiva pro ukladani mezivysleku interpreta po spočtení každého uzlu. Postupně se tak spočitá hodnotá finalního, vstupního, uzlu a hodnota se přepiše do cvalue. Ovšem, to nestačí, protože se také musí řešit evaluace funkcí.

Evaluace operatoru

Samotným jadrem jsou dilčí funkcí umožňujicí evaluaci konkretních operatoru. Funkce jsou zabalené do unie, ktera umožňuje sjednotit ruzné typy funkcí:

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

Funkce v poli jsou seřazene seřazené do skupin dle datových typu, kde každa skupina je seřazená dle operatoru. Pořadí určuje vždy přislušný enumerator. Obdobně třeba vypada funkce pro použití operatora, kde, lze vidět zpusob 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 připadě funkci se spočtená hodnota zapiše do ivalue každe vstupní proměnné. Funkce je pak teoreticky spočetná, ovšem zde se již musí pracovát ne s uzly výrazu, alse s libovolnymi uzly AST. Každy uzel tedy musí mit svou definovanou logiku jeho vypučtu. Obdobně vypada 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 muže obsahovát další volaní funkcí, nemohl jsem si vystačit jen s ivalue, a tak jsem přidal ještě atribut istack. Jak plyne z nazvu, istack měl umožnít ukladat hodnoty všech ostatních volání, každa funkce pak měla atribut istackIdx reprezentujicí index, ktery by měl byt použít pro získaní hodnot pro jeji kontext.

Funkce ovšem mohlá volát i samu sebe. Musel jsem tedy k atributu istackIdx zavest atribut icnt, ktery by počital každé takové volaní a těm samym sloužíl jako identifikace kontextu. U každé Value se pak nastavoval attribut hasValue na odpovidajicí index, a tak mohla byt provedena kontrola, jestli hodnota byla spočtena v nynejším kontextu. Pokud hodnota nebyla spočtena v dannem kontextu, tak se hodnota překopírovala do cvalue, v opačném případě se hodnota kopirovla zpět do ivalue.

I když to muže znít složitě, fakticky se jednalo o překopirování všech hodnot do cvalue, což lze udělat před voláním znovu sama sebe. Ve funkci se normalně pracovalo s ivalue a při navratu se hodnoty překopiroválí zpět. Akorta se to provadělo na mistě vypočtu, a tedy se kopirovali jen potřebné proměnné.

Protože ve vysledu by se musel po mimo hlavní časti udrožavat take všechny uzly v interpretueru a neustale je měnít při změnach či refaktoringu, tak jsem se rozhodl se jim moc nezabyvát dokud projek je v aktivním vyvojí. A tedy implementováno je pár uzlu, které slouží k demostrací funkčnosti řešení. Udržováné jsou jen aritmetické funkce, ktere se vnitřně využívaji třeba k vypočtu delek poli.

5.7 Generace C kódu

Překladač AST byl řešen na rozdíl od ostatních modulu abstraktně. Má totíž smysl umět překladat strom do ruzných IR, na rozdíl třeba od parsru, kterých nemá mit smysl několikero, jelikož pracuje vždy s jedním konkretním jazykem.

Překladačem je vlastně instance nasledujicí 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

Generovát vysledný kod jsem se rozhodl sekvenčně přímo do souboru. Proto jsem potřeboval několík souboru, abych mohl generovát ruzné časti AST do ruzných souboru a ve vysledku je spojit v potřebném pořadí.

Použil jsem nasledujicí soubory:

- main.c, byl použit jako hlavní soubor, kde byli přidané ostatní osatatní soubory. Zapisoval se kod, ktery bylo možné dat do main funkce.
- functions.h zapisovly se tam definice funkci, aby mohly byt dostupne napřič všemi ostatnimi funkcemi
- functions.c zapisovly se samotné definice funkcí
- typedefs.h zapisovaly se definice struktur a unii.
- variables.h zapisovaly se deklarace definice proměnnych, především globalních.
- foreign_code.c, veškery kod cizích jazyku, viz[]

pořadí....

5.7.2 Globalní rosah platnosti

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

Problematickou čatí byla definice statických poli, kde

5.7.3 Vykreslení pole

Nejspiš jedninou netrivialní věci při vykreslení bylo vykreslení vyrazu obsahujicích konkatinaci poli bez využití alokaci.

Implementaci jsem řešíl generaci for smyček. Myšlenka byla nasedujicí, v nezavislosti kde se vyskytuje oprator konkatinace z pohledu stromové struktury, každy takový vyraz vždy rozšiřuje vysledné pole a tedy se chová, řekněme, linearně a každy usek výrazu mezí operatory lze tedy reprezentovát for smyčkou.

Podivejme se na přiklad pro ilustraci.

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

Je-li na levé straně pole řadné velikostí (zajisti to je ukol validtora), tak lze postupně procházet výraz a zapisovát vyraz do for smyčky, kde na levé straně je ans s určitým odsazením a na pravou postupně zapisujeme vyraz. Smyčku uzavřeme pokuď narazíme na operator konkatinace. Pak posuneme offset levé strany o délku dilčího pole, které jsme připojovali a opakujeme.

Tedy, nejprve se nastaví vychozí 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, nastavime 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];
```

Vysledný vyraz je schodný s puvodním a dá se generovát sekvenčně. I když ukazano na statickým přikladu, dá se zobecnít na dynamicky.

Prakticky se generace realizuje retrospektivně. Tedy, nejprve je rekurzívně procházen výraz než se narazí na operator konkatinace, načež se vykreslí puvodní čast. To umožňuje získat délku pole před vykreslením aniž by delka byla uložena napříč všemí uzly ve vyrazu.

5.8 Vestavená kompilcae C kodu

Protože jsem chtěl, aby kompilator obsahoval konvenční možnost generace spustitelného souboru, tak jsem dospěl k integrací TCC kompilátoru. Obecně bych mohl použít třeba std::system("gcc build my code pls"), tak jsem jak nechtěl byt zavísly na něčem, co už má uživatel předinstalovany.

Opci by bylo buď distribuovát gcc, nebo i jiny překladač, spolu s kompilatorem, což není šikovný, prtože program pak nejde distribuovat jako zdrojový kod. Lepší cestou by bylo integrovát kompilator do kodu přes přislušnou knihovnu.

GCC

LLVM

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

Protože jsem nenašel moc rozumnou informaci o správném použití knihovný pro mojí potřeby, tak popišu použíty postup trochu detailnějc.

Samotná instalce knihovný je standardní, tcc obsahuje include a libtcc složky, které se musí předát compilatoru jako include path a lib path respektivě a jeden dll soubor. Knihovně potřebné pro samtonou kompilaci C kodu v run-timu jsou pak dosutpné ve složce lib.

Nejprve je nutné vytvořít instancí TCCState, předat potřebné cmd argumenty a nastavít vystupní programu. Zaleží 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);
```

Většína funkcí vrácí chybový stav zohlednění ktereho se v ukazce pominulo. K uvolnění resursu alokovaných tcc_new se použije funkce tcc_delete.

Ke kompilací C kodu

5.9 Spravá chyb a logovaní

Protože chyby a varování jsou fakticky jedinym komunikačním prostředkem kompilatoru s uživatelem a jejich kvalita je zcela zasadní, potřebovál jsem si vytvořít jednotny a robustní system chyb a definovat pravidla jednotná pravidla pro pro spravu a logovaní chyb.

Aby se docililo jednoznačnosti, definovla jsem si pro sebe pravidlo – chyba musí byt logovaná přímo ve funkcí vyskytu. Jinak se musí řešít jestli použitá funkce vracejicí chybu už provedla logovaní nebo ne. Navíc to umožňuje klasickou propagaci chyby až do mainu, kde se v každe funkcí, buď diky lenosti, nebo nerozhodností, se chyba neošetří a jen se předá dál.

Problemem při logování v mistě chyby muže byt ne vždy uplná znalost kontextu, ovšem, protože postupně parsujeme AST, tak lze připadně retrospektivně podrobnější informací získat. Navíc, vnější funkce muže v připadě nutnosti provest svuj doplňujicí log, pokud to z jejiho kontextu přijde vhodne.

Chybu jsem tedy reprezentoval jednoduše jako negativní integralní hodnotu, kde bezchybový explicitní bezchybový stav je 0. Každa funkce by tedy měla mit navratový typ int a vracet připadnou chybu. Pro každou chybu jsem definoval take standartní chybovou hlášku printf sytaxi. Hlašky byli umistěné v pole a indexovatelné absolutní hodnotou přislušné chyby.

Logovácí system se skladá z pár funkcí a statických hodnot pod namespacem Logger. Za pomoci bitových hodnot lze filtrovát typy hlašek, napřiklad potlačít informace a varování. Logovácí funkce pak umožňovly po mimo hezčího vypisu samotné hlášky take vypsát konkretní místo ve zdrojovém kodě v řadkovém formatě a podtrhnout nutnou část viz obr[].

Protože někdy chyba funkce nemusí známenat kompletní chybu parsingu, někdy se muže parser vydat jednou cestou, zijstít, že to nejdé rozparsovát a zkusít jinou cestu, tak je nutné umět se vyhnout logování. Pro takový to připad je v rámci Loggeru dostupná proměnná mute, kde každe vlákno muže nastavít vlastní bit v zavislostí 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 muže byt obecně složítá, a tedy i když řešení skrz samotný Logger je číste, tak neni optimalní. Lepší by bylo předavát potřebnou informací skrz vstupní proměnné funkce, v ideale zavést nějaky context a předavat obecně potřebné proměnné skrz něho.

6 Závěr