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.

12. 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

Seznam obrázků

Seznam zdrojových kódu

Seznam zkratek

TUL	Technická univerzita v Liberci
IR	Intermediate Representation
IL	Intermediate Language
GCC	GNU Compiler Collection
JVM	Java Virtual Machine
JIT	Just-In-Time (compilation)
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
AOT	Ahead-Of-Time
IDE	Integrated Development Environment
API	Application Programming Interface
VS Code	Visual Studio Code
Vim	Vi IMproved
JSON	JavaScript Object Notation
HTML	HyperText Markup Language
PDB	Program Database
DWARF	Debugging With Arbitrary Record Formats
GDB	GNU Debugger
LLDB	Low-Level Debugger
ELF	Executable and Linkable Format
WYSIWYG	What You See Is What You Get

Úvod

Dnes, v době, kdy člověk se spíš zeptá, zda něco „umí“ JavaScript, než zda na tom „běží“ Doom, zůstává jazyk C fundamentálním pilířem softwaru.

Ačkoli jazyk C mi vždy imponoval, malokdy jsem se v něm našel dělat vlastní projekty. Obvykle jsem sahal po jazyku C++, který nabízel některé moderní prvky, jež mi ve standartním C scházely. Nicméně, programování v C++ se vždy pojilo s frustraci narůstající s mírou použitých knihoven. Proto jsem si položil otázku, zda existuje alternativa – jazyk spojující filozofii C a odražejicí požadavky dnešní doby.

Odpovědí na tuto otázku byly jazyky Odin a Zig, které představují moderni alternativy k C. Nicméně jejich syntaxe se od C odklání směrujíc vice implicitním směrem v duchu Go. Pro mně však byla klíčová explicitní syntaxe C, která jasně specifikuje deklarace proměnných a vytvaří tím dojem jednoduchého a čitelného jazyka.

Ve vysledku jsem s těmito řešeními nebyl spokojen a zdálo se mi, že většina alternativ se spíše zaměřuje na nahrazení C++ a bezpečnost než na jednoduchý jazyk s plnou kontrolou nad pamětí, která mě na C tolik oslovila. Proto jsem dospěl k myšlence návrhu vlastního jazyka, což vedlo k napsání této práce.

V úvodní častí prace se dotknu teoretických základů týkajících se překladačů a programovacích jazyků a představím možnosti pro tvorbu nástrojů k zajištění podpory vlastního jazyka. Nasledně se budu věnovat samotnému návrhu jazyka, kde kromě zdůvodnění jednotlivých rozhodnutí se budu odkazovat na jiné jazyky a diskutovat jejich řešení. V závěru práce se zaměřím na konkrétní aspekty implementace překladače.

1 Překladač

Překladačem, nebo též kompilatorem, se nazve program, který převádí vstupní text do výstupního textu zachovávající význam, kde oba texty jsou zapsané nějakým jazykem. Samotný proces převodu se nazývá překladem nebo take kompilací. V kontextu programovacích jazyků jde o převod zdrojového kódu do jiného programovacího jazyka nebo přímo do strojového kódu.

Existence kompilátoru je zásadní pro libovolný programovací jazyk, 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 splyvají 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, např. Turbo Pascal.

Protože překladač je jen program jako každý jiný, může být napsán v libovolném programovacím jazyce a přeložen odpovídajícím kompilátorem. Dokonce může být napsán v jazyce, který sám překládá, a přeložen sám sebou — tento proces se nazývá bootstrapping. To vede k problému „kuřete a vejce“, který má v tomto případě jasné řešení, protože ve vysledku existuje stroj schopný vykonávat určitou sadu instrukcí. Typo instrukce vlastně tvoří jazyk, který je spustitelný a dá se 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. Obecně však dává smysl pouze jedna další vrstva, kdy se jazyk přeloží nejprve do tzv. přechodné reprezentace (IR — intermediate representation), a až poté do kódu pro konkrétní hardware. Účelem této abstrakce je vytvoření roz-

hraní mezi výrobci hardwaru a tvůrci jazyků. Část určená pro překlad do IR se označuje jako front-end a část převádějící IR do spustitelného kódu jako back-end.

Je nutno podotknout, že jak back-end, tak i front-end jsou samostatné celky, kkteré jsou implementovány pro specifické problémy nebo potřeby. Proto i jejich implementace mohou obsahovat vlastní front-endy a back-endy. Výrobci hardwaru či jazyků tak nemusí přímo implementovat podporu IR, ale mohou využít rozhraní existujících obecných back-endů a front-endů.

Samotná IR může být reprezentována buď jako rozhraní a objekty či struktury v programovacím jazyce, nebo přímo jako jazyk, tzv. mezijazyk (IL – intermediate language).[1]

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

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

LLVM IR Forma široce využívaná v rámci LLVM nástrojů, především pro účely optimalizace a kompilace. Jedná se o jazyk, který se nachází na pomezí C a assemblerem. Může být jak v standartní textové podobě, tak i přímo implementován v paměti programu.[2]

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

GCC GIMPLE Jedna z mezireprezentací využívaných GCC, která je klíčová při optimalizacích a generování kódu. Výrazy převádí do tříadresného formátu zjednodušujíc tím analyzu a transformaci kodu.[3]

```
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). Název je odvozen od skutečnost, že každá instrukce je reprezentována jedním bytem. Bytecode je využíván JVM k JIT (viz 1.2.3) kompilaci. Lze jej tedy spustit spustit na jakékoli platformě, na které je implementován příslušný JVM.[4]

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

```
    iadd
    ireturn
    .end method
```

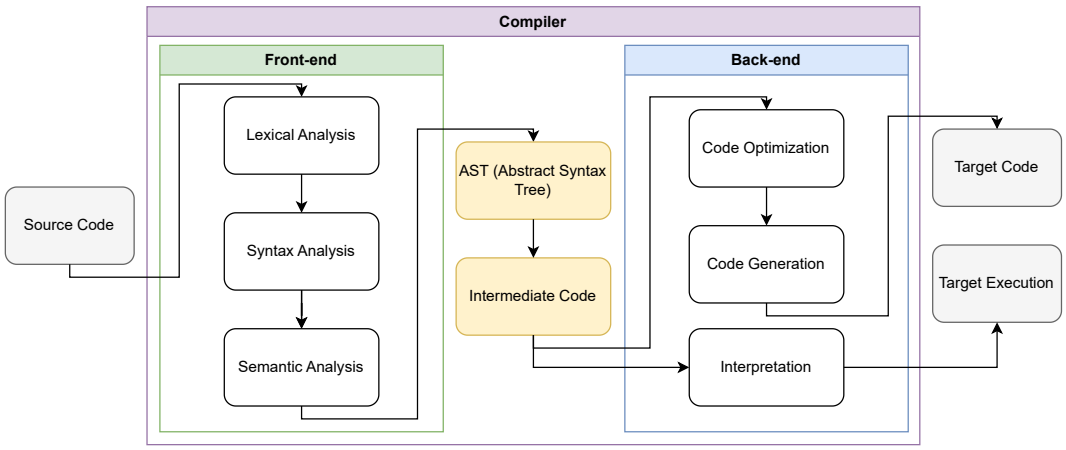
CIL Jedná se o zkratku pro Common Intermediate Language. Představuje obdobu Java bytecode, vyvinutou společností Microsoft. Pro spuštění CIL je nezbytná platforma podporující nějakou implementaci Common Language Infrastructure (CLI), jako je například .NET.[5]

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

C Samotný jazyk C může rovněž sloužit jako přechodná reprezentace, přestože nebyl pro to navržen[1]. Jedná se o jazyk s nízkou úrovní abstrakce využívaný v různých operačních systémech. Existuje pro něj tak velký výběr kompilátorů pro různé platformy a rozsáhlé množství dalších vývojových nástrojů.

1.2 Konstrukce překladače

Samotný překlad se může rozdělit do pár základních kroků. Nejprve je provedena lexikální a syntaktická analýza, kde čirý sled textu je převedena na abstraktní reprezentaci. Následně je tato reprezentace zvalidována podle příslušných sémantických pravidel. Validní reprezentace je následně převedena do zvolené IR nebo přímo do spustitelné podoby.[6, 7]



Obrázek 1.1: Struktura překladače

Obecně překlad může obsahovat více kroků, které se dále mohou štěpit. Nicméně,

vytknuté tři kroky jsou nezbytné pro jakýkoliv překlad. Možnou vizualizaci struktury překladače ilustruje obrázek [1.1].

1.2.1 Lexikální a syntaktická analýza

Cílem je převést zdrojový kód na základě gramatiky jazyka do abstraktní datové struktury v paměti překladače. Tato struktura je často reprezentována stromem, jelikož stromová struktura přirozeně odpovídá gramatické struktuře jazyka, a nese ustalený název AST (Abstract Syntax Tree).

Zde se nabízí zřejmá abstrakce mezi lexikální a syntaktickou analýzou. Kde modul lexikální analýzy se stará o zpracování vstupního textu a převádí slova na reprezentaci v paměti překladače nazývanou token. Syntaktická analýza pak může se slovy pracovat jako s abstraktními celky. Lexikální část se často nazývá lexer a syntaktická parser.

Zároveň se nabízí smysluplná abstrakce i mezi formální gramatikou jazyka a samotnou lexekalní a syntaktickou analyzou. Možnost formálního popisu jazyka za pomoci jistého standardu gramatiky (viz 2.1) umožňuje i existenci příslušných nástrojů napomáhajících při generaci AST.

Mezi takové nástroje patří třeba YACC a ANTLR.

YACC Neboli Yet Another Compiler-Compiler. Jedna se nástroj umožňující syntaktickou analýzu na základě formalní gramatiky jazyka (pro bližší představení samotného formátu gramatiky viz 2.1). YACC používá specifický formát připominající dialekt C. Jednotlivým syntaktickým celkům se dají přiřadit akce (funkce v C), jež se provedou při rozpoznání přislušné syntaktické struktury. Pro lexikální analýzu YACC využivá uživatelem definovanou funkci, přičemž standardně se využívá nástroj Lex. YACC ve výsledku generuje C kód (hlavně `yyparse` funkci), který se již používá v samotném kompilátoru. [8]

ANTLR Neboli Another Tool for Language Recognition. Jedná se o nástroj umožňující generaci parseru z gramatiky. Kromě generace samotného parseru dokáže ANTLR generovat i tzv. procházeče (visitors a listeners) stromu, které umožňují aplikaci vykonávat vlastní kód. Primárním cílovým jazykem pro generování parseru v ANTLR je Java, ale podporuje generaci i do jiných jazyků, jako C#, Python, Go atd. [9]

1.2.2 Sémantická analýza a Anotace AST

Během sémantické analýzy se provádí kontrola AST a doplňují nebo aktualizují se informace v jeho uzlech. Cílem je získat validní AST reprezentující původní text. Kontrola se může 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 přetížení funkcí, a podobně. Kromě validace se zároveň existujícím symbolům přiřazují odkazy na příslušné definice, je-li to relevantní z hlediska struktury navrženého AST. Například uzlu reprezentujícímu proměnnou se přiřadí odkaz na její definici.

1.2.3 Převod do finální podoby

V závěrečné fázi se AST transformuje do patřičné finální podoby. Obecně by pro každý typ uzlu v AST měla existovat odpovídající sekvence instrukcí pro jeho zpracování. Nejpřirozenější 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.

Způsoby transformace AST v závislosti lze rozdělit v zavisloti na finálním produktu.

Kód Výsledkem je kód v jiném jazyce, tedy buď v IL, nebo přímo strojový kód. V tomto případě buď překlad končí, anebo se předpokladá, že výsledný kód bude přeložen jiným nástrojem do spustitelné podoby. Může se jednat i o generování skriptů, které jsou pak součástí větších celku, jako jsou třeba herní enginy. Nebo se může jednat i o tzv. transkripci, jako v případě TypeScriptu.

JIT Ačkoli formálně spadá do předchozí kategorie, samotný koncept JIT kompilace je natolik vyznamný, že stojí za samostatnou zmínku. Zkratka JIT znamená Just-In-Time. Jedná se o metodu kompilace, při které je nejprve generovaná IR reprezentace, která je nasledně předána tzv. JIT kompilátoru. Ten pak přeloží IR do konkrétního strojového kódu mašiny, na které běží.

Důležitým aspektem JIT kompilace je umožnění specifických optimalizací kódu pro danou platformu a také možnost změny kodu za běhu programu. Na rozdíl od klasické, takzvané předčasné (AOT – Ahead-Of-Time) kompilace, která probíhá pouze jednou a pro obecně očekávaný hardware.

Interpretace Namísto generování výsledného kódu lze každý uzel AST přímo interpretovat. Tedy, namísto implementace funkce, jejíž výstupem by byl text v jiném jazyce, se přímo implementuje semantické chování uzlu. Takovéto kompilátory se zpravidla označují za interpretery.

1.3 Křížová kompilace

Někdy je vhodné přeložit program do strojového kódu jiného hardwaru, než na kterém běží kompilátor. Tomuto procesu se říká křížová kompilace (cross-compilation). Může k tomu docházet v případech, kdy je program vyvíjen na vysoce výkonném stroji s veškerým potřebným prostředím pro rychlou a efektivní práci, avšak vysledný software má odlišné cilové zařízení postradající takovou infrastrukturu. Příčinou může být operační systém, nebo i samotný hardware stroje.

Také se jedná o případy, kdy program je překladán i pro jiný operační systém, než na kterém je vyvíjen. Například, Doom byl vyvíjen na počitači NeXT s operačním systémem NeXTSTEP, přestože byl určen pro systémem MS-DOS.

Je zřejmé, že o křížové kompilaci má smysl mluvit pouze v případě kompilace do strojového kódu. V ostatních případech se jedná o kód, který je mezivýsledkem (například bytecode) a jeho spuštění závisí na jiném nástroji, který musí být sám přeložen pro cílovou architekturu.

2 Gramatiky

Při návrhu programovacího jazyka hraje důležitou roli samotná syntaxe, která ho z velké části definuje. Syntaxe totiž p5edstavuje jakési rozhraní mezi člověkem a jazykem, obzvlášť v případě programovacích jazyků, kde se v textových editorech či vývojových prostředích běžně různé syntaktické konstrukce vizualně odlišují. Proto je žadoucí mít možnost ji formálním způsobem popsat, a to jak z teoretického hlediska, tak i z praktického. Definice gramatiky jazyka může byt využítá v různých nástrojích, například, jak již bylo zmíněno, pro 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 neterminál.
- P je konečná množina pravidel.

[10]

Terminály jsou dále nedělitelné symboly jazyka. Jsou to například klíčová slova nebo jednotlivá písmena sloužící pro definici proměnných. Neterminály pak představují symboly, které se dále přepisují na jiné sekvence terminálů nebo neterminálů. Neterminál může například reprezentovat binární operátor, který nasledně bude definován pravidly obsahující již terminální symboly jednotlivých binárních operátoru. Prázdný symbol se označuje jako ϵ.

Obecně pravidlo gramatiky můžeme vyjádřit jako zobrazení: [1]

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

Tedy vzorem je posloupnost terminálů a neterminálů obsahující alespoň jeden neterminál. Obrazem pak je libovolná posloupnost terminálů a neterminálů.[10]

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

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

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

[10]

Z hlediska programovacích jazyků prakticky se lze omezit na gramatiky bezkontextové.[7]

2.1 Bezkontextová gramatika

Bezkontextovou gramatiku, kromě výše uvedené definice (2.2), lze také definovat na základě samotných pravidel, což bude názornější pro navazující text.

Definice 2.3. Gramatika $G = (\sigma, V, S, P)$ je bezkontextová právě tehdy, když pro každé pravidlo z P platí
$$A \rightarrow \gamma,$$
kde
$$A \in N, \gamma \in (N \cup \Sigma)^*$$

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

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

Pravidlo definuje nonterminál **`goto`** jako sekvenci terminálu `'goto'`, následovánou nonterminalem `identifier` (definováným v jiném pravidle představujícím identifikátor) a zakončeného terminálem `';'`.

[1] Hvězdička (*) představuje symbol libovolného opakování výrazu, a to i žádného.

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

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

`goto` pravidlo bude třeba generovat slova jako

goto FooLabel ;

goto Me ;

goto UnhandledException ;

Je zřejmé, že zápis pravidel může být různorodý. Pro sjednocení zápisu existuje několik standardních notací. Nasleduje přehled několika relevantních notací stručně představených na příkladu s **`goto`** příkazem.

BNF Zkratka pro Backus–Naur forma.

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

EBNF Rozšířená (Extended) Backus–Naur forma. Existuje několik verzí a má svůj ISO standard.

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

YACC notace Bližší seznámení s YACCem muže byt nalézeno zde: 1.2.1.

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

Složené závorky ohraničují C kód, který se provede při úspěšném parsingu přislušných syntaktických elementů. Neterminály a terminály z příslušných pravidel jsou přístupné za pomocí symbolu $. Například $ označuje proměnnou samotného rozpoznaváného pravidla, $1 proměnnou prvního symbolu pravé strany (`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 informace o ANTLRu lze nalézt zde: 1.2.1.

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

3 Vývojové nástroje

Kromě samotného překladače se při práci s programovacím jazykem běžně využívají různé nástroje usnadňující vývoj.

Základem je textový editor, bez kterého by nebylo možné samotný zdrojový kód v celku psát. Samotné editory pak, často prostřednictvím pluginů, umožňují přidávat podporu různých programovácích jazyků. Mezi takové populární editory patří například VS Code nebo Vim/Neovim. Lze je tedy využít jako platformu pro tvorbu jakéhosi integrovaného vývojového prostředí (IDE) pro vlastní jazyk.

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.[11, 12]

V případě Neovim je zde kromě klasických možností využití Vimscriptu, jak v případě Vim, dostupná možnost skriptování za pomoci integrovaného Lua script enginu[13]. Celé API editoru Neovim je pak přístupné prostřednictvím jazyka Lua. Lze tedy přímo přistupovat k bufferům a měnit rozhraní celého editoru. Pluginy jsou ve své podstatě jen zdrojové kódy, které se načitají při spuštění editoru na základě konfigurace. Obvykle za využití nějakého správce pluginů (například `lazy.nvim`), který umožní načtení složky s pluginem jedním řádkem. Standardním způsobem distribuce pluginů je pak git repozitář se samotným kódem pluginu, odkaz na který uživatel předá správci pluginů. Takto uživatel bude moct i jednoduše získávát aktualizace pluginu.

3.1 Zvýraznění kódu

Za základní standardní vlastnost se může klást zvýraznění kódu, které je dnes 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[14]. V případě Vim se používá vlastní formát definice zvýraznění syntaxe, který rovněž podporuje využití regulárních výrazu.[15]

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 (viz 3.2). Většinou totiž LSP je i tak aktivní a poskytuje například funkci doplňování slov. Tedy editor už obsahuje informaci o všech symbolech a jejich roli v jazyce. Oba vybrané editory mají vestavěnou podporu LSP[16, 17].

3.1.1 Dokumentace

Kromě zvýraznění synatxe v textových editorech je někdy potřeba 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[18]. 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. Výsledný HTML dokument pak neobsahuje žádný JavaScript běhový 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 potřeba napsánt skript, který by automaticky procházel HTML soubor a přepisoval zdrojový kód.

```
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 existuje opce využití vim.js[19], tedy portu Vim pro prohlížeče, který by mohl provádět zvýraznění syntaxe za běhu. 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 mezi jazykovým serverem a klientem, kterým může být IDE nebo textový editor. Jazykový server poskytuje klientovi informaci o zdrojovém kódě z hlediska sémantiky a syntaxe 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. Práce serveru je totíž od části shodná až do

fáze sémantické analýzy. Nicméně, kompilátor může obvykle ukončit překlad při první zjištěné chybě. LSP by naopak měl být schopen překládat i neúplně správný syntaktický a semantický kód a poskytovát informace o tom, co se podařilo převést do AST. Navíc, LSP by měl fungovat v reálném čase obnovující informaci o kódu po každém vstupu uživatele. Vývoj LSP tedy není triviálním úkolem i za podmínky existence překladače, jelikož jak překladač, tak i LSP vyžadují vyskou rychlost zpracování dat, ovšem jejich potřeby se protiřečí.

3.3 Ladicí program

Dalším klíčovým nástrojem pro vývoj programů je ladicí program. Zde opět lze využít vybraných textových editorů jako platformy. Ovšem psaní vlastního ladicího programu není zcela žádoucí, jelikož je to další aplikace, která se bude muset s jazykem vyvíjet a udržovat. Je proto výhodnější využít již existujících řešení prostřednictvím standardizovaných rozhraní.

3.3.1 Ladicí informace

Ladicí 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.[20]

Pro jazyk překladaný do strojového kódu pro účely ladění stačí mít vysledný 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.[20]

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.[21]

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. Standardně býva vestavěn do spustitelného souboru.[22]

Přímočarou možností je vlastnoruční generace těchto souborů. Naštěstí některé backendy umožňují generaci oných symbolů.

V případě LLVM IR lze třeba definovat podrobnější informace o původním kódu za pomoci maker `#dbg_value`, `#dbg_declare` a `#dbg_assign`[20]. Může to vypadat

následovně:[20]

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

První řádek zde představuje deklaraci proměnné `i` typu `int32_t`. Následující pak přidává oné deklaraci metadata. Dalších řádky specifikují detaily těchto metadata. Lze to použít jak pro generaci PDB, tak i DWARF[20].

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`[23], která umožňuje specifikovat číslo řádku a název souboru. Tato direktiva však ovlivňuje pouze bezprostředně následující řádek kódu, což lze řešit generací kódu bez konců řádků a přidáváním jich vždy při použití oné direktivy.

3.3.2 Integrace ladicích programu

Pokud je možné generovat ladicí informaci spolu se spustitelným souborem, lze pro ladění využít libovolného již existujícího ladicího programu, ktery by podporoval přislušný format.

Protože VS Code standardně poskytuje rozhraní pro integraci ladicích programu, lze pouze vytvořit vlastní konfiguraci pro již existující ladicí rozšíření a upravít konfiguraci překladu. Popřípadě lze tuto konfiguraci zabalit i do samostatného rozšíření.

Neovim nemá standardizované rozhraní pro ladicí programy, proto je konfigurace každého ladicího pluginu individuální, jestli je vůbec v konkrétním případě dostupná. Vždy ale lze udělat fork ...

4 Návrh jazyka

Nejprve bych vytvořil představu o vizi jazyka a objasnil svou motivaci. Vzhledem k tomu, že v zasadě je cílem přijit s moderní obdobou jayzka C, bude vhodné právě jím i začít.

Jazyk C mě v zásadě oslovuje svou jednoduchostí a mírou svobody vyjádření. Jazyk nabízí jen dostatečnou abstrakci nad assemblerem zachovávajíc představu o skutečném dění programu a neomezujíc přímou práci s pamětí. Příkazy jazyka neprovádějí skryté alokace paměti a s výjimkou `goto` neobsahuje skrytý tok řízení. Přímo z kódu je pak zřejmé, které instrukce budou provedeny a proč. Má explicitní a čitelnou syntaxi. Vždy jsou konkrétně specifikovány datové typy a modifikátory u deklarací. Nedochází k zneužívání neslovních symbolů, jedná se jen o operátory a závorky.

Jedná se o jazyk, ve kterém je zajímavě programovat, i když ne vždy je optimalní volbou pro rozsáhlá produkční řešení. Má však několik zásadních nedostatků, které mě ve většině případů odradzují od jeho využití. Například: samotné sestavení programu je příšerné, existuje neustálá duplicita informací v deklaracích a oddělených definicích, problémy s kolizí jmen při použití knihoven, makra atd.

Vycházeje z toho bych tedy viděl procedurální systémový jazyk pro všeobecné použití, umožňující robustní a explicitní vyhodnocování výrazů v době kompilace. Jazyk, který by umožňoval jednoduchou a neomezenou manipulaci s pamětí. Měl by být čitelný sám o sobě i na úkor osvědčených postupů. Interpretace kódu po přečtení by měla co nejvíce odpovídat skutečnosti. Například: deklarace proměnné by ve výchozím případě neměla být konstantní, protože po přečtení kódu, který nespecifikuje vlastnosti deklarace, je přirozenější se domnívat, že žádných vlastností nenabývá, než že má nějaké standardní skryté. Klíčovým aspektem je take intuitivní srozumitelnost syntaxe, která by neměla vyžadovat hluboké znalosti formální gramatiky. Preferováno je tedy vyjadření akcí spíše pomocí slov než abstraktních symbolů.

Navrhovaný jazyk není primárně určen k řešení konkrétního fundamentálního problému v nějaké specifické sféře. Jedná se o vytvoření nástroje pro mé osobní užití, jazyka, ve kterém bych mohl realizovat své programátorské záměry. Potenciálně by mohl oslovit i další jedince se srovnatelným náhledem na věc.

4.1 Předesloví k návrhu a přehled kapitoly

Obecně navrhovaný jazyk prošel oproti C větším počtem syntaktických a sémantických změn, než bude v této kapitole zmíněno, ale v zásadě principiálně zůstal podobný. Například funkce stále mají jen jednu návratovou hodnotu, je podporováno běžné implicitní přetypování mezi základními datovými typy, vestavěné číselné typy jsou definovány podobně jako odpovídající varianty dostupné v knihovně `<stdint.h>` atd. Pokud tedy nebude zmíněno jinak, pro porozumění následujícímu textu stačí předpokládat, že základní mechanismy fungují podobně jako v C.

Tato kapitola se zaměřuje především na popis výraznějších změn a nových konceptů, které nějakým způsobem řeší vybrané problémy nebo omezení jazyka C, případně zavádějí zcela novou funkcionalitu. Detailnější a kompletní přehled všech změn lze získat z přiložené dokumentace jazyka, formální gramatiky nebo testovacích ukázek kódu v příloze. (Externí dokumentace reflektuje aktuální stav vývoje překladače a může se tedy v detailech mírně rozcházet s tímto textem.)

V následujících sekcích této kapitoly se dále zaměřím na následující témat:

- Existující řešení
- Vstupní bod programu
- Alokace paměti
- Komentáře
- Pole (včetně typologie a vektorových operací)
- Řetězce (UTF-8, práce se symboly, nové operace)
- Jmenné prostory
- Systém importu
- Přetěžování funkcí
- Správa chyb (error sety, ‘catch‘)
- Exekuce za doby překladu (‘embed‘)
- Následující vývoj (budoucí směry)
  - Kontext (Implicitní kontext)
  - Metaprogramování (Pokročilejší nástroje)

4.2 Existující řešení

Při hledání moderní alternativy k jazyku C jsem zvažoval několik existujících systémových jazyků. Ačkoliv mnohé z nich nabízejí zajímavá a v určitých ohledech zdařilá řešení, žádný z nich jako celek plně neodpovídal mé osobní představě ideálního nástroje pro mé programátorské záměry – vždy jsem narazil na aspekty, ke kterým jsem měl výhrady. Považuji proto za přínosné v této sekci stručně představit ty nejrelevantnější z těchto jazyků a poukázat na klíčové body, v nichž se

jejich filozofie či konkrétní návrhová rozhodnutí lišila od mých preferencí a vedla mě k návrhu vlastního jazyka.

Rád bych podotkl, že případné výtky k těmto jazykům jsou formulovány z mého subjektivního hlediska a nevypovídají o obecné kvalitě daných vlastností, ale spíše o jejich (ne)souladu s mou představou a cíli tohoto konkrétního návrhu.

Vybral jsem jazyky, které považuji za nejrelevantnější ilustraci přístupů k řešení dané problematiky. Jedná se o jazyky C++, D, Zig a Odin. Existují i jiné, populárnější jazyky – Go a Rust – ktere na první pohled problém řeší:

Go Přestože se jedná o relativně jednoduchý jazyk vycházející z C, pro správu paměti se spoléhá na garbage collector, což neumožňuje plnou manuální kontrolu nad pamětí, která je jedním z cílů tohoto návrhu. Jako relevantnější představitel jazyka s podobnou filozofií[24], ale s manuální správou paměti, se pro toto srovnání jeví například Odin, který představuje spíše moderní pohled na jazyky jako Pascal a C.

Rust Rust, přestože je moderním systémovým jazykem, se od C výrazně liší svým primárním zaměřením na bezpečnost paměti (prosazovanou modelem vlastnictví a systémem výpůjček) a stylem bližším C++. Toto srovnání se soustředí na jazyky, které více následují filozofii C v oblasti nízkoúrovňové kontroly a relativní jednoduchosti. Role Rustu v poskytování silných abstrakcí je navíc v kontextu tohoto srovnání zastoupena jazyky C++ a D, které nabízejí větší flexibilitu.

4.2.1 C++

C++, bezprostřední následník jazyka C, který je s ním často nerozlučně pojen jako C/C++. C++ ponechává v základu C s drobnými změnami a staví na tomto základu za pomocí standardní knihovny. C++ tedy dědí i špatné vlastnosti jazyka C, jako jsou například makra a systém importu. Nové funkcionality v C++ často nejsou dosaženy fundamentálními změnami původních C konstruktů, ale spíše rozšířením standardní knihovny o nové šablonové třídy a funkce. To se týká i základních prvků, jako jsou pole. Zatímco C++ nabízí moderní a bezpečnější alternativy v rámci standardní knihovny (`std::array`, `std::vector`), samotná vestavěná pole, přímo integrovaná do syntaxe jazyka, zůstávají C poli se všemi jejich vlastnostmi a omezeními.

Jazyk obsahuje mnoho různorodých konceptů umožňujících řešit problémy paradigmaticky různými způsoby. I když se to může vnímat jako výhoda, a programátor si může vybrat podmnožinu vlastností, která mu vyhovuje, tak jen zřídka je veškerý kód napsán jedním člověkem. Například při práci s knihovnou, která řeší problémy objektově orientovaným způsobem, se musí potýkat i programátor, který sám nepíše objektově orientovaný kód. To vede k velmi nekonzistentnímu kódu. I samotná standardní knihovna, která implementuje mnoho zásadních vlastností jazyka, využívá metaprogramování a objektově orientované pří-

stupy.

Syntaktický je pak jazyk příšerný, protože kromě prolínání různých přístupu k programování se mísí i C a C++ kód. To znesnadňuje vnímání a čitelnost kódu, protože není vždy jasné, co se má dít, a jak objekty C++ nakládají s původními datovými typy a jak je interpretují.

Zdrojový kód 4.1 Ukázka syntaxe – unique_ptr[25] *Kód v jazyce C++*

```
#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
```

4.2.2 D

Jazyk syntaktický blízký jazyku C, zjednodušuje koncepty C++ a zbavuje se přímé závislosti na C. Import systému je řešen pomocí modulů, podobně jako v Javě. Jazyk nepodporuje makra a spoléhá na vyhodnocování v době kompilace a genericitu.

Ovšem, jedná se o objektově orientovaný jazyk s prvky skrytého toku řízení (např. try-catch), který jde spíše ve stopach C++ než C. Navíc obsahuje garbage collector, který sice lze zakázat, ale některé vnitřní operace ho stále budou používat [26].

Zdrojový kód 4.2 Ukázka syntaxe – paralelní inicializace polí[27] *Kód v jazyce D*

```
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");
    }
```

4.2.3 Zig

Jedná se o relativně moderní, jednoduchý, procedurální jazyk, který se inspiruje jazykem C. Jazyk staví na metaprogramování a exekucí prováděné v době kompilace nabizejíc široké spektrum možností jich využití. Neobsahuje skrytý řídicí tok programu. Práce s pamětí je manuální. Překladač nabízí různé varianty sestavení, které umožňují provádět doplňující bezpečnostní kontroly jak během kompilace, tak i za běhu programu. Navíc je podporována křížová kompilace programů.

Ve své podstatě se jedná o jazyk velmi blízky mým představám, až ná některé drobné detaily, jako například to, že ukazatel je standardně nenulovatelný. Hlavní výtku mám k syntaxi, která nepřijde dostatečně explicitní a využívá poměrně mnoho abstraktních symbolů.

Zdrojový kód 4.3 Ukázka syntaxe – parsing celých čísel[28] *Kód v jazyce Zig*

```
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);
    }
}
```

4.2.4 Odin

Podobně jako Zig se jedná o moderní analogii C. Procedurální jazyk s jednoduchou a minimalistickou syntaxí. Nabízí několik zajímavých vlastností, jako například kontext (viz 4.13.1) a vestavěné aritmetické operace s poli, a k tomu i maticový typ, který umožňuje například násobení matic, matic s poli a podobně.

Jako nedostatek bych vytkl absenci explicitní možnosti spouštění kódu v době kompilace, která je fakticky možná pouze při definici konstant. Syntakticky jazyk je relativně implicitní a podobá se jazyku Go. Syntaxe mi přijde více intuitivní než v případě Zigu.

Zdrojový kód 4.4 Ukázka syntaxe – programování s poli[29] *Kód v jazyce Odin*

```
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))
}
```

4.3 Vstupní bod programu

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

Pokud by například uživatel chtěl začít kód v lokálním rozsahu platnosti (scope), což je vlastnost, která se mu líbí na funkci main, měl by mít možnost to udělat přímo pomocí odpovídajícího syntaktického konstruktu.

Samotný koncept mi přijde obskurním z několika důvodů:

- Funkce main zbytečně zvyšuje úroveň odsazení kódu a komplikuje strukturu programu bez možnosti se tomu vyhnout. Pokud by uživatel chtěl logicky začít kód v lokálním rozsahu platnosti, měl by mít možnost to udělat přímo pomocí odpovídajícího syntaktického konstruktu, čímž by navíc jasně signalizoval svůj záměr.
- Zároveň povinná existence main funkce ruší intuitivní chápání pořadí vykonání instrukcí. Instrukce se běžně mohou objevit i v globálním prostoru (mimo definici funkce), ale protože program začíná v main funkci, tak není na zcela zřejmé jak je vztažené vykonaní globalních instrukcí k popisu toku v main funkce.

Jako vstupní bod programu jsem tedy zvolil počátek souboru, obdobně jako v Lua, nebo, když mám vybírat z C-like jazyku, jak v HolyC. Přijde mi to intuitivnější a ponechávající větší svobodu programátorovi. Za pomoci standardní knihovny pak budou jak ziskatelné vstupní argumenty programu, tak i možnost ukončení programu se specifickým vystupním kodem.

4.4 Alokace

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

Syntaxi jsem zvolil následující:

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

I když se jedná o specialní system, jde je o uroveň syntaxe. Obecně jazyk není typově bezpečny a přikaz alloc jen vratí ukazatel na alokovanou paměť, která obecně nemusí byt asociovana s datovým typem. Navíc alloc nemusí vždy uspět a teoreticky muže vrátit hodnotu null:

```
int^ ptr = alloc 8;
if ptr == null : { // Správa chyby }
```

Při alokaci během deklarace lze vynechat datový typ na pravé straně, pokud má být shodný s typem na levé straně. Formálně je to možné, protože `alloc` je alternativní pravá strana, oproti výrazu `new` v C++ či D, kteý by se měl řídit pravidly výrazu.

Následující řadky tedy vyjadřují ekvivalentní definice.

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

Navíc bych umožnil přímou inicializaci pomocí symbolu `:`, který se v jazyce používá jako počátek příkazu. Následující kód alokuje velikost odpovídající typu `int` a inicializuje hodnotu na příslušné adrese na `1`.

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

Pro uvolnění alokované paměti příkazem `alloc` slouží příkaz `free`:

```
free ptr;
```

Příkaz `free` musí obdržet buď ještě neuvolněný ukazatel získany příkazem `alloc` nebo hodnotu `null` (či ekvivalentní nulový ukazatel). Příkaz je neutrální vůči nulovému ukazateli. Předání jakékoli jiné neplatné hodnoty ukazatele však vede k nedefinovanému chování. Toto chování je zvoleno záměrně s prioritou na výkon, aby implementace alokátor obecně nemusela provádět validaci ukazatele.

Rozhraní pro vlastní alokátory je zatím nespecifikováno, jelikož se prvně musí přijít s definitivním řešením ohledně kontextu (viz 4.13.1), se kterým vlastní alokátory úzce souvisí.

4.5 Komentáře

V jazyce C, a například i v C++ a D, nelze vnořovat blokové komentáře `/**/`. Nejedná se o význačný nedostatek, ale stalé nepřijemný, pokud se komentuje něco, co už obsahuje komentář. Navíc je to nekonzistentní s řádkovými komentáři, které vnořovat lze.

Pro blokové komentáře jsem zvolil následující syntaxi, řádkové komentáře jsem ponechál jak jsou v C.

```
/{
    /{ komentář /}
/}
```

Důvodem volby syntaxe `/{ ... /}` byla snaha o vizuální konzistentnost napříč jazykem. Symboly `{}` obecně vymezují blok kódu. Symbol `/` přitom koresponduje se symbolem pro jednořádkový komentář (`//`). Kombinace tedy intuitivně signalizuje „blokový komentář“.

Odin například také umožňuje vnořené blokové komentáře, využívající syntaxi z C. Zig na druhou stranu nedovoluje žádné vnořené komentáře, aby byla nezávislá možnost převodu každého řádku na tokeny[30].

4.6 Pole

Jedná se o jednu z nejzákladnějších a nejužitečnějších datových struktur vyskytujících se v programování. V C se ovšem pole dají používat efektivně jen ve statických případech, kdy jejich velikost lze stanovit již při kompilaci. Avšak i v případech, kde je to dostačující, nastává problém při použití pole jako argumentu funkce. Funkce se buď musí definovat pro konkrétní velikost pole, anebo obecně pro ukazatel.

První případ je použitelný jen zřídka, jelikož nepřináší abstrakci, která se intuitivně pojí s vybraným datovým typem. Musela by se vytvořit samostatná funkce pro různé velikosti polí. Předání pole přes ukazatel to řeší, ale vyžaduje předání velikosti pole jako doplňujícího parametru, nebo ukončení pole nějakou specifickou hodnotou.

V takovémto případě se však ztrácí vyhoda vybraného datového typu, a vlastně i konceptualní smysl onoho. Kód je ve výsledku méně explicitní a navíc náchylnější k chybám, jelikož informace známá v době překladu, která se pojí pouze k jedné proměnné, se rozvádí do dvou hodnot známých za běhu programu.

Definuji tedy následující základní požadavky pro pole. Mělo by být využitelné ve funkcích bez ztráty identity a přitom být implicitním ukazatelem na počátek svých dat pří přiřazeni do ukazatele. Navíc, rozšířit typ i na dynamické[1] pole konstantní délky a dynamické pole variabilní délky.

4.6.1 Délka pole

K získání délky pole navrhuji následující syntaxi:

```
int[8] arr;
arr.length; // Vrátí délku pole, tedy 8
```

Alternativní přístupy jako funkce `len()` v Go nebo i výraz **`sizeof`** v C mohou být zavádějící. Jejich syntaxe připomíná běžná volání funkcí, ale sémantika je často speciální a funkční zápis může nesprávně implikovat běhový výpočet délky. Syntaxe `arr.length` je naproti tomu srozumitelnější, neboť přirozeně naznačuje přístup k již spočtené hodnotě.

Při předání pole do funkce by se tedy interně předával ukazatel na data a jako skrytý parametr i velikost pole. Pro případy, kdy není potřeba předávat velikost, by se použila implicitní konverze pole na ukazatel.

Protože jazyk integruje i array list, tak je třeba zavést i syntaxi pro získání velikosti alokované paměti. Navrhuji pro tento účel použít standardní název `capacity` (kapacita):

[1] Dynamickým polem se v následujícím textu bude mínit libovolné pole alokované za běhu programu. Klasické dynamické pole s proměnlivou délkou se bude přímo specifikovat jako „array list“

```
int[auton] arr;
arr.length;     // Aktuální počet prvků (0)
arr.capacity;   // Celková alokovaná kapacita (>0)
```

Protože u ostatních typů polí budou tyto veličiny vždy shodné, tak zavední atributu kapacity se nijak nepromitné na interní implementaci. V případě array listu se jedna o nezbytný parametr při implementaci, a proto jeho poskytnutí není problém.

4.6.2 Typy polí

Kromě klasického rozdělení polí na statická a dynamická, navrhuji umožnit dělení v závislosti na variabilitě jejich délky. To by umožnilo vytvářet funkcím více specifická rozhraní pro práci s poli. Například funkce, která nepotřebuje měnit velikost vstupního pole, by mohla přijmout jak statické pole, tak i dynamické pole.

Dále navrhuji integrovat koncept array listu přímo do jazyka v rámci typového systému polí, jelikož se jedná o často využívanou datovou strukturu.

Pole konstantní délky známe za doby překladu

Jedná se o pole analogické standardnímu poli v C. Velikost musí být známá v době překladu.

```
int[8] arr; // Vytvoří pole 8 prvků typu int
```

Délka existuje jen za doby překladu, pro její uchování se za běhu negeneruje žádná dodatečná proměnná.

Pole konstantní délky známe za běhu programu

Toto pole je alokováno za běhu programu, ale jeho velikost je po alokaci neměnná. Jedna se o analogií alokaci konstantního ukazatele v C, ktery by byl využivan jako pole.

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

Navrhovaná syntaxe:

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

Překladač vygeneruje kód pro uložení délky `n` spolu s ukazatelem na data. Pole by nešlo realokovat, protože délka je obecně známa až v době alokace.

Pole variabilní délky známe za běhu programu

Toto pole je alokováno za běhu programu a jeho velikost lze měnit manualně novou alokaci. Jedna se o analogií běžného použití ukazatele na dynamicky alokovanou paměť v C, kde programátor manuálně spravuje velikost.

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

Navrhovaná syntaxe:

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

Překladač opět zajistí uložení aktuální délky.

Array list

Jedná se o dynamické pole se specifickým chováním: jakýkoliv index je považován za platný pro zápis; pokud je index mimo aktuální meze pole, pole se automaticky rozšíří tak, aby zápis byl možný. Pole neposkytuje explicitní operace jako append, rozhrání (přístup k prvkum, získaní velikosti atd.) je konzistentní napříč všemi poli.

Případné rozšíření pole, způsobené zápisem na index mimo aktuální rozsah, neprovede žádné dodatečné inicializace (například nulování). Provede se pouze nezbytná alokace paměti a zápis nové hodnoty na cílový index. Vysledkem čtení předem nedefinovaných hodnot tedy je nespecifikovaná hodnota.

Potenciální implicitní alokace při zápisu odlišují tento typ od ostatních dynamických typů polí (`int[const]`, `int[dynamic]`), které vyžadují explicitní alokaci pomocí `alloc`. Aby bylo toto speciální chování zřejmé již z deklarace a aby se předešlo možné záměně s polem bez specifikované délky, je vhodné použít specifický kvalifikátor.

Navrhuji pro tento typ pole kvalifikátor `auton` (od autonomous), který explicitně signalizuje jeho autonomní chování při rozšiřování:

```
int[auton] arr;
// Nebo se specifickou počateční delkou
int[auton] arr = alloc int[8];
```

Tato varianta zvyšuje čitelnost a explicitnost kódu — programátor si musí speciální chování array listu explicitně zvolit kvalifikátorem. I když by se mohla nabízet syntaxe s prázdnými závorkami (`int[]`) jako symbol nejobecnějšího pole, byla by právě kvůli skryté alokaci potenciálně zavádějící.

Tato syntaxe také elegantně řeší, sice jednoznačny, ale na první pohled zavadějicí, sytl zapisu přiřazení literálů, kde se délka statického pole pomijí na straně levé.

```
// Vytvoří array list
int[auton] arr_list = [ 1, 2, 3 ];
// Vytvoří statické pole (typ odvozen z literálu)
int[] arr_static = [ 1, 2, 3 ];
```

4.6.3 Práce s polem

4.6.4 Práce s polem

Protože cílem je vytyčit jasnou identitu pole oproti ukazateli, navrhuji zdůraznit tento rozdíl i v sémantice operací. Pole reprezentuje souvislý blok paměti obsahující, na rozdíl od ukazatele, sekvenci prvků stejného typu. Navrhuji tedy pole vnímat primárně jako hodnotový typ (value type), kde operace aplikované na

samotnou proměnnou pole (bez indexace) se týkají všech prvků pole (element-wise).

Tím se docíli vyrazného rozdilu od C, kde pole, ač formálně odlišné od ukazatele, často podléhá nejednoznačné interpretaci. Například při deklaraci `const int arr[8]` se kvalifikátor `const` aplikuje na jednotlivé prvky, ale aritmetické operace jako `arr + 1` s polem zacházejí jako s ukazatelem na první prvek. V důsledku podobné nekonzistence není v C povoleno ani přímé přiřazení mezi dvěma poli stejného typu a velikosti.

Jasné rozdělení mezi ukazatelem a polem jako hodnotovým typem v mém návrhu umožňuje eliminovat tyto nejednoznačnosti. Stanovuji tedy následující pravidla:

- Kvalifikátory aplikované na typ pole se vždy vztahují na jeho prvky.
- Přiřazení pole do jiného pole je definováno jako kopírování jednotlivých prvků. Pole musí mít kompatibilní typy prvků a stejnou velikost (u dynamických poli není možno kontrolovat délku za kompilace, a tak odpovědnost je na programatorovi).
- Binární aritmetické a logické operace mezi dvěma poli stejné velikosti nebo mezi polem a skalárem jsou definovány jako operace prováděné po prvcích.
- Výjimkou je explicitní přiřazení pole do ukazatele (jedná se o implicitní konverzi na ukazatel na první prvek) a operace zřetězení polí (viz 4.7.2), která má odlišnou sémantiku.

Tento přístup umožňuje přirozené vektorové operace: **Zdrojový kód 4.5** Vektorové operace *Kód v navrhovaném jazyce*

```
/ Inicializace všech prvků na 0
int[8] arr1 = 0;

// Sečtení dvou polí (po prvcích)
int[3] a = [ 1, 2, 3 ];
int[3] b = [ 10, 20, 30 ];
int[3] sum = a + b; // [ 11, 22, 33 ]

// Operace pole a skaláru
int[3] scaled = a * 10; // [ 10, 20, 30 ]
```

Další operace tykající se poli budou zmíněne v kontextě textových řetězcu viz 4.7.2

4.6.5 Příklady jiných jazyků

C++

Ponecháva vestavené statické pole jak v C, vlastní řešení nabizí pomoci kontejnerů standardní knihovny (std::array a std::vector). Neobsahuje však možnost správy dynamického pole konstantí délky. Nabizí pouze `std::span` – ne-vlastní

pohled na souvislou sekvenci na prvků[31]. Nelze ho tedy použít k uložení dat, ale muže byt použít pro tvorbu odpovidajicího rozhrání funkce.

```
std::array<int, 8> arr_static;    // Statické pole
std::vector<int> arr_list;        // Array list
std::span<int> view = arr_static; // Pohled na data pole

// Délka získatelná .size()
arr_static.size();
arr_list.size();
view.size();

// U std::vector lze získat skutečnou alokovanou kapacitu
arr_list.capacity() // V bajtech
```

D

Obohacuje jak statická tak i dynamická pole s získatelnou délkou. Dynamická pole v D mužou byt spravována jak garbage collectorem, tak i manuálně. Navíc mohou měnit velikost.

```
int[8] arr;             // Statické pole
int[] arr = new int[8]; // Array list

// V obou připadech délka ziskatelná jako
arr.length;
```

Odin

Má statická pole, dynamicky alokovatelné `slices` (arr := make([]int, N)), které jsou pohledem na paměť (na rozdíl od std::span se muže provest jejich přima inicializace nově alokovaným blokem paměti), a tzv. dynamická pole, která fungují jako array listy. Pro všechny je dostupná funkce `len()` vracející jejich délku. Pro zjištění alokované kapacity se dá použit funkce `cap()` (nepoužitelný v připadě pohledu).

```
arr: [5]int             // Statické pole
arr := make([]int, 8)   // Pohled přímo inicializovany anonymně
arr: [dynamic]int       // Array list

// Ve všech připadech lze použít
len(arr);
// Ve připadě statického pole a array listu
cap(arr);
```

4.7 Řetězce

V jazyce C jsou řetězcové literály pouze více konvenční variantou zapsaní pole konstantních znaků. Tento přístup je ve své podstatě primitivní, avšak zcela postačující. Problémem je zde absence identity pole jako datového typu, jak již bylo zmíněno viz 4.6. V důsledku toho se s každým řetězcem pracuje jako s ukazatelem.

Jelikož já definuji pole odlišně, lze jejich možnosti rozvinout tak, aby ve výsledku umožňovaly lehčí práci i s řetězci. Samotný datový typ pro řetězec nebude existovat, bude jen vestavěná podpora řetězcových literálů, které se při kompilaci transformují na pole.

4.7.1 UTF-8

Bylo by vhodné rozšířit podporu literálů z ASCII na jiné kódován, které by umožnílo jednoduchou manipulaci se složitějšími symboly. Jako takové kódování jsem zvolil UTF-8, neboť je zpětně kompatibilní s ASCII, jeho základním blokem je bajt, tudíž není závislé na endianitě, a je velmi rozšířené.

Protože symboly v UTF-8 mají variabilní délku, navrhuji vyhodnocení největší délky potřebné pro uložení jednoho symbolu daného literálu v době kompilace a následnou konverzi na pole integrálních hodnot onoho patřičného typu. Každý element výsledného pole pak bude reprezentovat jeden samostatný symbol, interně kódovaný v UTF-8. Tento přístup umožní pracovat se symboly samostatně, využívat všechny výhody polí a pro texty obsahující pouze ASCII znaky mít stejně velké pole jako v C.

```
u16 str = "čau";
str[0]; \\ 'č'
str[1]; \\ 'a'
str[2]; \\ 'u'
```

Levá strana přiřazení může definovat libovolnou velikost v ramci integralních typu pro uložení symbolu:

```
u32[] str = "čau";
```

Rozdíl velkostí datových typů prvků levé a pravé strany je řešen standardně jako rozdíl velikosti libovolných prvků (dojde ke konverzi se ztrátou informace).

Aby bylo možné pracovat s řetězcovými literály jako s prostým blokem paměti, navrhuji definovat tzv. surový (raw) řetězcový literál, označený postfixem `R`:

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

Jedna se o řetězcový literal, kde každy bajt řetězce zapsaného v UTF-8 se vníma jako znak. Tedy fakticky jde o statické pole typu `u8`.

Protože vznikla potřeba vyjadřovat i jednotlivé znaky, zavedl jsem UTF-8 podporu i pro znakové literály:

```
u8 ch8 = 'a'
u16 ch16 = 'č';
u32 ch32 = 'は';
```

Navíc navrhuji povolit definici víceznakových literálu:

```
u32 mch = 'ABCD';
```

První znak vždy odpovida nejvyššymu bajtu. Tedy by první bajt hodnoty `mch` měl kód znaku `'A'`, druhý bajt kód `'B'`, třetí `'C'` a čtvrtý `'D'`.

Příklady jiných jazyků

V jazyce D jsou řetězcové literály standardně ve formátu UTF-8 jako neměnné (immutable) pole znaků. Pomocí postfixu (`w`, `d`) je lze interpretovat jako pole `wchar` (UTF-16) nebo `dchar` (UTF-32). Nabízí také WYSIWYG (what you see is what you get) řetězce.

```
"hello"w; // wchar
"hello"d; // dchar
r"ab\n";  // Wysiwyg (obsahuje 'a', 'b', '\', 'n')
```

V jazyce Odin jsou řetězce také ve formátu UTF-8. Obsahuje koncept tzv. `rune` pro práci s Unicode kódovými body. Nabizí take datový typ `cstring` pro kompatibilitu s C.

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

V jazyce Zig jsou řetězcové literály reprezentovány jako nulou ukončená pole bajtů, které lze v kodě zapisovát pomoci UTF-8 symbolu. Ale samotný datový typ neposkytuje přímou podporu pro práci s Unicode znaky. Rozšířená práce s řetězcy je dostupná prostřednictím standardní knihovny.

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

4.7.2 Operace

Jako jediné konvenční operace nad řetězcy, ktere považuji za vhodné integrovat do syntaxe, jsou zřetězení a vyběr podřetězce. Ostatní operace by už měly byt obsažené ve standardní knihovně.

Zřetězení

Níc nového bych nevymyšlel, a použil operator `..` jako třeba v Lua. Podstatným je, že operator je odlišen od operatoru aditivity, který se v některých jazykach používa (Go, Java), jelikož to mi přijde zavadějicí.

```
u8[] str1 = "Hello";
u8[] str1 = "World";
// Operace .. sama o sobě nealokuje, jen popisuje výsledné pole
u8[] str3 = str1 .. "␣" .. str2; // str3 je pole u8 s délkou 11
```

Jelikož délka libovolného pole je v době kompilace (pro statická pole) nebo za běhu (pro dynamická pole) zjistitelná, lze tuto operaci zobecnit na jakýkoliv typ pole, za předpokladu, že datové typy prvků jsou kompatibilní. Je však důležité zdůraznit, že samotný operátor `..` by neměl provádět žádnou implicitní alokaci

paměti na haldě, bylo by to zavadějicí. Připadné vysledné pole by se muselo explicitně alokovat standardnimi prostředky jazyka. Pro dynamická pole by taková alokace mohla vypadat například následovně:

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

Výběr podřetězce

S konceptem vyběru podřetězců (výřezem) se lze setkat v různých jazycích v různých podobách. Obvykle se jedná o reprezentaci libovolné souvislé části pole, která sama o sobě nealokuje novou paměť pro data, ale odkazuje data v původní pole. Například v C by se výřez (slice) dal reprezentovat třeba nasledujicím způsobem:

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

kde `dataPtr` by ukazoval na nějaký element v puvodním poli, a `len` by specifikoval délku. Odin a Zig, napřiklad, implementují vyřezy pravě tímto způsobem – jako strukturu obsahující ukazatel na data a délku. Odin, Zig a D navíc vnímají vyřezy jako svébytné datové typy a využivají je například jako rozhraní pro práci s dynamickými poli nebo řetězci.

Já však nevnímám výřez jako samostatný datový typ, ale spíše jako operaci nad polem, jejímž výsledkem je opět hodnota typu pole. Vyřezy je tedy datového typu pole. Pro operaci vyběru navrhuji nasledujicí syntaxi:

```
u8[] arr = [ '0', '1', '2', '3', '4' ];
// Výběr prvků od indexu 1 (včetně) do indexu 3 (včetně)
u8[3] tmp = arr[1 : 3]; // [ '1', '2', '3' ]
// Výřezy lze použít i na levé straně přiřazení
tmp[1 : 2] = arr[3 : 4]; // [ '1', '3', '4' ]
```

Navíc, navrhují nasledujicí syntaxi pro vyběr podřetězce od počátku do specifického indexu a od specifického indexu do konce:

```
u8[] arr = [ 1, 2, 3, 4, 5 ];
// Vybere všechny prvky od arrA[1] do konce
arr[1:]; // [ 2, 3, 4, 5 ]
// Vybere všechny prvky od počatku do arrA[2]
arr[:2]; // [ 1, 2, 3 ]
// Vybere cely řetězec
arr[:]
```

A take syntaxi připojení na konec řetězce (fakticky použitelné jen v ramci array listu):

```
u8[auton] arr;
arr[] = [ 1, 2, 3, 4, 5 ];
```

4.8 Jmenné prostory

Jmenný prostor představuje jednoduchý ale zručný nástroj pro organizací kodu. Umožňuje sdružovat související deklarace pod jedním společným názvem, který je rozlišitelný překladačem. Na rozdíl od manuálního používání prefixů nebo postfixů v názvech identifikátorů je z hlediska nástrojů pracujících s kódem (např. LSP) strukturním celkem.

Jmenné prostory rovněž umožňují při kompilaci hromadně pracovat s obsaženými deklaracemi, čehož se dá efektivně využívat i pro import a export částí kódů. Příkladem může být mechanismus importů v Pythonu:

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

To, mimo jiné, umožňuje řešit potenciální kolize názvů, které mohou nastat při importování knihoven. Pokud by importované jmenné prostory vždy nesly pouze svůj původní název (jak je tomu v C++), riziko kolizí by se síce snížilo, ale samotná jmena jmenných prostoru by stále mohly byt přičinou. Proto je vhodné umožnit přejmenování importovaného jmenného prostoru v importujícím kódu. Například opět jako v Pythonu:

```
import foo as boo;
```

Jmenné prostory bych vnímál jednoduše jako pojmenované rozsahy platnosti. Zvolil jsem syntaxi c C++, protože vcelku jasně a jednoduše vystíhuje myšlenku:

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

I v připadě přístup k prvkům uvnitř jmenných prostorů:

```
Foo::x;
```

4.9 Systém importu

Systém importu založený na hlavičkových souborech považuji za jednu z nejproblematičtějších částí jazyka C. Jejich hlavní nevýhodou je duplicita definic. Slouží však k dobrému úmyslů – izolaci implementace a definici rozhraní. Cílem tohoto návrhu tetdy je zachovát tuto myšlenky, ale vyhnout se jak použití preprocesoru, tak i duplicitě kódu.

Základní jednotkou samotné kompilace a importu bude soubor, jelikož se jedná o to co se ve výsledku předává překladači. Překladač dostane jen jeden vstupní soubor, který následně pomocí prostředků jazyka může umožní načíst obsah dalších souborů. Veškeré importy budou probíhat v rámci AST (na rozdil od textového vkladaní v C). Každý soubor by tedy měl představovat samostatně syntakticky

analyzovatelný celek. Systému importu nepovoluje cyklické závislosti mezi soubory a cesty k importovaným souborům jsou interpretovány relativně vůči umístění importujícího souboru.

Intuitivně se nabízí možnost přímého importu souboru následující syntaxí:

```
import filename;
```

Této možností bych se však vzdal. Domnívám se, že by jen vybízela k „nesprávnému“ přístupu, který by zvyšoval riziko kolizí názvů -– což bylo podrobněji rozebráno v sekci 4.8 -– a misů. Omlouvám se, pokud má předchozí odpověď vyzněla, že tyto body chybí – byly tam, jen jsem je možná při předchozím čtení plně nedocenil v kontextu naší diskuse.

Váš systém správy chyb a jeho zdůvodnění se zdají být v textu adekvátně popsány.nepřinášela nic, co by nešlo řešit jinak.

Zavedl jsem tedy variantu, která vždy zajišťuje určité zabalení importovaného souboru ze strany importujícího kódu:

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

Tímto přikazem by se vytvořil nový jmenný prostor `Foo`, kam by se nasledně překopírovál kořenový uzel rozparsovaného souboru `filename`.

Syntaktický se specialně specifikuje klíčové slovo `namespace` umožňujíc využití daného konstruktu k implementaci i jiných zpusobu zabalení souboru. Například:

```
import filename as scope;   // zabalení do bloku platnosti
import filename as fcn foo; // zabalení do funcke foo
```

Zabalení do bloku platnosti může být použito, pokud se má kód z importovaného souboru vykonat jednou při zpracování importu, aniž by jeho symboly zaplnily aktuální jmenný prostor. Kód zapouzdřený do funkce se naopak může spustit explicitním zavoláním dané funkce (`foo()`) libovolný početkrát a v libovolném místě kódu. Nejedná se o potenciálně často používaný konstrukt, ale umožňuje do budoucna beze změny syntaxe řidávat další specifické způsoby importu nebo zpracování souborů.

Dále navrhuji tento konstrukt rozšířit rozšířit a zavest import jen vybraných symbolu ze souboru.

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

Patříčná syntaxe umožní importovát identifikatory foo a boo ze souboru filename a zabalít je do nového jmenného prostoru Foo.

V zásadě tento přístup umožňuje robustní import a další prostředky nejsou nezbytně nutné. Zbývá zohlednit viditelnost importovaných identifikátorů.

Lze vycházet buď z toho, že vše je ve výchozím stavu viditelńe, a viditelnost se omezuje, nebo naopak – vše je ve výchozím stavu nepřístupné a přístup se rozšiřuje. Druhý přístup je víc prakticky, ale je méně intuitivní, protože, jelikož se intuitivně očekává, že při importu souboru se alespoň nějký obsah bude dsotupný.

Podstatnější je však otázka viditelnosti vnořených importů. Tedy, importuje-li soubor identifikátory z jiného souboru, budou-li viditelné při importu taky. Zřejmé je, že pokud jsou přístupné při jednom importu, tak by měly být přístupně i pro další importy, jelikož jsou ve vysledku na stejné úrovní jako kód souboru a nekladlá se žádná omezení.

Navrhují proto umožnit omezení viditelnosti na úrovni importu, než na úrovni jednotlivých identifikátorů. Poskytovalo by to explicitní kontrolu nad viditelnosti symbolu, aniž by se to muselo řešit poprvkově. Navíc by stale byla možnost vytvoření připadného rozhraní z dostupných symbolu. Symboly sloužicí jako veřejné rozhraní modulu by se umistili do jednoho souboru a zbyle by se importovali lokalně.

K označení těchto lokálních importů navrhuji použít klíčové slovo `local`:

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

Samotná klasifikace proměnných dle viditelnosti by se mohlá připadně řešit v budoucnu již za pomoci direktiv překladače. Napřiklad:

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

4.10 Přetěžování funkcí

Přestože se jedna o implicitní mechanismus, který může od čtenáře skrývat identitu konkrétní volané funkce, tak přináší z mého hlediska jednu zásadní výhodu – zjednošuená jména funkcí. Napřiklad namísto nutnosti vypisovat datové typy do názvu funkce (např. `max_int`, `max_float`) pro její rozlišení lze uvést název vystihující pouze její činnost (např. |max|).

To usnadňuje vnímání samotného programu, jelikož při práci s komplexními uživatelsky definovanými datovými typy, názvy funkcí budou už znatelnou zatíží. Navíc jména samotných funkcí s použitím identifikujicích prefixu a sufixu nejsou vnímány jako atomické celky nástroju pracujicich s kodem. I když by to šlo částečně řešit za pomoci jmenných prostorů, tak by to jen vedlo by to k dalšímu prodlužování zapisu volání (např. `Math::Int::max` zamisto `Math::max`).

Samotná abstrakce nad konkrétní volanou funkcí není pro čtenáře kódu nikterak zavádějící. Nebo spíše, je zavádějící stejně jako smyčka **`for`**, která abstrahuje instrukce skoků. Smysl čtenář získává ze samotného názvu funkce a typů vstupních argumentů, přičemž konkrétní funkce je pro něj často jako „černá skříňka“. I když funkce přebírá například **`int`**, volající nemůže vědět, zda onen **`int`** neni

hned první instrukcí přetypován na `float`. Jedině to tedy může ovlivnit čas potřebný k nalezení definice konkrétní přetížené varianty funkce (bez užití LSP), což však nepovažuji za závažný problém.

Z mého hlediska je tedy lepší možnost přetěžování v jazyce mít, než nemít. Jelikož jazyk podporuje implicitní konverze základních datových typů obdobně jako v C, tak klíčovou je volba typu přetěžování. Muže se jednat buď o implicitní přezežování, kde při hledaní vhodné se bude zohledňovát i imlicitní přetypování argumentu, například:

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

Nebo explicitní přetěžování, kde datové typy argumentu musí striktně sedět:

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

Při explicitním přetěžování datové typy argumentu přímo identifikují volanou funkci. Ovšem zda existuje potřebná varianta, se lze při psaní kódu dozvědět jen z LSP. V takovém to případě je explicitní přetypování z hlediska informace totožné s implicitním přetěžováním. Faktické využití explicitního přetěžování se pak svadí k vymezení argumenty konkrétní varianty funkce s očekavanou chybou překladu při její absenci. Pokud by některy z argumentu musel byt explicitně přetypován, tak dojde k zdelšení zapisu volání (pře se s cílem) a duplikaci informace. Navrhuji proto použít implicitní přetěžování jako výchozí chování, ale zároveň poskytnout opci vyžadání přesné shody datových typů konkrétního volání. Zvolil jsem nasledujicí symboliku – přidání `!` za název funkce při volání:

```
foo!(arg1, arg2);
```

Využití prapodivného symbolu v tomto případě není zavadějicí, jelikož očekavané intuitivní chovaní vyrazu se nemění. jedná se stále o volání funkce, které nijak nemění očekávané výsledky ani vstupy z hlediska čtenaře, je prakticky irelavantní.

4.10.1 Definice shod

Pro rozlišení volané funkce definuji následující úrovně shody datových typů argumentů, od nejvyšší priority po nejnižší:

Přesná shoda Typy argumentu volání a parametru funkce se přesně shodují bez jakékoli konverze. (Např. `int` → `int`, `double` → `double`).

Rozšiřující konverze Implicitní bezztrátové konverze (Např. `int` → `long long`, `short` → `int`, `uint` → `ulong`).

Konverze na typ s plovoucí řádovou čárkou Implicitní konverze z celočíselného typu na reálný typ. Může dojít ke ztrátě přesnosti u velkých celých čísel. (Např. `int` → `double`, `long` → `float`).

Konverze znaménkovosti Implicitní konverze mezi znaménkovými a neznaménkovými celočíselnými typy stejné velikosti. Může dojít ke změně interpretované hodnoty. (Např. `int32` → `uint32`, `uint32` → `int32`).

Zužující / Ztrátová konverze Implicitní konverze, která **potenciálně** může vést ke ztrátě informace. (Např. `double` → `int`, `long` → `short`, `double` → `float`).

Ostatní konverze Libovolné ostatní povolené implicitní konverze (např. konverze ukazatelů).

Překladač pak bude moci vybrat správnou funkci v závislosti na nejlepší shodě všech argumentů (dle nejvyšší dosažené priority). Jestliže pro alespoň jeden argument není možná ani nejnižší povolená konverze, dojde k chybě překladu. Pokud je nalezeno více funkcí se stejnou nejlepší shodou, volání je považováno za nejednoznačné a výsledkem je chyba při překladu.

4.10.2 Přístup jiných jazyků

Odin obsahuje pouze explicitní přetěžování, jelikož jazyk umožňuje definovat vnořené funkce ve funkcích, a tudíž rozlišení konkretní funkce, ktera se ma zavolat, není trivialní[32].

Zig nemá přetěžování funkcí[30], ale podobného chování (jedna funkce pracující s více typy) lze docilit při kompilaci za pomoci tzv. „duck typing“ a metaprogramování.

Zdrojový kód 4.6 „duck typing“ alternativa přetěžování *Kód v jazyce Zig*

```
fn add(comptime T: type, a: T, b: T) T {
    return a + b;
}

const result = add(i32, 1, 2);
const resultFloat = add(f32, 1.0, 2.0);
```

D a C++ Mají implicitní přetěžování funkcí[33, 34].

4.11 Správa chyb

4.11.1 Implementace

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

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

To vše mě ve výsledku vede k myšlence o přidání standardního systému pro správu chyb do jazyka.

Z metod řešení standardizované správy chyb v jiných programovacích jazycích lze v zásadě vyčlenit dva přístupy:

Návratová hodnota Chyba je vracena jako návratová proměnná nebo její součást. Obvykle je to spojeno s možností návratu několika hodnot, kde se vyčleňuje jedna pozice (např. poslední) pro případnou chybu (Odin), nebo je přímo speciální doplňující návratová hodnota vyhrazena jen pro chybu (Go). Také se může vracet struktura obsahující jak případnou chybu, tak i běžnou návratovou hodnotu (Rust). Tento přístup je přímočarý a explicitní a dává svobodu programátorovi, jak a kde s chybou naložit. Zpracování chyby je pak přímou součástí toku programu. Chybový stav je tedy prakticky jen dalším stavem programu.

Try-Catch Využívá se systém tzv. výjimek, kde případné chybové místo kódu je zabaleno do `try` bloku a případná chyba je odchycena do `catch` bloku. To umožňuje například nezatěžovat hlavní logiku kódu správou chyb a psát kód `try` bloku tak, jako kdyby žádná chyba nastat nemohla. Odchycená chyba se následně zpracuje 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, jenž onu funkci volal.

4.11.2 Definice požadavků

Neprve bych si definoval požadavky chybového systému:

- Jednotný datovy typ – měl by existovat jednotný způsob reprezentace chyby.
- Chyby by mělo být možné seskupovát do skupin (množin), které by se mohly kompozičně skládat. Například existují-li samostatné skupiny chyb pro čtení a záspis do souboru, mělo by být možné je spojit do společné skupiny reprezentující obecné souborové operace.
- Definice funkce by měla explicitně specifikovat množinu chyb, které mohou byt při jejím volání vráceny.
- Umožnit jednoduchou propagací chyby zasobníkem funkcí dal. Tedy zjednodušít obdobny často se vyskytujicí konstrukt:

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

4.11.3 Implementace

Z metod řešení standardizované správy chyb popsaných výše jsem pro tento jazyk zvolil cestu návratové hodnoty. Rozhodl jsem se tak, protože nahlížet na chybu pouze jako na další stav programu, i když řekněme speciální, je z mého hlediska přirozenější a tento přístup neobsahuje skrytý tok řízení (jako výjimky).

Jelikož funkce v tomto jazyce standardně má k dispozici právě jednu návratovou hodnotu, chyba se bude vracet samostatným kanálem. Nicmeně, nechtěl bych vnímat chybu přímo jako druhou navratovou hodnotu určenou jen pro chybu, jak je tomu např. v Go. Protože pak se pro každé volaní funcke musí řešit dvě vystupní proměnné. To ve vysledku povede k vytvoření buď implicitních pravidel (jako v Go při redeklaraci chyby), nebo k rozvláčné syntaxe.

A tedy obdobné řešení mi nevyhovuje. Pro bližší ilustraci, k jakým nejednoznačnostem nebo syntaktickým kompromisům může vést navrat chyby jako druhé návratové hodnoty, uvedu následující příklad v Go. Symbol `:=` vyjadřuje deklaraci s inicializací.

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

val1, err := foo();
if err != nil { /* zpracování chyby */ }

val2, err := foo();
if err != nil { /* zpracování chyby */ }
```

Intuitivně zde není zcela zřejmé, co se děje při druhém volání. Prvně se provádí definice `val1` a `err`, načež se ve stejném rozsahu platnosti provádí definice `val2` a opětovná inicializace `err`. Samozřejmě je to zohledněno pravidly jazyka, kód je kompilovatelný a nová definice `err` se neprovede (použije se existující proměnná `err` v daném rosahu platností). Ovšem dochází zde ke sporu syntaxe a sémantiky, kde ze syntaktického hlediska se `err` tváří jako běžná druhá návratová hodnota, ale ze sémantického hlediska pro ni platí speciální pravidla při použití `:=`, jen protože se jedná o chybovou hodnotu typu `error`.

Navíc se situace komplikuje přidáním kvalifikátorů. Bude-li se například chtít označit `val1` jako **`const`** ale ne `err`. To lze řešit na úkor upovídané syntaxe, , bude-li se chtít zachovat explicitnost, nebo přidáním dalších implicitních pravidel.

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

Navrhují následující syntaxi s klíčovým slovem `catch`:

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

Kde `error` by reprezentoval jednotný datový typ chyby a případná chyba vrácená funkcí `foo` by se v tomto případě uložila do proměnné `err`.

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

Kuriozně se lze v takovém to případě dopustít jedné vyjimky – pominutí samotné defince chyby před odchycením – jelikož je redundantní, místo odchytu totíž muže jen pracovát s datovým typem error. Protože by to však bylo zavadějicí, tak navrhuji, aby tato varianta byla povolena pouze v kombinaci s bezprostředně následujícím blokem platností. Chyba by v takovém to případě byla odchycená lokalně a platna jen počinaje naledujicím blokem platností. Samotný blok platnosti by se nelišil chováním od klasických. Viz ukázku:

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

Jedná se o kompromis umožňující stručnou syntaxi pro oba běžné scénáře (sdílená vs. lokální chyba).

Množiny chyb

Samotná chyba by měla byt jednoduše identifikovatelná svým jménem, aby ji bylo možné používat pro určení stavu programu. Například:

```
foo() catch err {
    if (err == ErrName) {
        // Porovnání odchycené chyby s konkrétním typem chyby
    }
}
```

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.

Zavedu tedy k reprezentací chyb tzv. množiny chyb, které budou jednoznačně rozlíšitelné svym jménem:

```
error ErrorSetA {
    ErrorA;
    ErrorB;
};
error ErrorSetB {
    ErrorSetA; // Zahrnutí ErrorSetA jako podskupiny
    ErrorB;
```

```
};
```

Zde `ErrorSetB` obsahuje vnořenou podskupinu `ErrorSetA` a prazdnou množinu `ErrorB`. Libovolná z těchto množin je identifikovatelná svým plně kvalifikovaným jmenem pomocí operátoru `::` a muže byt přiřazena do proměnné datového typu error. Hodnota `null` reprezentuje stav bez chyby. Viz:

```
error err = ErrorSetB::ErrorB;
error err = ErrorSetB::ErrorSetA::ErrorA;
error err = null;
```

Protože tyto množiny chyb mají smysl primárně při definici chybového rozhraní funkcí a definice funkce ve funkci není v jazyce umožněna, je jejich definice uvnitř těl funkcí zavádějící, a tudíž zakázana. Lze tedy tyto množiny vnímát jako nadstavbu nad jmennými prostory specificky pro chyby, a proto k rozlišení jejich prvků, jak již bylo ukázano, používat stejný operátor `::`.

K definici chybového rozhraní funkce se pak použije následující syntaxe s klíčovým slovem `using`:

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

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

Toto řešení je jednoduché a relativně všestranné. Umožňuje například rozšířit libovolnou množinu (i prazdnou) o nové prvky, aniž by se rozbil kód využívající původní množinu. Ovšem má jeden základní nedostatek – vrací se pouze stav. Tedy nelze přímo vrátit doplňující informaci o chybě. Teoreticky je to řešitelné přidáním nových stavů pro každou variantu informace, ovšem to zdaleka není praktické.

Prakticky toto omezení vadí jen při logování chyby, protože se jinak vždy popisuje stav programu, který je nezbytný z hlediska jeho činnosti. Tudíž přidani v takových to připadech nového chybového stavu je vlastně nezbytné (uvažuje-li se, že daný stav je vhodné vnímat jako chybovy, obecně to lze řešít standardní cestou).

Ve vysledku předávání doplňujicích informací o chybě slouží jen jako doplnění systemu. Jako 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 snadno standardizovatelný a tak či onak se jedná o konkrétní záležitost.

Pokud by se navržený model zobecnil například definicí chyby pomocí struktury nebo unie (aby chyba mohla nést data), vlastně by došlo k rozporu s cílem standartizace. Systém by se totiž zobecnil natolik, že by mohl být využíván i pro jiné účely a mnohými způsoby, a tudíž by se postavený problém nevyřešil, jen by se problém přesunul jinam. Možná by bylo přijatelným kompromisem povolit přiřazení konkrétních celočíselných hodnot k identifikátorům chyb v jejich defi-

nici. To by umožnilo například indexovat pole chybovými kódy (podobně jako u enumů) a mohlo by v některých případech sloužit jako velmi silný nástroj.

Návrat chyby

Možnost v chybovém stavu vrátit i normální hodnotu z funkce je užitečná záležitost. Může posloužit například jako doplňující informace k chybovému stavu. Navíc je to v jistém smyslu i nutná záležitost, jelikož vnímáme chybu jen jako další stav programu, nikoli jako něco fundamentálně odlišného.

Navrhuji následující intuitivní syntaxi pro vrácení hodnoty a chyby:

```
return value, err;
```

Kde `value` představuje proměnnou nebo výraz s návratovou hodnotou a `err` navratovou chybu.

Standardní návrat jen hodnoty tak zůstává nezměněn:

```
return value;
```

Otázkou je návrat jen chyby. Lze k tomu přistoupit tak, že takovýto případ vlastně neexistuje — spolu s chybou je vždy vráena i nějaká hodnota. To by také zaručilo, že proměnná, do které se zapíše návratová hodnota, nebude mít nikdy neurčenou hodnotu po volání funkce, která mohla selhat. I když je toto bezpečné chování, nelze ho vždy vynucovat. Má-li mit jazyk nizky uroveň abstrakce nad assemblerem, musí také dát programátorovi kontrolu. Nelze jen tak zbytečně vnucovat přiřazení nebo inicializaci návratové hodnoty, pokud není potřeba.

Proto navrhuji použít symbol _ označující explicitní zahození hodnoty:

```
return _, err;
```

Pro možnost jednoduché propagace chyby bych přidal nasledujicí syntaktický konstrukt:

```
foo() catch return;
// by bylo ekvivalentní s
foo() catch err {
    if (err == null) return _, err;
};
```

4.11.4 Přístupy jiných jazyků

Přístupy jazyků jako C++ a D, které správu chyb řeší klasicky pomocí výjimek, není v kontextu tohoto návrhu nema smysl je detailně rozebírat. Zajímavější je zaměřit se na jazyky Zig a Odin, jejichž pohled na danou problematiku nabízí relevantní srovnání.

4.11.5 Přistupy jiných jazyku

Řešít přistupy C++ a D nemá moc smysl, jelikož řeší problem klasicky za pomoci vyjimek. Zajimavější je se podivát na příklad Zigu, ktery má zajimavější pohled na věc a je dost podobny a Odin.

Zig

V jazyce Zig za reprezentaci chyb odpovídají tzv. error sety, které jsou podobné enumům. Každé chybě (identifikátoru v rámci error setu) je překladačem přiřazena unikátní integrální hodnota. Definice může vypadat následovně:

```
const FileOpenError = error{
    AccessDenied,
    OutOfMemory,
    FileNotFound,
};

const AllocationError = error{
    // Stejný název chyby může existovat ve více sadách
    OutOfMemory,
};
```

Chyb lze spolu spojovat do jedné větší množiny nasledovně:

```
// bude obsahovat chyby z FileOpenError a AllocationError
const FinalError = FileOpenError || AllocationError;
```

Specifikovat chybové rozhraní funkce lze pak následujícím způsobem:

```
fn divide(a: f32, b: f32) MathError!f32;
```

Funkce pak vrátí buď chybu typu MathError, nebo běžnou návratovou hodnotu. Specifikace konkrétní množiny chyb se může v signatuře funkce pominout, funkce pak vrací obecnou chybu (`anyerror`).

K odchycení a zpracování chyby používá Zig klíčové slovo `catch` nasledujicím zpusobem:

```
const a = divide(1.0, 0.0) catch |err| {
    // zpracování chyby
}
// Pokud chyba nenastala, 'a' obsahuje výsledek dělení

// Odchycení a zpracování pomocí switch
var a = divide(1.0, 0.0) catch |err| switch (err) {
    // zpracování chyby přímo pomoci switch
}
```

Chybu lze take snadno propagovát dal pomocí klíčového slova `try`:

```
const a = try divide(1.0, 0.0);
```

Navíc je v jazyce dostupny třeba `defer` a `errdefer`. Kdežto `defer` umožňuje vykonávát kod vždy při opuštění bloku, `errdefer` vykonává kod jen tehdy nastane-li chyba.

```
fn createFoo(param: i32) !Foo {
    const foo = try tryToAllocateFoo();
    // Provede se uvolnění foo při libovolné
    // při libovolném chybovém navratu z funkce
    errdefer deallocateFoo(foo);

    ...

    return foo;
}
```

Odin

Odin fakticky nemá jednoznačně definovanou standardizovanou správu chyb, ale obsahuje zajímavý operátor `or_return`, který může vést k určité konvenci. Aplikace operatoru na výraz způsobí okamžitý návrat z aktuální funkce, pokud poslední hodnota vrácená výrazem je `nil` nebo `false`. Protože Odin umožňuje návrat libovolného počtu hodnot, poslední návratovou hodnotu lze konvenčně využívat pro signalizaci chyby (např. vrácením nil pointeru nebo false booleanu při selhání) a využívat tuto vlastnost operátoru pro skutečnou propagací selhání.[32]

Pro reprezentaci různých typů chyb (obdobu množin chyb) lze v Odinu využít unie, která na rozdíl od C unie funguje jako algebraický datový typ (sum type / tagged union), a je tedy porovnatelná [32]. Konkretní chyba pak muže byt reprezentována strukturou a obsahovát doplňujicí informaci. Viz přiklad:

```
Error :: union {
    MathError,
    ...
}

MathError :: union {
    DivideByZeroError,
    ...
}

DivideByZeroError :: struct {
    a: f32,
    b: f32,
    str: string,
}

divide :: proc(a: f32, b: f32) -> (f32, MathError) {
    if b == 0.0 {
        return 0.0, DivideByZeroError{ a = a, b = b,
            str = "Division_by_zero"}
    }
    return a / b, nil
}
```

```
foo :: proc() -> Error {
    ansA, err := divide(1.0, 0)
    if err != nil {
        return err
    }

    ansB := divide(1.0, 2.0) or_return
    ...
}
```

4.12 Exekuce za doby překladu

V rámci optimalizací kompilátory mohou provádět výpočty některých výrazů, pokud mají dostatek informací. Tedy například:

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

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

Ovšem to vše je prováděno implicitně. Podstatné však je mít kontrolu nad exekucí za doby překladu ze strány jazyka. To umožní mít programátorovi jistotu, že vše, co se má provést za doby překladu, se skutečně provede za doby překladu, a to v nezávislosti na úrovni zvolených optimalizací, verzi překladače atd. Koncept je zpravidla obsažen v nějaké formě v jazycích určených ke kompilaci do nativního kódu, jako například C++, D, Zig, Rust atd.

K deklaraci proměnné známé za doby překladu navrhuji využít klasifikátor `embed` od slova embeded (vestavený), protože slovo odráží smysl (hodnota je „vestavěna“ do kódu) a vzniká obdobně jako `const`. Navíc má stejnou délku, což by při po sobě jdoucích deklaracích vypadalo dobře:

```
const int x; // Může být inicializována i za běhu
embed int y; // Musí být známá v době překladu
```

Implementačně by proměnná deklarovaná jako `embed` v běhovém prostředí neexistovala (nebyla by pro ni alokována paměť), ale při jejím použití by se vždy přímo využívala její v době překladu spočtená hodnota.

Takovýto jednoduchý klasifikátor pak umožní provádět velmi složité výpočty za doby překladu programu. Protože ze své podstaty `embed` specifikuje, že hodnota proměnné má být spočtena při kompilaci, musí se kompilátor pokusit o její výpočet (bez ohledu na složitost výrazu na pravé straně) a buď uspět, nebo ukončit kompilaci s chybou.

Lze se tedy například pokusit vypočíst hodnotu libovolné funkce při kompilaci, přičemž funkce se nemusí specialně předeklarovat jako „compile-time“ (jako například v C++ pomocí constexpr):

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

Je zřejmé, že provést libovolný výpočet za doby překladu není zcela možné, zejména má-li jazyk umožňovat přímý přístup k paměti. Kromě rozdílů mezi paměťovým prostorem reálně běžící aplikace a prostředím kompilátoru, které zajišťuje výpočty v době překladu, mohou být některé symboly (obzvláště funkce, např. z dynamických knihoven) definovány až za běhu.

Přesná omezení `embed` proto porostou postupně s vývojem překladače. S každou verzí bude možné povolovat více jazykových konstruktů a funkcí standardní knihovny, čímž se bude překladač blížit co nejvěrnější emulaci možností běhového prostředí. Nejprve tedy se muže začit z podpory běžných aritmetických výrazu, dale volánní funkcí a obecnému vykonání toku programu bez řešeí ukazatelu, dále přidaní podpory definováných ukazatelu atd.

Hlavní vyhodou je konvence použití a zvýšená míra znovupoužitelnosti kódu. Předem se nemusí složitě zjišťovat zda je daný výpočet v době překladu možný — jednoduše se deklaruje proměnna jako `embed` a překladač se pokusí hodnotu vypočítat.

Dojde-li následně ke změně nějaké funkce použité ve výrazu tak, že již nebude umožněn její výpočet v době překladu, překladač na tuto skutečnost upozorní chybou přímo u deklarace. V takovém případě lze situaci analyzovat. Byl-li výpočet za doby překladu otázkou pouze optimalizace a není striktně vyžadován `embed` kvalifikator se může jednoduše zamění na **`const`**.

Navíc tato chyba překladu může signalizovat nevhodnou nebo nekompatibilní změnu ve funkci (například při aktualizaci externí knihovny). Pokud se jedná o místo v kódu, kde je vyhodnocení v době překladu zásadní, přislušná deklarace zajistí, že se na tento problém upozorní ihned při překladu a umožní jej řešit v čas.

4.12.1 Obdobné chování v jinych jazycích

Obdobně to funguje v D, kde se dá využít enumerátorů (**`enum`**) k definici konstanty známé za doby překladu a spočíst hodnotu výrazu na pravé straně:

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

Nebo třeba v Zigu klíčové slovo `comptime` umožňuje vyhodnocovat bloky kódu v době překladu nebo vytvářet funkce s parametry známými v době překladu:

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

V C++ se používá klasifikátor `constexpr` k definici prvků (proměnných, funkcí, konstruktorů atd.) používaných a vyhodnotitelných v době překladu:

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

Není to tedy tak obecné jako v jiných případech, protože jen konstrukty označené jako `constexpr` mohou interagovat navzájem v `constexpr` kontextu. Například, aby bylo možné použít funkci ve výrazu pro `constexpr` proměnnou, , musí být i samotná funkce označena jako `constexpr`.

4.13 Následující vývoj

V předešlých kapitolách byly představeny, alespoň konceptuálně, definitivní návrhy syntaxe i funkcionality některých důležitých konceptů odlišných nebo neobsažených v C. Chtěl bych se však zmínit ještě o některých konceptech, které bych chtěl do jazyka nějakým způsobem zařadit, ale nepřišel jsem zatím s vhodným a uspokojivým návrhem.

4.13.1 Kontext

V jazycích např. Odin a Jai dochází k použití tzv. implicitního kontextu (context). Fakticky se jedná o ukazatel implicitně předávaný jako první skrytý argument do funkce s odkazem na buď lokální, nebo globální kontext programu (podobně jako `this` v případě objektu v C++).

V každé funkci pak lze přistupovat ke kontextu a získávat buď výchozí informace definovanou implicitně jazykem, nebo uživatelské informace doplňující definici kontextu. Například v Odinu lze přistoupit ke kontextu následujícím způsobem:[32]

Zdrojový kód 4.7 Ukazka kontextu [32] *Kód v jazyce Odin*

```
main :: proc() {
    // Přidá/nastaví proměnnou do aktuálního kontextu
        context.user_index = 456
    {
        // V tomto bloku můžeme kontext dále upravit
        // Změny jsou lokální pro tento blok

        // Nastavení vlastního alokátoru
        context.allocator = my_custom_allocator()
        context.user_index = 123

        // kontext bloku je implicitně předán do supertramp
        supertramp()
    }

    // Zde jsme zpět v původním kontextu bloku main
    // Hodnota context.allocator je původní
        assert(context.user_index == 456)
}

supertramp :: proc() {
    // Tento kontext je stejný jako v bloku volání
    c := context
```

```
    // V tomto příkladu bude c.user_index == 123
    // a c.allocator ukazuje na my_custom_allocator()

    // Procedury pro správu paměti používají context.allocator
    // jako výchozí, pokud není specifikován jiný
    ptr := new(int)
    free(ptr)
}
```

Velkou výhodu vidím v možnosti definici prostřednictvím kontextu vlastních alokátorů paměti a logovacích systémů. Pak se může jednoduše alternovat nastavení knihoven (a obecně cizího kódu) jednoduchou změnou chování standardních konstruktu jazyka.

Například, je-li knihovna napsaná využívající standardního způsobu alokace, v mém případě konstruktu alloc (viz 4.4), tak při jejím používání může být nastaven konkrétní uživatelský alokátor, který bude v daném kontextu použití výhodnější. Ze strany samotné knihovny pak nemusí existovat podpora, změna je prováděna v uživatelské části nastavením kontextu.

Obdobná využití kontextu mi přijdou zajímavá a užitečná. Nicméně, představa skrytého argumentu v každé funkci se mi příliš nezamlouvá a zatím jsem nepřišel se způsobem implementace, který by to nějak řešil, jestli vůbec existuje.

4.13.2 Metaprogramování

I když jazyk již obsahuje exekuci za doby překladu, tak ona samotná neřeší všechny problémy, které byla schopná řešit makra v C. Je tedy třeba nějakého metaprogramovacího nástroje, který by mohl plnohodnotně nahradit makra, ale být součástí jazyka.

Obvykle se metaprogramovaní projevuje v podobě obecných datových typů v definicích funkcí (třeba šablon (templates) v C++) umožňujících definici jedné funkce pro různé datové typy pokud lze s různými datovými typy pracovat identickým způsobem. Viz ukázky:

Zdrojový kód 4.8 Ukazka šablon *Kód v jazyce C++*

```
// Šablonová funkce pro sečtení dvou hodnot libovolného typu
template<typename T>
T add(T a, T b) {
    return a + b;
}

// Použití funkce
// int + int -> výsledek: 7 (typ je odvozen automaticky)
add(3, 4);
// double + double -> výsledek: 4.0
add(2.5, 1.5);
// const char* -> výsledek: "Ahoj světe" (pokud je přetížen +)
add("Ahoj ", "světe");
```

```
// Explicitní specifikace typu pomocí <int>
add<int>(10, 20);
```

Tento přístup je prakticky užitečný, ale neodráží veškerou sílu maker. I když bych obdobnou funkcionalitu chtěl implementovat, bylo by její zavedení nyní nežádoucí, protože by řešení mohlo být součástí většího, obecnějšího systému k jehož řádnému návrhu bych nejdříve chtěl specifikovat jasné požadavky na systém, což momentálně nedokážu.

5 Implementace překladače

K implementaci kompilatoru byl zvolen jazyk jazyk C++ ve standardu C++ 20. Vývoj a primární testování probíhalo na platformě Windows s využitím Visual Studio předkladače (`cl`) a vývojového prostředí Visual Studio 2022 k ladění.

Jako cílový IL jsem vybral jazyk C, především pro vysokou přenositelnost, širokou dostupnost nástrojů a mojí zkušenost. Pro překlad generovaného C kádu jsem zvolil Tiny C Compiler (TCC), konkrétně jeho knihovnu `libtcc`, zejména pro její malé rozměry, rychlost překladu a snadnou možnost vestavění přímo do aplikace. To eliminuje externí závislost na předinstalovaném C překladači a nabídne koncovému uživateli konvenční způsob interakce s překladem kódu. Ačkoliv jiné IL (např. LLVM IR) mohou nabízet pokročilejší možnosti, pro účely tohoto projektu, který spočíval v demonstraci konceptu jazyka, byla upřednostněna jednoduchost práce s C kódem.

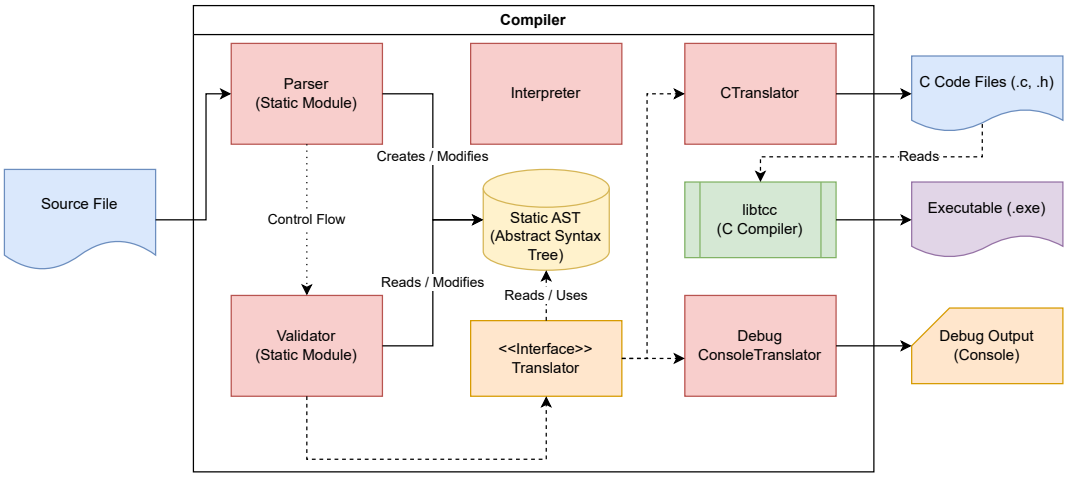
Jedná se o konzolovou aplikaci pracující se souborovým systémem. Využití standardní knihovny C++ 20 z větší části minimalizovalo závislost na konkrétní platformě. Odlišné zpracování vyžadovala implementace příkazu `run` (viz sekce **??**) a zajištění cest ke knihovnám potřebným pro běhové prostředí `libtcc`. Základní funkčnost překladače jsem úspěšně ověřil i na systému Ubuntu 22.04.5 LTS, což naznačuje dobrou přenositelnost na jiné Unixové systémy. Nicméně nelze vyloučit potřebu drobných úprav specifických pro dané prostředí nebo závislostí `libtcc`.

5.1 Struktura

Základní struktura a tok zpracování je následující. Proces počíná zpracováním argumentů příkazové řádky, které definují konfiguraci překladu. Tato konfigurace je využita hlavním statickým modulem překladače `Compiler`. Tento modul postupně provadí jednotlivé fáze překladu:

1. **Parsování:** Parser analyzuje vstupní soubor a inicializuje a naplňuje interní AST.
2. **Validace:** Následně modul `Validator` provede sémantickou analýzu a kontrolu vygenerovaného AST (např. kontrola typů, rozlišení jmen) a provede případnou doplňujicí anotaci AST novou sémantickou informaci.

3. **Překlad:** Validní a anotovaný AST je předán modulu `Translator`, který z něj generuje cílový kód (momentalně C).
4. **Exekuce/Tvorba spustitelného souboru:** V závislosti na argumentech příkazové řádky může být vygenerovaný kód následně přeložen do binární spustitelné podoby (pomocí `libtcc`) a případně ihned spuštěn.



Obrázek 5.1: Struktura implementováného překladače

Hlavním cílem této implementace bylo dospět k robustnímu základu, který by demonstroval funkčnost navrženého jazyka a sloužil jako platforma pro budoucí vývoj, směřující k idealní, plně optimalizované implementaci překladače. Klíčovým úkolem bylo navrhnout takovou strukturu AST, která by byla dostatečně flexibilní pro generování různých cílových reprezentací (zde C, potenciálně jiných IL) a zároveň by poskytovala dostatek informací pro vestavěný interpreter.

V této fázi vývoje nebyla primárním cílem rychlost překladu, ale především předložít ověření konceptu navrženého jazyka.

5.2 Uživatelské rozhrání

Pro komunikaci s uživatelem překladač využívá argumenty příkazové řádky. V základu uživatel specifikuje jeden ze tří hlavních příkazů, následovaný jménem vstupního souboru, na který se má příkaz aplikovat:

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 volby dostupné pro každy přikaz:

-ol (Output Language) Umožňuje vybrat cílový IL, prozatím jen C.

-of (Output File) Specifikuje jméno připadného výstupního spustitelného souboru (bez přípony).

-od (Output Directory) Určuje adresář, do kterého se uloží výstupní soubory (přeložené IL soubory, spustitelný soubor).

-gd (Generate Debug information) Pokud je volba přítomna, překladač se pokusí vygenerovat ladicí informaci kompatibilní s GDB/LLDB (formát DWARF).

-h (Help) Vypíše nápovědu k použití příkazů a voleb.

-b (Batch/Bash) Signalizuje překladači, že není spouštěn přímo uživatelem, ale skriptem. V tomto režimu příkaz se připadny přikaz `run` neprovede a při uspěchu program vrátí specificky kod. Volající skript pak může na základě tohoto kódu například spustit vygenerovaný spustitelný soubor specifickým zpusobem. Byl přidán, aby umožnil obejít problém s interakcí s terminálem (viz níže). Jedná se v zasadě o interní volbu.

Samotné zpracování argumentů příkazové řádky bylo realizováno trivialním způsobem. Využívá for smyčky a porovnávání řetězců pro identifikaci hlavního příkazu a následných voleb. Volby jsou nezavisle na přikazech a oboje jen konfigurují modul `Compiler`. Obecně se muže jednat o relativně dlouhodobé a postačujicí řešení, avšak v budoucnu, kdy počet argumentu muže byt kolosalní by se mohla použít předdefinovaná mapa pro rychlejší a snadnější vyhledávaní příkazů.

Jedinou implementačně zajímavější částí byla realizace příkazu `run`. Cílem bylo umožnit spuštění přeloženého programu bezprostředně po překladu tak, aby program běžel ve stejném terminálu a bylo možné s ním přímo komunikovat.

Použití standardní funkce `std::system` je sice jednoduché, ale původní proces překladače musí čekát na dokončení takto spuštěného procesu (má-li běžet na popředí ve stejném terminalu). Což je nevyhovující.

Na POSIXových systémech lze požadovaného chování snadno docílit kombinací systémových volání `fork` (vytvoření kopie procesu) a následného `exec` (nahrazení kopie procesu novým programem). Windows však nenabízí přímou obdobu `fork`. Disponuje funkci `CreateProcessA`, ktera sice umožňuje detailní nastavení nového procesu, ale neumožňuje úplné nahrazení procesu v rámci stejné konzole. I když se zajistí komunikace se stávajícím terminálem, předáním handles, po ukončení původního procesu překladače samotná konzole však detekuje ukončení procesu, který původně spustila (překladače), a v důsledku zobrazí nový příkazový řádek. To narušuje iluzi plynulého běhu.

Proto na platformě Windows pro docílení idealního chování bylo zvoleno pragmatické řešení využívající pomocný bash skript. Spustitelné soubory na Windows se nemusí volát s koncovkou, a tedy .bat soubor muže z uživatelského hlediska zastinit volání samotného programu. Bash skript nasledně zavola samotný překladač a nastaví volbu `-b`. Překladač program pouze zkompiluje a vrátí speciální

návratový kód signalizující úspěch. Volající bash skript tento kód zkontroluje a při uspěšném překladu následně spustí zkompilovany program. Tím je dosaženo požadovaného chování i když za cenu použití externího skriptu. Toto řešení reflektuje známá omezení standardních nástrojů Windows pro manipulaci s procesy v konzoli.

5.3 AST

Centrální datovou strukturou překladače je statické AST reprezentující strukturu, a i sémantiku zdrojového kódu. Základním stavebním kamenem stromu v této implementaci je uzel reprezentovaný strukturou `SyntaxNode`.

Každý uzel stromu nese základní informaci nezbytnou pro libovolný konkretní uzel stromu:

```
    struct SyntaxNode {
        NodeType type; // Typ uzlu (např. NT_WHILE_LOOP)
        Scope* scope;  // Rozsah platnosti, do kterého uzel patří
        Location* loc; // Umístění v původním zdrojovém kódu
        int parentIdx; // Index uzlu v rámci jeho rodiče
                       // pro data-flow analýzu závislou na pořadí
        ...
};
```

Každý tedy v zasadě obsahuje informaci o svém typu, rozsahu platnosti a přesném umístění ve zdrojovém kódu (např. užitelny pro hlášení chyb nebo generaci ladicí informaci). Avšak jsou zde umistěne i pomocné statické prvky (např. globální tabulka symbolů) pro zrychlení přístupu.

Konkrétní syntaktické konstrukce jazyka jsou pak reprezentovány strukturami, které dědí ze základní struktury `SyntaxNode` a obsahují doplňujicí atributy. Příkladem může sloužit uzel reprezentující cyklus **`while`**:

```
    struct WhileLoop : SyntaxNode {
        Operand* condition; // Reprezentace podmínky cyklu
        Scope* bodyScope;   // Reprezantace těla cyklu

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

I přesto, že `SyntaxNode` představuje uzel stromu, tak neslouži k vyjadření všech syntaktických elementu. Výrazy nejsou přímo reprezentovány jako uzly dědící ze `SyntaxNode`, ale tvoří vlastní hierarchii struktur. Důvodem toto rozdělení je konceptualní předpoklad jazyka, že teoreticky vše je spočitatelné za doby překladu, a tedy z pohledu samotného stromu je vždy vyraz vníman jako proměnna se svou hodnotou. Hodnota proměnne je pak vyjadřitelna vyrazem, ktery se dá počitat.

Obecný základ pro výrazy:

```
struct Expression {
    ExpressionType type; // Typ výrazu (např. EXT_BINARY_OP)
```

```
    ...
};
```

Konkrétní typy výrazů, například binární operace, pak dědí z `Expression`:

```
// Společný předek pro výrazy využivající operátory
struct OperatorExpression : Expression {
    OperatorEnum operType;
};

// Binární výraz
struct BinaryExpression : OperatorExpression {
    Operand* operandA; // Levý operand
    Operand* operandB; // Pravý operand

    BinaryExpression() { type = EXT_BINARY_OP; };
};
```

Je důležité si všimnout, že operandy nejsou typu `Expression*`, ale `Operand*`.

Struktura `Operand` slouží jako univerzální reprezentace hodnoty v AST — může představovat literál, odkaz na proměnnou, nebo výsledek komplexního výrazu. Její definice těsně spojuje informace potřebné pro standardní překlad se specifickými daty pro vestavěný interpreter:

```
    struct Operand : SyntaxNode {
        VariableDefinition* definition; // Definice proměnné
        Expression* expression;         // Výraz definující hodnotu

        // --- Informace pro překladač / sémantickou analýzu ---
        Value cvalue; // Hodnota známá/vypočtená při překladu

        // --- Atributy specifické pro vestavěný interpreter ---
        Value ivalue; // Aktuální hodnota během interpretace
        std::vector<Value> istack; // Zásobník volání
        int icnt; // Počítadlo rekurze pro interpreter

        ...
    };
```

Proměnná v jazyce je pak reprezentována jako pojmenovaný `Operand`. Hodnota operandu je tedy dána buď přímo, odkazem na definici, nebo výrazem. Samotná konkrétní hodnota a její datový typ jsou uloženy ve struktuře `Value`:

```
    struct Value {
        DataTypeEnum dataType; // Datový typ (DT_INT32)
        int hasValue = 0;      // Obsahuje-li konkretní hodnotu

        union {
            int32_t   i32;
            int64_t   i64;
            ...
            Array*    arr;
            ErrorSet* err;
```

```
            ...
        };
    };
```

Jak je patrné z atributů jako `ivalue`, `istack` a `icnt`, návrh klíčové struktury `Operand` je silně svázán s vestavěným interpreterem. Podrobněji se danné atributy popíší v souvislostí s implementaci interpreteru v odpovidajicí sekci 5.6.

5.4 Parsování

Při implementaci parseru jsem se rozhodl nepoužít standardní cestu generátorů (YACC/Bison) ani samostatný lexer (jako Lex/Flex) a zvolil jsem manuální implementaci, konkrétně metodou rekurzivního sestupu. K tomuto rozhodnutí mě vedlo několik důvodů:

Primárním cílem byla potencialní maximální kontrola nad kvalitou chybových hlášení překladače. Z mého pohledu, podpořeného zkušenostmí s YACC, dosažení detailních kontextově zavislých hlášek je v generovaných parserech komplikované. Manuální přístup mi poskytuje plnou kontrolu nad stavem analýzy v okamžiku vyskytu chyby a umožni formulovat uživateli srozumitelnější zpětnou vazbu, což považuji za duležity aspekt.

Manuální přístup mi dále nabízí přímou kontrolu nad správou paměti vstupních bufferů a případnou alokací dalších datových struktur přímo během syntaktické analýzy. Zároveň jsem se chtěl vyhnout mísení aplikační logiky se specifickým zápisem gramatiky v nástroji YACC. Takové spojení může podle mého názoru nekoherentnímu kódu, který je navíc logicky rozdělen mezi definiční soubor gramatiky a vlastní kód překladače. To z mého hlediska snižuje nejen čitelnost, ale i jednoduchost a robustnost celého procesu sestavení programu (vyžadován separátní krok generováce parseru).

Ačkoliv generátory nabízejí zjevné výhody, jako je automatická konstrukce efektivního stavového automatu pro syntaktickou analýzu a usnadnění údržby při změnách gramatiky, vyhodnotil jsem jejich přínos v kontextu tohoto projektu jako méně zásadní. I při změně gramatiky je totiž často nutné upravit související sémantické akce (např. konstrukci AST uzlů). Navíc, vzhledem k jasně definovanému stylu a směřování navrhovaného jazyka, nepředpokládám natolik radikální změny syntaxe v budoucnu, které by nebylo možné zohlednit běžnymi abstrakcemi při manuální implementaci. A tudíž bych výhodu snadnějšího prototypování syntaxe plně nevyužil. Navíc, argument potenciálně vyššího výkonu generovaného praseru nepovažuji za relevantní, protože, i když pomijím teoretickou možnost rychlejší implementace manualního řešení díky znalosti specifik jazyka, samotné parsování typicky nepředstavuje výkonnostní úzké hrdlo překladače.

Rozhodnutí neprovést abstrakci ve formě lexeru bylo vedeno čistou kurioznosti

ve snaze vyzkoušet alternativní přístup i na ukor méně modulárního kódu parseru.

5.4.1 Práce s pamětí při parsování

Každý relevantní zdrojový soubor je načten celý do paměti a zůstává v ní až do ukončení překladače. To umožňuje snadný přístup k částem kódu pro účely logování chyb a analýzy. Vychazí se z předpokladu, že obecně v libovolný okamžik překladač potřebuje přistup k libovlné informaci v libovolném souboru (především za učelm vypisu varovaní a chyb). Pro optimalizaci se pro řetězcové literály a identifikátory v AST nealokuje nová paměť, ale používají se přímo ukazatele do bufferů s načtenými zdrojovými soubory.

Takové to řešení je na relativní dlohou dobu vyvoje překladače postačující, i když se muže zdát, že udržení všech souboru v paměti je naročné, tak například zdrojové souboru jádra Linuxu zabirají „jen“ 1.5GB.

5.4.2 Zpracování importů

Systém importů řeší závislosti mezi soubory a detekuje připadne cyklické importy. K reprezentaci zavislostí souboru se používá používá stromová struktura vzuživajicí uzlu `ImportNode`:

```
    struct ImportNode {
        FileId* fileId;
        Scope* fileScope; // Kořen rozparsováného souboru
        ImportStatement* importStmtNode; // Uzel AST reprezentující import
        ImportNode* parent;
        std::vector<ImportNode*> children;
    };
```

Aby se předešlo opakovanému parsování souboru při zpracování importů, je zasadní je jednoznačně identifikovat. Id souboru bylo definovano jako čás tvorby a velikost:

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

Tento přístup umožňuje rychle porovnávat ruzné soubry, což bylo kličovym při potřebě neustale procházet strom a ověřovát cyklické importy. Alternativně by šlo použít například buď absolutní cesty, ktere obecně mohou byt dlouhé nebo hash souboru, ktera je pomalejší na vypočet. Přehled vyhod a nevyhod danych řešení je shrnuto v tabulce 5.1.

Rozparsované soubory (jejich hlavní `Scope`) jsou ukládány do mapy s `FileId` jako klíčem pro konvenční a rychly přistup. Samotný rekurzivní algoritmus zpracování importů lze zjednodušeně popsat pseudokódem 5.4.2;

Tabulka 5.1: Srovnání metod identifikace načtených souborů

Metoda	**Výhody**	**Nevýhody**
Cesta k souboru	Jednoduché použití; identifikace snadno čitelná člověkem.	Nespolehlivé při přesunu nebo přejmenování souboru; potenciálně velmi dlouhé cesty; problematické sdílení mezi různými systémy/uživateli.
Metadata (velikost + čas modifikace)	Rychlé zjištění; nízké paměťové nároky na uložení identifikátoru.	Nespolehlivé: možné kolize (změna obsahu bez změny velikosti/času); problematická granularita a spolehlivost časových značek napříč souborovými systémy a OS.
Hash obsahu	Velmi spolehlivá identifikace obsahu (*téměř* unikátní); nezávislost na názvu, cestě nebo metadatech souboru.	Výpočetně náročnější (nutnost přečíst celý soubor a spočítat hash); mírně vyšší paměťové nároky na uložení hashe.

Zdrojový kód 5.1: Algoritm zpracování importů

```
function processImport(parentNode):
    // Projdi všechny importy deklarované v parentNode
    for each import_node in parentNode.children:
        // 1. Získej reálnou cestu a identifikátor souboru
        real_path = getRealFilePath(import_node.importStmtNode.filename)
        import_node.fileId = generateFileId(real_path)

        // 2. Zkontroluj cache již zpracovaných souborů
        cached_scope = findInMap(parsedFiles, import_node.fileId)

        if cached_scope exists:
            // Soubor již byl naparsován, použij výsledek
            import_node.fileScope = cached_scope
        else:
            // Soubor ještě nebyl naparsován
            // 3a. Naparsuj soubor (rekurzivně najde další importy)
            parseFile(real_path, import_node)
            // 3b. Ulož výsledek do cache
            insertIntoMap(parsedFiles, import_node.fileId,
                import_node.fileScope)

        // 4. Zkontroluj cyklické závislosti projitím cesty k rootu
        if detectsCycle(parentNode, import_node.fileId):
            reportCircularImportErrorAndExit()

        // 5. Začleň importovaný scope do AST (např. pod jmenný prostor)
        integrateImportIntoAST(parentNode.fileScope, import_node)

    // 6. Rekurzivně zpracuj importy v nově načtených souborech
    for each import_node in parentNode.children:
        // Zpracuj pouze pokud nebyl z cache (abychom nešli znovu)
        if import_node.fileScope was newly parsed:
            processImport(import_node)
```

Je třeba podotknout, že pokud se používá pro ukládání potomků array list, může být vkládání importovaných symbolů na začátek `Scope`, pokud by to sémantika jazyka vyžadovala (např. vkladaní funkcí obecně nezavisi na pořadí a mužou se zařadit na konec), neefektivní kvůli nutnosti posunu existujících prvků. Možnou optimalizací je provést všechny importy pro daný soubor najednou a teprve poté provést jedno hromadné vložení/posunutí. Popřipadě zvažit přechod na jinou datovou strukturu, pokud by se statisticky jednalo o zavažny problem (pro ostatní použití je array list obecně vhodnější než například list).

5.4.3 Detaily implementace parseru

Parser byl implementován procedurálně, metodou rekurzivního sestupu, kde každému neterminálu typicky odpovidala buď funkce nebo přislušná větev switch-case ve více všestranné funkci, ktera měla za úkol rozpoznat a zpracovat přislušnou syntaktickou konstrukci. Funkce parseru pracují přímo s bufferem přislušného zdrojového souboru indexovanym pomoci structury `Location`, ktera ucho-

vavala i informaci o aktualním indexu a příslušném řádku.

Jak bylo zmíněno, absence samostatného lexeru vede k tomu, že rozhodování o následujícím kroku parsování a zpracování jednotlivých slov či symbolů se provádí individuálně v každé funkci parseru přímou manipulací s bufferem. To je méně přehledné a robustnější než přístup založený na tokenech, kde by bylo možné zobecnít libovolnou funkcí na `switch` příkazy nebo pole ukazatelu funkcí.

Klíčová slova jazyka jsou interně mapována na celočíselné konstanty, což umožňuje abstrakci od konkrétních řetězců a efektivnější zpracování. V současné implementaci se pro rozpoznávání klíčových slov používá rozsáhlý `switch`, který by v budoucnu bylo vhodné nahradit efektivnější metodou, například pomocí hash mapy.

Operátory jsou reprezentovány jako celočíselné hodnoty (`uint32_t`), což umožňuje jejich snadné zpracování, ale zároveň omezuje délku textové reprezentace na 4 znaky (případně 8 při použití `uint64_t`). Toto omezení se nikterak nejeví jako zásadní, protože většinou operatory jsou reprezentovany matematickymi symbolami nebo kratkymi slovy. Vyhledání odpovídajícího enumu operátoru podle jeho číselné reprezentace lze pak implementovat efektivně pomocí `switch` nebo indexace pole:

```
// Ilustrativní příklad vyhledání binárního operátoru
OperatorEnum findBinaryOperator(uint32_t word) {
    switch (word) {
        case /* hodnota pro '+' */ : return OP_ADDITION;
        // ... další operátory ...
        case /* hodnota pro '<<' */: return OP_SHIFT_LEFT;
        default                    : return OP_NONE; // Nenalezeno
    }
}
```

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 provaděno 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 připadě definice je dostupná pro proměnnou nachází-li se v rozsahu její platnosti. Ve druhém připadě navíc musí platít, že definice předchází proměnnou.

První připad se dá řešit jednodušě, stačí aby strukturá definujicí rozsáh platností vždy měla kontejner pro uložení odkazu na vyskytováné v ní definice. Vhodným kontejnerem je hash mapa, prtože ta zaroveň umožní zachovát identitu nazvu už při parsingu. Pak stačí v připadě 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 ditěté do rodíče.

Pokud by se situace zjednodušila a omezilo se jen na jeden soubor, tak řešit by to šlo na urovní parseru. Opět by se použilo hash mapy a každá proměnná by se testovála jako v prvním připadě přímo při parsingu. V takovém to připadě 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 zaručit, 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 duplikovat.

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 přidává 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 připadě 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 učely bylo dostačujicí. 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 (například i32 do i64) bez snižení přesnosti. Na což navazuje obobný připad akorat se sníženim přesnosti (například 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 zmíněno, 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. Ob-

dobně 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 sloužil 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 připadě 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. Zapisoál se kod, ktery bylo možné dat do main funkce.
- functions.h zapisovly se tam definice funkci, aby mohly byt dostupne napříč 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 řešil 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

Seznam literatury

[1] WIKIPEDIA. *Intermediate representation* [online]. 2025. [cit. 2025-04-04]. Dostupné z: https://en.wikipedia.org/wiki/Intermediate_representation.

[2] LATTNER, Chris. *LLVM: The Architecture of Open Source Applications (Volume 1)* [online]. 2011. [cit. 2025-04-04]. Dostupné z: https://aosabook.org/en/v1/llvm.html.

[3] MERRILL, Jason. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In: *Proceedings of the GCC Developers Summit*. Red Hat, Inc., 2003.

[4] WIKIPEDIA. *Java bytecode* [online]. 2025. [cit. 2025-04-04]. Dostupné z: https://en.wikipedia.org/wiki/Java_bytecode.

[5] WIKIPEDIA. *Common Intermediate Language* [online]. 2025. [cit. 2025-04-04]. Dostupné z: https://cs.wikipedia.org/wiki/Common_Intermediate_Language.

[6] GRUNE, Dick; REEUWIJK, Kees van; BAL, Henri E.; JACOBS, Ceriel J.H.; LANGENDOEN, Koen G. *Modern Compiler Design*. Second Edition. Springer, 2012. ISBN 978-1-4614-1202-3. Dostupné z DOI: 10.1007/978-1-4614-1202-3.

[7] COOPER, Keith D.; TORCZON, Linda. *Engineering a Compiler*. Second Edition. Morgan Kaufmann, 2012. ISBN 978-0-12-088478-0.

[8] JOHNSON, Stephen C. *Yacc: Yet Another Compiler-Compiler*. 1975. Tech. zpr., 32. Bell Laboratories.

[9] PARR, Terence. Preface. In: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013, relevant preface page numbers (if any). ISBN 978-1-934356-99-9. Dated 2012.

[10] JIANG, Tao; LI, Ming; RAVIKUMAR, Bala; REGAN, Kenneth W. Formal Grammars and Languages. In: ATALLAH, Mikhail J. (ed.). *Algorithms and Theory of Computation Handbook*. CRC Press, 1998. ISBN 978-0-8493-2649-3.

[11] MICROSOFT. *Announcing WebAssembly Language Support in VS Code* [online]. [cit. 2025-04-05]. Dostupné z: https://code.visualstudio.com/blogs/2024/05/08/wasm.

[12] MICROSOFT. *Extension Anatomy - Visual Studio Code* [online]. [cit. 2025-04-05]. Dostupné z: https://code.visualstudio.com/api/get-started/extension-anatomy.

[13] NEOVIM. *Neovim documentation: lua* [online]. [cit. 2025-04-05]. Dostupné z: https://neovim.io/doc/user/lua.html.
[14] MICROSOFT. *Syntax Highlight Guide - Visual Studio Code* [online]. [cit. 2025-04-05]. Dostupné z: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide.
[15] VIM DOCUMENTATION PROJECT. *Vim documentation: syntax* [online]. [cit. 2025-04-05]. Dostupné z: https://vimdoc.sourceforge.net/htmldoc/syntax.html.
[16] NEOVIM. *Neovim documentation: lsp* [online]. [cit. 2025-04-05]. Dostupné z: https://neovim.io/doc/user/lsp.html.
[17] MICROSOFT. *Language Server Extension Guide - Visual Studio Code* [online]. [cit. 2025-04-05]. Dostupné z: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide.
[18] MATSU. *Shiki: A beautiful syntax highlighter for the web.* [online]. [cit. 2025-04-05]. Dostupné z: https://shiki.matsu.io/.
[19] COOLWANGLU. *vim.js: JavaScript port of Vim* [online]. [cit. 2025-04-05]. Dostupné z: https://github.com/coolwanglu/vim.js.
[20] LLVM PROJECT. *Source Level Debugging with LLVM* [online]. [cit. 2025-04-05]. Dostupné z: https://llvm.org/docs/SourceLevelDebugging.html.
[21] MICROSOFT. *Visual Studio Debugger Documentation* [online]. [B.r.]. [cit. 2025-04-05]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/debugger/.
[22] EAGER, Michael J. *Introduction to the DWARF Debugging Format*. 2012. Often found online as a PDF document.
[23] MICROSOFT. *#line Directive (C/C++)* [online]. [cit. 2025-04-05]. Dostupné z: https://learn.microsoft.com/en-us/cpp/preprocessor/hash-line-directive-c-cpp?view=msvc-170.
[24] BILL, Ginger. *Frequently Asked Questions | Odin Programming Language* [online]. 2024. [cit. 2025-04-10]. Dostupné z: https://odin-lang.org/docs/faq/.
[25] MICROSOFT. *Welcome back to C++ - Modern C++* [online]. [cit. 2025-04-05]. Dostupné z: https://learn.microsoft.com/en-us/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=msvc-170.
[26] THE D LANGUAGE FOUNDATION. *Garbage Collection - D Programming Language* [online]. [cit. 2025-04-05]. Dostupné z: https://dlang.org/spec/garbage.html.
[27] THE D LANGUAGE FOUNDATION. *D Programming Language* [online]. [cit. 2025-04-05]. Dostupné z: https://dlang.org/.
[28] THE ZIG PROGRAMMING LANGUAGE. *The Zig Programming Language* [online]. [cit. 2025-04-05]. Dostupné z: https://ziglang.org/.
[29] THE ODIN PROGRAMMING LANGUAGE. *The Odin Programming Language* [online]. [cit. 2025-04-05]. Dostupné z: https://odin-lang.org/.

[30] LANGUAGE, Zig Programming. *Zig Documentation* [online]. [cit. 2025-04-05]. Dostupné z: https://ziglang.org/documentation/master.
[31] CPPREFERENCE.COM. *std::span - C++ Reference* [online]. 2025. [cit. 2025-04-09]. Dostupné z: https://en.cppreference.com/w/cpp/container/span.
[32] LANGUAGE, Odin Programming. *Odin Documentation* [online]. 2025. [cit. 2025-04-06]. Dostupné z: https://odin-lang.org/docs/overview/.
[33] LANGUAGE, D Programming. *Function Overloading - D Language Specification* [online]. 2025. [cit. 2025-04-06]. Dostupné z: https://dlang.org/spec/function.html#function-overloading.
[34] CPPREFERENCE.COM. *Overload Resolution - C++ Reference* [online]. 2025. [cit. 2025-04-06]. Dostupné z: https://en.cppreference.com/w/cpp/language/overload_resolution.
[35] PROJECT, ANTLR. *ANTLR Parser Generator*. [B.r.]. Dostupné také z: https://www.antlr.org/download.html. Accessed April 4, 2025.