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.

25. 3. 2025 Maxim Osolotkin

Návrh jazyka odvozeného z C a implementace nástrojů pro překlad

Abstrakt

Klíčová slova: programovací jazyk, překladač

Design of a C-derived language and compiler tools implementation

Abstract

Keywords: programming language, compiler

Poděkování

Obsah

Úvod

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

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

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

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

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

Nyní vás opustím a předám trpnému rodu.

1 Kompilátor

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

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

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

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

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

1.1 Přechodná reprezentace

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 Konstrukce kompilátoru

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

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

1.2.1 Lexikální a syntaktická analýza

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

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

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

Mezi takové nástroje patří třeba YACC a ANTLR.

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

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

1.2.2 Validace a linkování AST

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

1.2.3 Vykreslení

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

Vykreslení lze rozdělit v závislosti na finálním produktu.

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

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

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

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

1.3 Cross-Compilation

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

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

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

2 Gramatiky

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

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

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

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

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

Obecně pravidlo gramatiky můžeme vyjádřit jako zobrazení: [1]

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

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

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

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

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

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

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

2.1 Bezkontextová gramatika

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

Definice 2.3. Gramatika je bezkontextová právě tehdy, když pro každé pravidlo z P platí buď
$$S \to \varepsilon,$$
nebo
$$\alpha A \beta \to \alpha \gamma \beta,$$
kde
$$A \in N, \alpha, \beta \in (N \cup \Sigma \setminus \{S\})^* \text{ a } \gamma \in (N \cup \Sigma \setminus \{S\})^+$$

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

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

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

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

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

`goto` pravidlo bude třeba generovat slova jako

$$\text{goto FooLabel ;}$$

goto Me ;

goto UnhandledException ;

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

BNF zkraceno od Backus–Naur forma.

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

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

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

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

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

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

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

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

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

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

3 Vývojové nástroje

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

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

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

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

3.1 Zvýraznění kódu

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

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

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

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

3.1.1 Dokumentace

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

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

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

```
vim -c 'syntax on' -c 'TOhtml' -c 'wq' myfile.html
```

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

3.2 Language server protocol

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

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

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

bla bla bla

3.3 Debugger

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

3.3.1 Debug informace

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

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

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

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

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

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

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

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

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

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

3.4 Řešení

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

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

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

4 Návrh jazyka

Prvně bych vytvořil nějakou představu o vizi jazyka. Jazyk je nastroj, a jako každy nastroj by měl mit nějaky problem k řešení kterého by měl byt určen. Libovolný programovací jazyk řeší popis programu. Je tedy otázka jak a kterých programu. Odpověď bych viděl jako univerzalní proceduralní jazyk.

Jazyk by měl byt čitelný sam o sobě i na ukor osvědčeným postupum. Interpretace kodu po přečtení by měla co nejvíce odpovidat skutečnosti. Tedy, například, deklarace proměnné by neměla byt ve vychozím připadě konstatní, protože po přečtení kodu, ktery nespecifikuje vlastností deklarce je přirozenější se domnivát, že žadných vlastnotí nenabyva, než že jsou nějaké standardní skryté.

Jazyk by měl umožňovat jednoduchou a neomezenoun manipulaci s pamětí.

Protože

4.1 Drobností

Věcí, které stojí za zmínku, ale nejsou moc zavažné pro samostatnou kategorii.

4.1.1 Vstupní bod programu

Obvykle vstupním bodem programu ve vyšším programovacím jazyce je nějaká tzv. main funckce. Taková funkce muže mit za ukol předání vstupních argumentu programu a oddělení globalního scope.

Samotný koncept mi přijde obskurdním.

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

Sklonil bych se tedy k vstupnímu bodu programu jako k počátku souboru, obdobně jako v Pythonu, nebo, když mám vybírat z C-like jazyku, jak v HolyC.

4.1.2 Alokace

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

To sice nevyřeší ...

Syntaxi bych volil nasledujicí

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

Navíc bych umožníl při deklaraci vynachávat datový typ na pravé straně, jestli má byt schodný s tím na levé. Formalně je to možné, protože `alloc` je alternativní rvalue, oprotí výrazu `new` v C++ či D, ktrey by měl splňovat pravidla vyrazu.

Nasledujicí řadky by tedy vyjadřovali to same.

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

4.1.3 Komentáře

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

Volil bych nasledující syntaxi.

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

4.1.4 Datové typy

4.2 Pole

Jedna se o nejzákladnější a nejužitečnější datovou strukturu, která se v programovani vyskytuje. V C se ovšem pole daji používat jen ve statických případech, kdy velikost lze stanovit ještě při kompilaci. Ovšem, i v připadech, kde je to dostačující, tak při snaze využít pole jako argument funkce, tak buď se ona funkce musi definovat pro konkretni velikost pole, nebo využit ukazatele.

První případ je použitelný jen zřídka, jelikož nepřináší abstrakci, která se intuitivně pojí s voldbou datového typu, muselá by se vytvořít pro různé velikosti poli samostatná funkce. To se řeší předaním pole přes pointer a, po případě, ukončením pole nějakým specifickým symbolem, nebo předáním doplňujícího parametru délky.

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

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

4.2.1 Délka pole

Nasledně bych zavedl získatelnou délku pole.

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

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

4.2.2 Typy poli

Po mimo klasického rozdělení poli na statická a dynamicka, bych chtěl umožnít jejich dělení v zavislostí na variabilnosti délky. To by umožnílo vytvařet více specifické rozhraní pro využírí poli ve funkcích. Napřiklad by `int[const]` by vyznačovalo konstantní délku.

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

Pole konstantní compile-time známé délky

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

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

By vytvořilo pole o delce 8. Spočtená délka by se vždy dosazovala compile-time, realná proměnna by se pro jeji uchovaní negenerovala.

Pole konstantní run-time známé délky

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

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

Tedy

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

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

Pole variabilní run-time známé délky

Analogie využití ukazatele jako pole v C.

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

Tedy

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

By alokovalo pole o delce 8 na heapu a vygenerovalo by přislušnou proměnnou pro uložení délky někde v pamětí programu. Pole by šlo realokovat.

Array List

Šel by vytvořít nasledovně

```
int[] arr;
```

nebo se specifickou počateční delkou, v tomto připadě 8.

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

Volba kvalifikatoru

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

Taková to varianta se protířečí s zakladní představou o jazyce viz. Ale na druhou stranu se zbavuje skrytého flow, kdy se akce, ktera nese v sobě implicitní alokace není jako vychozí, ale musí se konkretně zvolit.

Příklady jiných jazyku

C++ ponecháva vše jak v C. Řeší vše za pomoci konteineru definovaných v std knihovně. Nema však analog dynamického pole, ale nabizí std::span, který muže sloužít jako interface pro souvislý kontejner, nemuže však sloužit přímo jako kontejner.

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

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

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

Odin ma k dispozici statická pole, slicy, ktere se využivají obdobně jako std::span, ale umožňují alokaci, a tzv dynamicka pole, ktere je analogem array listu pro jehož deklaraci se využívá kvalifikatoru na mistě délky. Všechny maji dostupnou funkcí len vracející jejich delku a dynamické pole muže využit funkce cap k získaní realné alokované délky.

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

4.2.3 Práce s polem

Protože cílem je vytknout identitu pole oprotí pointru, tak bych zdurazníl rozdíl mezí nimi i z pohledu prace s daty. Pole je na rozdíl od pointru kus pamětí obsahující stejně elementy za sebou.Tedy bych pole vnímal jako proměnnou definující stejné prvky. Akce provaděne se samotnou proměnnou jako takovou (bez indexace), by se tykalí všech prvku v poli.

I když C fakticky rozděluje mezi ukazatelem a polem, tak parkticky struggles identity. Při nasledné deklarace pole

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

se vytvoří kontejner pro osum proměnných: `arr[0]` .. `arr[7]` o datovém typu `int`, který se specifikuje jen jednou. Použíje-li se kvalisifikator, tak se taky aplikuje na všechny prvky.

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

Aritmetické operace se však chovají k proměnné jako k pointru a nejou aplikované na všechny prvky.

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

V dusledku dvojakosti nasledující přířazení není dovoleno,

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

jelikož pak není jasny zda se jedna o přiřazení všech prvku arr2 do arr1, nebo přepsaní ukazatele arr1 na ukazatel arr2. Což není z přečtení kodu zjevné, jelikož standardně se využivá kvalifisikatoru const k zakazaní přiřazení do proměnné. Navíc, syntaxe řešící onen problem již v jazyce je, což přinaší jetě špetku zmatku.

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

Jasné rozdělení mezi ukazatelem a polem na urovní typu umožňuje se vyhnout podobným problemum. Navrhoval bych, že kvalifikatory jsou vždy vztažené ke všem proměnným v poli, že přiřazení pole do pole je definováno jako přiřazení jednotlivých prvku poli, že libovolné operace jsou vztaženy vždy na všechny prvky pole. Vyjimkou by bylo přířazení pole do ukazatele, kde se jedna o cast, a operace concatinace viz[].

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

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

Nebo sečíst dva pole

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

atd.

4.3 String

V C string literaly jsou pouze hezčí verzí zapsani pole constantních charu. Vcelku, i když je to primitivní, tak zcela postačující. Problemem je zde 'absence' identity pole jako typu, jak již bylo zmiňěno viz.[]. Tedy vlastně se s každym stringem pracuje jako s pointrem.

Jelikož já definují pole jinak, tak lze rozvinout možnosti pole, aby umožňovali ve vysledku lehčí praci i se stringy. Samotný typ pro string existovat nebude, ale bude jen podpora string literalu, ktery se při kompilaci rozloží na pole.

4.3.1 UTF-8

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

Protože symboly v utf8 jsou variablní délky, víděl bych jako nejlepší možnost compiletime vyhodnocení nevětší delky potřebné pro uložení jednoho symbolu přislušného literalu a naslednou konverzi na pole intu o patřičné velikosti, kde každy element bude samostatnym symbolem zakodovaným v utf8.

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

To umožní pracovát se symboly samostatně, využívát všechny vyhody pole a mit pro ASCII text stejně velké pole jako v C. Tedy, například, leva strana muže definovat libovolnou velikost k uložení symbolu.

```
int32 str = "čau";
```

Aby bylo umožněno pracovat se string literaly jako prostě s kusem pamětí, tak bych přidal možnost definovat tzv. raw string literal nasledujicím zpusobem.

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

Jiné jazyky

V D jsou string literaly jsou standardně v utf8 formatu jako imutable pole znaku, ale za pomoci postfixu u string literalu lze je konvertovát do wcharu(utf16), či dcharu (utf32).

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

V Odinu jsou stringy taky utf8, má ale koncepci tzv. rune umožnujicích operovat v takových stringach po samostatných symbolech, kde prave rune represents a Unicode code point. Ma take specialní datový typ reprezentujicí null-terminated stringy s C, cstring.

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

V Zig string literal je faktický jen null-terminated pole bytu, které lze zapsat pomoci utf8 symbolu. Další rozšíření prace Prace je skrz standardní knihovnu.

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

4.3.2 Operace

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

Zřetězení

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

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

Jelikož delká libovolného pole je získatelná, lze to zobecnít na jaky koliv typ pole. Je však duležité, aby implementace neobsahovalá žadnou alokaci pamětí, bylo by to

zavadějicí. Tedy připadné vysledné pole by se muselo samostatně standardně alokavt prostředky jazyka.

Pro dynamické pole by to mohlo vypadadat nasledně

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

Slice

Se tzv. slice se dá setkat v mnoha jazykach v té nebo jiné podobě. Jedná se o reprezentací jen libovolné souvislé častí pole, ktera sama o sobě neobsahuje data, ale odkazuje se na puvodní pole. Tedy v C by se slice dal definovat třeba nasledujicím zpusobem

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

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

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

Já bych nevnímal slice jako datový typ, ale spiš jako operací , protože ve své podstatě je to to same. . . .

4.4 Namespace

Zručný nástroj k organizací kodu. Umožňuje zhlukovat proměnné pod jedním společným názvem, který je rozlišitelný parsrem. Na rozdíl od použití identifikujících prefixu / postfixu v názvech je strukturním celkem z hlediska nástrojů operujících s kódem (např LSP).

Umožňuje také při kompilaci hromadně pracovat s vevnitř definovanými proměnnými, a tedy se dá dobře využívat i pro import a export částí kódů. Například v Python

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

To, po mimo jiné také umožňuje řešit kolizí názvu při importování knihovén (pokud nejste C++).

Namespace bych viděl jednoduše jako pojmenovaný scope.

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

Nebo možná prostě

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

K přístupu prvku bych využíl se syntaxe z C++ `Foo::x;`.

4.5 Import

Klasificiroval bych hlavičkové soubory a s ními související systém importu jako nejhorší část C. Hlavní nevyhodou kterých je duplicita definic. Slouží však k dobrému úmyslů, k izolaci implementace a definici rozhraní. Cílem tedy bude tuto myšlenku ponechat, a vyhnout se jak preprocesoru, tak i duplicitě.

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

Intitivně se nabizí možnost přímého importu souboru nasledujicí syntaxi.

```
import filename;
```

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

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

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

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

Syntakticky se specialně specifikuje `namespace`, protože by onen konstrukt mohl byt využít k implementaci jiných zpusobu zabalení souboru.

Např.

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

apod.

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

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

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

Lze buď vycházet z toho, že vše je viditelńe, a my omezujeme viditelnost, nebo naopak, vše je nepřístupné, a my rozšiřujeme přístup. Druhy přístup je víc prakticky, ale

ztrácí na explicitnosti, protože, když importujeme soubor, tak intuitivně očekáváme, že se nám tam naimportuje všechno (nebo alespoň něco), než nic.

Podstatnější je otázka viditelnosti vnořených importu. Tedy, importuje-li soubor identifikátory z jiného souboru, budou-li viditelné také. Zřejmé je, že pokud jsou p5istupné při importu, tak by měly být přístupně i pro další importy, jelikož jsou na stejné úrovní jako kód souboru, a nekladli jsme žádným způsobem omezení.

Tedy, navrhoval bych umožnit omezit viditelnost importu, než omezovat viditelnost samostatných identifikátorů. Pak by byla decentní explicitní možnost omezení viditelnosti symbolu, aniž by se to muselo řešit poprvkově, a navíc by jsme stale měli možnost vytvoření připadného rozhraní z dostupných symbolu, ktere by se umisitili do jednoho souboru a zbyle by se importovali lokalně.

K označení lokalních importu bych použíl slovo `local`

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

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

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

4.6 Function Overloading

I když se jedna o implicitní konstrukt, ktery skryva od čtenáře pravou identitu volané funkce, tak přinaši, z mého hlediska, jednu zasadní věc, zjednošuená jmena funkcí. Tedy, zamisto třeba vepisovani datového typu do jmena funkce, k rozlíšení funkce, lze jen uvest jeji činnost.

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

Navíc jmena samotných funkci s použitím postfixu/prefixu, které si zvolime pro identifikaci, nebudou samostatnými celky z hlediska nastroju pracujicich s kodem, tedy v zakladu samotným kompilatorem a např LSP.

Samotná abstrakce nad konkretní volanou funkci pro četnáře není nikterak zavadějící. Nebo spíš, je stejně zavadějící jako for loop, ktery za misto abstraktní intrukce for provadí skoky a sem a tam. Smysl čtenář získava ze samotného nazvu funkce a vstupních proměnných, a vníma konkretní funkcí jako černou skříňku. I když ona funkce dostava int, tak nemuže vědět, že ten int neni hned první instrukci přetypovan do floatu. Tedy jediné co to ovlivní je rychlost nalezení spravné funkce při potřebě se podivát na jeji kod, což, bez užití LSP, bude zřejmě delší, ovšem, neřekl bych, že se jedná o něco zavažného.

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

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

nebo pro jiny datový typ musi byt explicitní cast

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

Explicitním vepisem datového typu se identifikuje volaná funkce. Ovšem, existuje-li potřebná varianta se dá dozvědět při psání kodu jen z LSP. V takovém to případě je cast, jen z hlediska informace, navíc (porovnavame-li s implicitnim overloadingem). Nebo až po kompilací.

Tedy, faktické využití explicitního overloadingu je jako assert, kdy se explicitně vyžaduje konkretní varianta funkce a v případě absencí se očekavá chyba při kompilací. Ovšem, cast se bude muset specifikovat u každého volaní overloaded funkce, což se pře s tím, že overloading zavadím hlavně z duvodu zjednodušené jmeńné stopy. A to nemluvě o tom, že vlastně tutež informaci existuje dva krát v kodě, jednou při definici, po druhy při volání.

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

```
foo!();
```

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

4.6.1 Implementace

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

My budeme postupovat obdobně.

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

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

4.6.2 Přistup jiných jazyku

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

Zig nema function overloading, ale podobného chování lze docilit při kompilaci za pomoci tzv. duck typing.

D a C++ maji implicitní function overloading.

4.7 Spravá chyb

Uvažuje-li se C, tak jazyk nenabizí přímý způsob správy chyb. Chyby se mohou řešít například navratovou hodnotou, nějakým specifickým stavem očekavané vystupní promměnné předane přes ukazatel (většinou `NULL`), specialní funkci, ktera vraci poslední chybu atd. V zasadě je to na programatorovi, aby vytvořil nějaky system pro spravu chyb, a jestli vubec.

Při práci s libovolným kodem je pak nutne číst komentaře k funkcím, externí dokumentaci apod. To opět vede na problem, kdy duležita informace není současti strukturních elementu kodu, ke kterym by měly ruzné nastroje přístup. Take to postrada jednotnost, kde různé knihovny mohu řešit zpravu chyb vždy jinak, a ve vysledném programu se bude muset řešít zbytečný problem, jak s tím naložit.

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

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

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

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

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

S try-catch se většínou pojí i tzv. throw mechanizmus umožňujicí označit připadné chyby, které muže kod nějaké funkce vyvolat, a propagovat jejich ošet-

řění do bloku, jež onu funkci volal.

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

4.7.1 Definice požadavku

Neprve bych si definoval požadavky na chybový system:

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

4.7.2 Implementace

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

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

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

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

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

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

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

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

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

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

Navrholval bych nasledujíci syntaxi.

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

Kde se připadná chyba uloží do proměnné err.

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

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

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

Množiny chyb

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

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

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

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

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

```
        ErrorB;
    };
```

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

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

K definici chybového rozharní funkce se pak použije nasledujicí syntaxi.

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

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

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

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

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

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

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

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

Navrat chyby

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

Volil bych nasledující intuitivní syntaxi:

```
return value, err;
```

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

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

```
return _, err;
```

Implementace v jiných jazycích

4.8 Kontext

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

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

print, mem alloc, etc.

4.8.1 custom alloc

4.9 Compile-time exekuce

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

V C++

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

5 Implementace kompilatoru

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

Jedná se o konzolovou aplikaci, která prácuje se soubory na disku. Tedy, zavislot na operačním systemu není tak velka a Unix verze nebude složíta k realizaci. Jedná se třeba o rozdíly v ziskaní getExePath cesty, která nejde získat skrz std::filesystem. Navíc, je zde rozdíl v edianech, který mohl nebyt někde v kodě zohldněny, protože se často operovalo s byty. Proto, i když samotný program se kompiluje pod linux a i funguje, tak nemužu zaručít jeho funkí bez řadného testování.

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

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

Zatím se za cíl nekladla rychlost řešení, ale nabidnout proff of koncept navrženemu jazyku, ktery by se dal využit...

5.1 Uživatelské rozhrání

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

aplikovane:

build Přikaz sestaví program do spustitelné podoby.

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

translate Přikaz pouze přeloží do vybraného IL.

Nasledně mohou byt uvedené doplňujicí možností dostupné pro každy přikaz:

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

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

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

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

-h (Help) Vypiše napovědu ohledně uživatelského rozhrání.

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

Přikaz run potřeboval umožnít spustít zkompilovaný program po kompilaci. Lze třeba použit std::system, ale protože ... . Lze použít napřímo systemových funkcí, v připadě windows se jedná o CreateProcessA, ktera umožní vytvořít process a specifikovat ruzné možností, napřiklad jako, jeslti se má process zpustit v nové konzoli. Pro připad emulace, lze použít WaitForSingleObject, což umožní počkát na vykonání kodu programu. Tedy ze strany uživatele se to bude chovát jako jeden program.

Já jsem volil cestu ukončení hlavního programu a

5.2 AST

Centerm programu je statické AST jehož uzel se reprezentuje za pomoci struktury SyntaxNode. Kořen stromu je statickým prvkem SyntaxNode, což děla zapis . SyntaxNode obsahuje i další statické prvky, které odkazují na kontejnery s cachovanou informaci, ktera by byla dostupná přímo, aniž by se musel procházet strom, jako odkazy na proměnné, na definice, funkce atd. Samotná definice pak vypadala obdobně:

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

Každy uzel teda ve své podstatě nese informaci o svém typu, v jakém scope se nachází a lokaci ve zdrojovém kodě.

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

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

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

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

Obecný výraz je definovan velmi jednoduše:

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

Pak konkretní výraz vypadá například nasledovně:

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

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

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

Vyjadření hodnoty proměnné je pak představeno nasledovně:

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

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

    std::vector<Value> istack;

    Expression* expression;
    ...
}
```

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

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

Jak lze usoudít z atributu ivalue a istack v definici operandu, doplňující abstrakce nad hodnotou ve formě Value je především z hlediska interpreteru, ktery s onym celkém pracuje. Podrobnějc oné atributy budou představené v přislušné sekci [].

5.3 Parsing

Rozhodl jsem se nejit cestou generatoru jako je YACC, protože, ze zkušeností, bych stejně musel napsat veškerý kod pro sestavení stromu, neměl bych jednoduchý zpusob vypisování chyb zpusobem, kterým bych chtěl, měl bych omezeny přistup k buffrum souboru, neměl bych kontrolu nad paměti atd. Navíc bych nevyužíl vyhod při protytopování nad syntaxi, protože mam konktretní typ jazyka, který chcí implementovát, tedy bych jen mohl implementovát nutnou abstrakci pro zaměnu syntaktických celku.

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

5.3.1 Importy

Zvnějšku parser je jen jedná funkce přijimajicí na vstup jméno souboru, ktery oná vníma jako main soubor programu. Vnítřně oná již využívá funkce k parsingu samotných souboru. Každy soubor se parsuje do předem dodané instance Scope. Při parsování souboru se připadné importy zařazují postupně do fronty. Prvním rozparsovaným souborem je main. Každy připadný importovaný soubor je nasledně parsován a vysledek převaděn do potřebné podoby a zařazen do stromu.

Aby se předešlo parsování 2x téhož souboru, každy rozparsovaný soubor se označí unikatním id. Id je ve formě:

Protož

Nevyhodou je

Je nutné podotknout, že importovat symboly dopředu scope je relativně drahé, protože se prvky v každém array-list budou muset posunout. V připadě třeba funkci nezaleží na pořadí, protože neosahují sam o sobě spustitelný kod a jejich definice

nemusí předcházet použití. Tedy je lze prostě zařadít nakonec. V připadech, kdy to nejde, krom zřejmé změny kontejneru na třeba list, lze alespoň posunout prvky jen jednou za soubor udělav posun až po zpracování všech imporu.

5.3.2 Samotný parsing

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

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

5.4 Validace AST

5.5 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.

Ke kompilací C kodu

5.6 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