
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# PŘEKLADAČ NOVÉHO MODULÁRNÍHO PROGRAMOVACÍHO JAZYKA

**COMPILER FOR A NEW MODULAR PROGRAMMING LANGUAGE**

**SEMESTRÁLNÍ PROJEKT**
**TERM PROJECT**


**AUTOR PRÁCE** **DANIEL ČEJCHAN**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. ZBYNĚK KŘIVKA, Ph.D.**
**SUPERVISOR**



**BRNO 2017**

# Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.



# Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.



# Klíčová slova

Líhnutí kódu, programovací jazyk, CTFE, metaprogramování, OOP



# Keywords

Code Hatching, programming language, CTFE, metaprogramming, OOP



# Citace

ČEJCHAN, Daniel. *Překladač nového modulárního programovacího jazyka*. Brno, 2017. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Křivka Zbyňek.

# Překladač nového modulárního programovacího jazyka

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph. D. ... Další informace mi poskytli... Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Daniel Čejchan
28. ledna 2017

## Poděkování

Velký dík patří i Stefanu Kochovi, který se na internetu vyskytuje pod přezdívkou UplinkCoder; ve své ochotě se mnou prodebatoval velkou část konceptu jazyka.

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

# Obsah

# Kapitola 1

# Úvod

Tento dokument popisuje a zdůvodňuje ta nejdůležitější rozhodnutí, která byla vykonána při procesu návrhu imperativního kompilovaného programovacího jazyka Beast a vzorového překladače pro něj. Rámcově prozkoumává syntaktické i sémantické prvky moderních programovacích jazyků a popisuje i nové koncepty a prvky.

Tato práce se zejména zaměřuje na nový koncept „líhnutí kódu," který má poskytovat rozsáhlé možnosti metaprogramování v kompilovaných jazycích. Navržený programovací jazyk se hodně inspiruje jazyky C++ a D[1]. Tyto jazyky budou používány pro srovnávání syntaxe a efektivity psaní kódu.

Příklady kódů z jazyka Beast uvedené v tomto dokumentu čerpají z kompletní specifikace jazyka, případně pracují se standardní knihovnou (která ještě ani nebyla navržena); kompilátor vytvořený v rámci bakalářské práce ale implementuje pouze malou část zdokumentované funkčnosti – většinu uvedených kódů tedy kompilátor není schopný zpracovat.

Motivací a současně i cílem této práce je snaha vnést nový pohled do oblasti kompilovaných jazyků, jak celkovou kompozicí jazyka, tak novými nápady, a přiblížit jej akademické komunitě skrze demonstrační kompilátor.

[1] http://dlang.org/

# Kapitola 2

# Vlastnosti a syntaxe jazyka

Naším cílem je navrhnout jazyk pro obecné použití, který by se dal použít na maximálním počtu platforem, například i ve vestavěných systémech. Složitější struktury se tím pádem budeme snažit řešit spíše vhodnou abstrakcí než zaváděním prvků, které kladou zvýšené nároky na výkon a paměť.

Budeme navrhovat kompilovaný jazyk. Ačkoli interpretované jazyky mají jisté výhody, platí se za ně pomalejším kódem a nutností zavádět interpret. Pro zjednodušení práce při psaní překladače nebudeme ale překládat přímo do strojového kódu, ale do jazyka C/C++.

Abychom maximálně usnadnili přechod programátorů k našemu jazyku a zlepšili parametry křivky učení, je vhodné se inspirovat již existujícími jazyky. Vzhledem k tomu, že se jedná o kompilovaný jazyk, je nejrozumnější vycházet ze syntaxe rodiny jazyků C, které jsou hojně rozšířené a zažité. Tento jazyk se inspiruje konkrétně jazyky C++ a D. Z těchto jazyků bude přejímat i paradigmata a základní koncepty. Náš jazyk bude tedy umožňovat strukturované, funkcionální i objektově orientované programování.

## 2.1 Existující prvky programovacích jazyků

V oblasti návrhu programovacích jazyků již bylo vytvořeno mnoho konceptů a kategorizováno mnoho vlastností, které jazyk definují. V tomto oddílu si přiblížíme ty nejznámější z nich a budeme diskutovat o jejich začlenění do našeho programovacího jazyka.

Informace v tomto oddílu jsou čerpány z referencí programovacích jazyků C++[4], D[2], Rust[3] a tutoriálových stránek pro jazyky Java[1] a Scala[10].

### 2.1.1 Modularita

Pro větší projekty, které jazyk Beast také cílí podporovat, je rozdělení kódu do menších částí – modulů – nezbytné. Přístup, kde u modulů byla oddělená deklarační (hlavičková) a definiční část (například u jazyků Object Pascal a C/C++), byl již překonán a kompilátory moderních jazyků již rozhraní mezi moduly odvozují z definic uvnitř jednotlivých modulů. Tento přístup je praktičtější, protože vytváření a udržování hlavičkových sekcí bylo časově náročné (programátor jinak musel vše psát a upravovat dvakrát). Jazyk Beast zavede systém modulů podobný jazykům D a Java, které jsou vytvořeny v tomto duchu.

### 2.1.2 Systém dedičnosti tříd

U kompilovaných jazyků se nejběžněji využívá třídní systém využívající dědičnost a tabulky virtuálních metod. Tohoto konceptu využívají i jazyky C++ a D, nicméně v detailech se poněkud liší. Zatímco v C++ existuje jen jeden typ objektu, který pracuje s dědičností (*class* a *struct* jsou z pohledu dědění identické), a to s dědičností vícenásobnou a případně i virtuální, D používá obdobný systém jako Java – třídy (*class*) mohou mít maximálně jednu rodičovskou třídu (myslí se jeden přímý předek, rodičovská třída může dědit od další třídy), navíc ale existují rozhraní (*interfaces*). Ta však mohou obsahovat pouze funkce (tedy žádné proměnné), což omezuje jejich možnosti.

Jazyk D poskytuje k vícenásobné dědičnosti částečnou alternativu ve formě tzv. *template mixins*[1] – které fungují podobně jako kopírování bloků kódu přímo do těla třídy. Ty se kombinují s rozhraními – funkce deklarované v rozhraních obsahují implementace v *mixins*, které se vkládají do těl tříd. Proměnné jsou napodobeny *gettery* a *settery*, se kterými se díky tzv. *property functions*[2] dá pracovat téměř jako s proměnnými.

```
1  // D
2  mixin template ITestImpl() {
3
4  private:
5    int x_;
6
7  public:
8    void x( int set ) {
9      x_ = set;
10   }
11   int x() {
12     return x_;
13   }
14   void foo() {
15     // Do something
16   }
17
18 }
19
20 interface ITest {
21   void x( int set );
22   int x();
23   void foo();
24 }
25
26 class Parent {
27 }
28
29 class Child : Parent, ITest {
30   mixin ITestImpl;
31 }
32
33 void main() {
34   Child child = new Child;
35   child.x = 5; // Ok (equals to x(5) )
36   child.x += 5; // Error, this is not possible
37
```

[1] https://dlang.org/spec/template-mixin.html
[2] Viz https://dlang.org/spec/function.html#property-functions

```
38    ITest itest = cast( ITest ) child;
39    itest.foo();
40  }
```

(2.1) Příklad použití *mixin template* v jazyce D

Použití *mixin templates* není ale plnohodnotnou náhražkou vícenásobné dědičnosti:

- Rozbíjí model dědičnosti (na *mixins* se nelze odkazovat, proto se kombinují s rozhraními).
- Protože jsou vložené funkce prakticky součástí třídy, nefunguje v určitých případech kontrola přepisování (*overridingu*).
- Programátor musí udržovat zvlášť deklarace v rozhraních a definice v *mixin templates*.

Ačkoli je implementace systému vícenásobné dědičnosti tak, jak je v C++, složitější, její implementace je uskutečnitelná, kód nezpomaluje (v porovnání s kódem s ekvivalentní funkčností napsaným alternativním způsobem) a fakticky rozšiřuje možnosti jazyka. C++ model dědičnosti umí vše, co umí Javovský model, a ještě víc. Proto bude jazyk Beast umožňovat vícenásobnou třídní dědičnost; nicméně implementace systému třídní dědičnosti není primárním cílem projektu, a tak bude systém v rámci demonstračního kompilátoru značně omezen.

### 2.1.3 Automatická správa paměti – garbage collector

Zavedení *garbage collectoru* (dále jen GC) nenabízí pouze výhody [15] – programy mohou být pomalejší, GC zvyšuje nároky na CPU a paměť (těžko se zavádí v mikroprocesorech); navíc efektivní implementace GC je velice složitá.

Rozumným východiskem se jeví být volitelné používání automatické správy paměti. Beast by měl umožňovat efektivní napsání GC přímo v jazyce; GC by tedy mohla být jedna ze základních knihoven. Vzhledem k už tak velkému rozsahu plánované práce je však GC jen plánem do budoucna.

### 2.1.4 Implicitní konstantnost proměnných

Koncept konstantnosti proměnných byl zaveden jednak prvek statické kontroly při psaní kódu. Označení proměnné za konstantní ale u jazyku C++ (i D, Java, ...) vyžaduje napsání dalšího slova (modifikátoru *const* u C, C++ a D, *final* u Javy), a tak tuto praktiku (označovat všechno, co se dá, jako konstantní) spousta programátorů neaplikuje, zčásti kvůli lenosti, zčásti kvůli zapomnětlivosti. Konstantnost je, když už je zavedena, třeba dodržovat v celém projektu (díky jejím tranzitivním vlastnostem) a její pozdní zavedení bývá pracné a zpravidla nelze udělat postupně kvůli tzv. *const poisoning* efektu – zavedení konstantní korektnosti [11][8] v části kódu vyžaduje konstantní korektnost všech typů, které daný kód používá (a ty typy vyžadují konstantní korektnost všech funkcí, které využívají, a tak dále).

Některé jazyky (například Rust[3]) přišly s opačným přístupem: všechny proměnné jsou implicitně konstantní a programátor musí pro deklaraci nekonstantních proměnných použít speciální syntaktickou konstrukci. Tento přístup má vést programátory ke korektnímu používání konstantních a mutabilních proměnných. Jazyk Beast tento přístup také aplikuje.

[3] https://doc.rust-lang.org/nightly/book/mutability.html

Je tedy třeba určit syntaktickou konstrukci pro označování nekonstantních proměnných. V rámci koherence syntaxe jazyka (která je rozvedena v dalších kapitolách tohoto textu), připadají v úvahu dvě možnosti:

1. Vytvoření dekorátoru[4] v kontextu `typeWrapper`[5], nejlogičtěji `@mutable` nebo `@mut`
2. Vyčlenění operátoru; nejlepším kandidátem je suffixový operátor `Type!`, protože nemá žádnou standardní sémantiku, je nepoužitý a znak vykřičníku je intuitivně asociován s výstrahou, což je asociovatelné s mutabilitou.

Při návrhu jazyka Beast byla zvolena druhá možnost, především kvůli „upovídanosti“ kódu a ještě z dalšího důvodu, který je popsán v následujícím oddílu.

### 2.1.5 Tranzitivita konstantnosti přes reference

Jazyk D je navržen tak, že je-li ukazatel konstantní (nelze měnit adresu, na kterou ukazuje), je přístup k paměti, na kterou ukazuje, také konstantní. Není tedy možné mít konstantní ukazatel na nekonstantní paměť. Tento mechanismus se nejvíce projevuje při konstantním referencování dat:

```
1  // D
2  class C {
3    int *x;
4  }
5
6  void main() {
7    C c = new C;
8    c.x = new int( 5 );
9
10   const C c2 = c;
11   *c2.x = 6; // Error: cannot modify const expression *c2.x
12 }
```

(2.2)

Polemika nad tímto tématem je mimo rámec tohoto projektu; zdá se, že zatím ani nebyla veřejně vedena, nicméně střípky informací můžete nalézt na [16] a [14]. Vynucená tranzitivní konstantnost se zdá být zbytečně limitující a bez zřejmých přínosů. V jazyce Beast nebude; můžeme mít konstantní ukazatel na nekonstantní data.

Chceme-li mít nekonstantní ukazatel na nekonstantní data, musíme tedy specifikovat mutabilitu dvakrát. Syntaktický zápis pro takovou referenci by vypadal takto (syntaxe pro referenci je `Typ?`, viz oddíl 2.1.7; příklady reflektují alternativy syntaktického zápisu referencí, které jsou uvedené v oddílu 2.1.4):

1. `@mut ( @mut Typ )?`
2. `Typ!?!`

Další možností by bylo se inspirovat gramatikou C++ (po vzoru **`const int * const`** by měl zápis formu `@mutable Int ? @mutable`), nicméně ta je všeobecně považována za velice matoucí – dochází tam ke kombinování suffixových a prefixových modifikátorů podle neintuitivních pravidel. Čistě suffixový zápis má jasná a jednoduchá a značně snižuje potřebu využívat závorky.

[4] Viz specifikace jazyka Beast (v příloze), oddíl *Decorators*
[5] Viz specifikace jazyka Beast (v příloze), oddíl *Decoration contexts*

**Srovnání syntaxe C++, D a Beast**

```
// C++
int a, b; // Mutable integer
const int c, d; // Const integer
int *e, *f; // Mutable pointer to mutable integer
const int *g, *h; // Mutable pointer to const integer
int * const i, * const j; // Const pointer to mutable integer
const int * const k, * const l; // Const pointer to const integer
```

(2.3)

```
// D
int a, b; // Mutable integer
const int c, d; // Const integer
int* e, f; // Mutable pointer to mutable integer
const( int )* g, h; // Mutable pointer to const integer
// const pointer to mutable integer not possible
const int* k, l; // Const pointer to const integer
```

(2.4)

```
// Beast
Int! a, b; // Mutable integer
Int c, d; // Const integer
Int!?! e, f; // Mutable reference to mutable integer
Int?! g, h; // Mutable reference to const integer
Int!? i, j; // Const reference to mutable integer
Int? k, l; // Const reference to const integer
```

(2.5)

### 2.1.6 Typová kontrola, konverze typů a odvozování typů

Beast má podobný mechanismus typových kontrol jako jazyky C++ a D:

- Typy mohou být implicitně konvertibilní do jiných typů. Proces konverze je definován v jednom z typů (přetížením funkcí `#implicitCastTo` nebo `#implicitCastFrom`)[6]. Implicitní konverze se provádí sama v případě potřeby[7].
- Typy mohou být explicitně konvertibilní do jiných typů (funkce `#explicitCastTo` a `#explicitCastFrom`). K explicitnímu přetypování se používá výhradně funkce `var.to(Type)` (lze použít i k implicitnímu přetypování).

Důvodem k jiné syntaxi oproti C++ je opět snaha o zjednodušení pravidel a zvýšení přehlednosti kódu.

```
// C++
const int x = *( ( const int* )( voidPtr ) );
```

[6] Viz specifikace jazyka Beast (v příloze), oddíl *Type casting*

[7] Implicitní konverze ovlivňuje rezoluci volání přetížených funkcí, viz specifikace jazyka Beast (v příloze), oddíl *Overload resolution*

(2.6)

```
1 // Beast
2 Int x = voidRef.to( Int? );
```

(2.7)

Problémem souvisejícím s tímto tématem je i odvozování typů pro výrazy, u kterých se může typ měnit na základě kontextu. Takovým případem jsou například lambda výrazy v jazyce D. V něm je platný například tento kód:

```
1 // D
2 int function( int ) func = x => x * 2;
```

(2.8)

Kód `x => x * 2` je validní syntaktická konstrukce, která definuje funkci, která přijímá jeden parametr a navrací dvojnásobek jeho hodnoty. Tato konstrukce však samotná neobsahuje dostatečná data pro definici funkce – ta se dedukují až na základě kontextu, v tomto případě podle toho, že se danou lambda funkci pokoušíme uložit do ukazatele na funkci s konkrétním prototypem.

Při sémantické analýze tedy kompilátor musí při zpracovávání výrazu brát v potaz i to, jaký typ je u výrazu očekáván. Tento fakt může způsobit, že se jeden výraz musí analyzovat vícekrát, například v tomto případě:

```
 1 // D
 2 import std.stdio;
 3
 4 void foo( int function(int) func ) {
 5   writeln( "INT RESULT: ", func( 3 ) );
 6 }
 7
 8 void foo( string function(int) func ) {
 9   writeln( "STRING RESULT: ", func( 3 ) );
10 }
11
12 void main() {
13   foo( x => x + 3 );
14 }
```

(2.9)

Zde se sémantická analýza výrazu `x => x + 3` musí provést pro každé přetížení funkce `foo` zvlášť. V případě funkce na řádku 8 sémantická analýza selže, protože výraz `x + 3`, kde `x` je typu **`int`**, je také typu **`int`**, který není implicitně konvertibilní na typ **`string`**.

Typové odvozování se používá i v dalších případech, například u literálů polí, `auto` deklaracích apod. Jazyk Beast cílí podporovat všechny tyto případy, v rámci této práce však z časových důvodů nemusí být do demonstračního kompilátoru zavedeny.

V souvislosti s odvozováním typů zavádí jazyk Beast novou konstrukci `:identifier`, viz oddíl 2.2.1.

### 2.1.7 Reference, ukazatelé a jejich syntaxe

Typ ukazatel umožňující ukazatelovou aritmetiku je v dnešní době považován za potenciálně nebezpečný prvek, který by se měl užívat jen v nutných případech. Toto se řeší zavedením

referencí, které v různých jazycích fungují mírně rozdílně, všeobecně se ale dá říci, že se jedná o ukazatele, které nepodporují ukazatelovou aritmetiku.

C++ reference neumožňují měnit adresu odkazované paměti. V praxi ale ji programátor poměrně často potřebuje měnit; musí se proto uchylovat k ukazatelům, které podporují ukazatelovou aritmetiku a pro přístup k odkazované hodnotě je třeba buď použít dereferenci (která je implementována prefixovým operátorem `*ptr` a má tendenci znepřehledňovat kód) nebo speciální syntaktickou konstrukci (místo `x.y` `x->y`) pro přístup k prvkům odkazované hodnoty.

D k problematice přistupuje takto:

- Třídy jsou vždy předávané odkazem; adresa odkazované instance třídy se dá měnit, ukazatelová aritmetika není podporována (jako v Javě), přetěžování operátoru `a = b` není povoleno pro třídy jako levé operandy. Kromě tříd existují i struktury, které jsou předávané hodnotou; ty však nepodporují dědičnost.
- Existuje typ ukazatel, který pracuje s ukazatelovou aritmetikou. Pro přístup k odkazovanému prvku je třeba použít dereferenci (`*x`); k prvkům odkazované hodnoty lze použít klasické `x.y` (u C++ je třeba `x->y`). U dvojitého ukazatele je již potřeba použít dereferenci.
- Existuje i reference podobná té v C++, nicméně ta se dá použít pouze v několika málo případech (například v parametrech a návratových typech funkcí)

Vynucené předávání instancí tříd odkazem má ale několik nevýhod:

- Programátor musí zajišťovat konstrukci (případně i destrukci) objektů. V jazyce D se automatická konstrukce zajišťuje automaticky těžko (jde to, ale jen s určitými kompromisy).
- Způsobuje více alokací a dealokací.
- Dynamická alokace tříd, které jsou prvky jiných tříd/struktur narušuje lokalitu dat, což vede ke zvýšeným *cache misses* a zpomalení aplikace.

### Řešení v jazyce Beast

Beast má umožňovat nízkoúrovňové programování, typ ukazatel s ukazatelovou aritmetikou je tedy nutné zavést. Na druhou stranu je ale vhodné nabídnout alternativu pro „běžné“ případy užití, které se většinou týkají vytváření instancí tříd na haldě a manipulace s nimi. Proto jazyk standardně nabízí dva ukazatelové typy, které jsou pojmenovány ukazatel a reference.

Způsob deklarace ukazatelů je v C++ i D problematický z hlediska syntaktické analýzy. Výraz `a * b` může totiž znamenat buď výraz násobení `a` krát `b`, stejně tak ale může znamenat deklaraci proměnné `b` typu ukazatel na `a` (obdobně i u reference). Nejjednodušším řešením je použití jiného znaku pro označování ukazatelů.

V jazyce Beast je tímto znakem otazník (`?`). V C++ a D je používán pouze v ternárním operátoru (`cond ? expr1 : expr2`), kteréhož funkčnost Beast obstarává jiným způsobem[8].

Ukazatel v jazyce Beast funguje skoro stejně jako v C++; podporuje ukazatelovou aritmetiku. Na dereferenci se ale nepoužívá prefixový operátor `*ptr` kvůli gramatickým konfliktům a protože může způsobovat nepřehlednost kódu a nejasnosti v prioritě operátorů

[8] Viz oddíl 2.2.4

(například u výrazu `*p++`). Místo toho se používá `ptr.data`; přístup k prvkům odkazované hodnoty není možný jiným způsobem. K získání ukazatele se také, kvůli stejným důvodům jako u dereference, nepoužívá prefixový operátor `&variable`, nýbrž `variable.addr`. Typ ukazatel není deklarován pomocí zvláštní syntaktické konstrukce (suffixový operátor `Typ?` přenechává hojněji používaným referencím), jedná se o kompilátorem definovanou třídu `Pointer( @ctime Type referencedType )`.

Reference je podobná referencím v C++, navíc ale umožňuje změnu odkazované adresy (může mít i hodnotu `null`). Neumožňuje zanořování – nelze definovat referenci na referenci (ukazatel na referenci ale možný je). Chová se stejně jako odkazovaná hodnota, až na několik výjimek:

- Je přetížen operátor `ref := var` a `ref := null`, který je určen pro změnu odkazované adresy reference.
- Je přetížen operátor `ref is null`, který navrací, zda má reference hodnotu `null`.
- `ref.addr` vrací ukazatel na referenci. `ref.refAddr` vrací ukazatel na odkazovaný objekt.

Všechny typy jsou implicitně konvertibilní na referenci (daného typu i jeho předků). Reference jsou implicitně konvertibilní na odkazovaný typ, na reference předků odkazované třídy a explicitně konvertibilní na referenci jakéhokoli typu; při těchto explicitních konverzích probíhá dynamická typová kontrola, jejíž výsledek může být `null`.

Reference v jazyce Beast pokrývají naprostou většinu případů užití ukazatelů při programování na vyšších úrovních abstrakce, a to se syntaxí takovou, že programátor nemusí rozlišovat referenci od normální proměnné. K programování na nižší úrovni se dá použít konvenční typ ukazatel `Pointer( T )`.

### 2.1.8 Dekorátory

Pro tento koncept mívají jazyky, které ho zavádějí, různé pojmenování. V Javě se používá pojem „anotace,“ v Pythonu „dekorátory,“ v D „uživatelem definované atributy,“ v C# a Rustu „atributy.“ Všechny zmíněné jazyky používají podobnou syntaktickou konstrukci, která se umísťuje před deklarace proměnných, tříd, typů, atp. Jejich potenciál se jazyk od jazyku liší; souhrnně se ale dá říci, že programátorům umožňují některé z těchto možností:

- Přiřazování metadat k symbolům; tato metadata pak jazyk umožňuje v době kompilace, případně za běhu, v rámci reflexe získávat.
- Měnit chování symbolů; například nahrazení funkce jinou (syntaktická podpora pro návrhový vzor dekorátor [7]).
- Direktivy kompilátoru – například `@Override` v Javě.

Jazyk Beast zamýšlí poskytnout všechny výše uvedené případy užití. Syntaxi dekorátorů převezme z D/Javy (`@identifier`, případně `@identifier(args)`, které se umísťují před dekorovanou deklaraci/příkaz/konstrukci). Pro koherenci jazyka budou dekorátory využity na všechny dostupné direktivy a modifikátory pro kompilátor; například jazyky Java a D používají pro některé direktivy dekorátory (například `@Override`) a pro jiné klíčová slova (například **`public`**).

Z časových důvodů bude v rámci tohoto projektu implementováno užití dekorátorů pouze jako direktiv pro kompilátor. Pro více informací o dekorátorech v jazyce Beast viz specifikace jazyka Beast (v příloze), oddíl *Decorators*.

### 2.1.9 Vykonávání fukncí za doby kompilace

Vykonávání funkcí za doby kompilace je jedna z nutných prekvizit pro realizaci konceptu líhnutí kódu, viz kapitola 3.

### 2.1.10 Metaprogramování

Metaprogramování je koncept, jehož realizace dává programátorovi možnost přistupovat ke kódu „jako k datům," tedy kód pomocí kódu samotného určitým způsobem číst (například získání seznamu proměnných ve třídě, zjištění návratového typu funkce, atp.) a přidávat/upravovat (například vytvořit třídy pro binární strom nad zadaným typem nebo na základě seznamu proměnných ve třídě vygenerovat funkci, která zajistí její korektní serializaci). Do tohoto konceptu spadá například šablonování a reflexe.

Současné kompilované jazyky realizují metaprogramování jen v omezené míře. Jazyk Beast zavádí koncept líhnutí kódu (viz kapitola 3), který v tomto směru nabízí mnohem více možností.

**Deklarace s `if` v jazyce D** V rámci šablonování nabízí jazyk D velice praktickou konstrukci, kdy se mezi hlavičku a tělo deklarace (třídy, funkce, mixinu, ...) vloží `if( expr )`, kde `expr` je logický výraz, který se vyhodnocuje za doby kompilace. Tento výraz může mimo jiné obsahovat šablonové argumenty deklarace a pokud je vyhodnocen jako `false`, daná deklarace je pro zadané šablonové argumenty považována za neplatnou a kompilátorem ignorována[9].

Níže uvedená funkce v jazyce D

```
1 // D
2 void writeToBuffer( T )( Buffer buf, const ref T data )
3   if( isScalarType!T )
4 {
5   buf.write( &data, T.sizeof )
6 }
```

(2.10)

je například platná jen pro skalární typy. Tomuto kódu by odpovídal následující C++ kód až na ono omezení, které typy šablona přijímá – taková kontrola v mnoha případech v C++ není realizovatelná vůbec a v jiných jen pomocí různých „triků", které činí kód nepřehledným.

```
1 // C++
2 template< typename T >
3 void writeToBuffer( Buffer &buf, const T& data ) {
4   buf.write( &data, sizeof( T ) );
5 }
```

(2.11)

Jazyk Beast tuto konstrukci přejímá z jazyka D beze změn.

[9] Bližší dokumentace o této konstrukci v jazyce D je na stránce https://dlang.org/concepts.html.

### 2.1.11 Mixins

Dalším existujícím konceptem v programovacích jazycích jsou tzv. *mixins*. V podstatě se jedná o prostředek ke „slepému“ vkládání/kopírování kódu. Jako příklad poslouží několik kódu z jazyka D, který tento koncept implementuje[10].

```d
// D
mixin template MyMixin {
  int a;
  void setA( int val ) {
    a = val;
    onAChanged();
  }
}

class C {
public:
  int b;
  mixin MyMixin;
  void onAChanged() {
    writeln( "C.a changed" );
  }
}

class D {
public:
  int c;
  mixin MyMixin;
  void onAChanged() {
    writeln( "D.a changed" );
  }
}

void main() {
  import std.stdio;

  C c = new C;
  D d = new D;

  c.setA( 5 );
  writeln( c.a );

  d.setA( 9 );
  writeln( d.a );
}
```

(2.12)

Ve výše uvedeném příkladu jsme vytvořili `mixin template MyMixin`, který jsme následně vložili do tříd `C` a `D`. Jak bylo uvedeno dříve, *mixins* jsou velice podobné kopírování textu a kompilátor se chová téměř stejně, jako kdybychom obsah `MyMixin` přímo vložili do daných tříd. Tato vlastnost umožňuje do jisté míry suplovat vícenásobné dědění, které jazyk D nepodporuje (místo toho nabízí *interfaces*, podobně jako Java), nicméně program ztrácí na přehlednosti a bezpečnost kódu je nižší. Delší rozvedení problematiky je mimo rámec této práce.

[10] Bližší dokumentace *mixins* v jazyce D je na stránkách https://dlang.org/mixin.html a https://dlang.org/spec/template-mixin.html.

Jazyk D nabízí ještě jednu formu *mixins* – jedná se o konstrukt **mixin**( code ), kde code je výraz, který je za doby kompilace vyhodnocen jako řetězec. Obsah tohoto řetězce je pak vložen na místo konstruktu jako kód.

```
1  // D
2  string generateSampleCode( string operator ) {
3    return "a " ~ operator ~ " 4;";
4  }
5
6  void main() {
7    int a = 3;
8    mixin( generateSampleCode( "+=" ) ); // Becomes "a += 4";
9    // a == 7
10 }
```

(2.13)

Tato forma *mixin*u společně s vykonáváním funkcí za doby kompilace (CTFE) značně zvyšuje možnosti metaprogramování. V praxi ale bývá kód, který *mixins* využívá, nepřehledný.

Ačkoli jazyk Beast podporuje vícenásobnou dědičnost, koncept *mixins* nabízí rozšíření možností a proto jejich zavedení stojí za úvahu. Z časových důvodů je ale prozatím přeskočen.

### 2.1.12 Výjimky

Systém výjimek má napříč jazyky velice podobnou formu využívající konstrukce typu **try-catch-finally** a příkazu typu **throw**. Koncept je všeobecně známý, není tedy třeba ho tu rozepisovat.

Přestože je systém výjimek takřka nezbytnou součástí moderních programovacích jazyků, tato práce se zaměřuje spíše na implementaci nového konceptu líhnutí kódu (viz kapitola 3) a další inovativní koncepty. Výjimky nejsou triviální na implementaci; jejich absence nenarušuje syntaktickou celistvost jazyka a pro demonstrační kompilátor nebude příliš velikým omezením funkčnosti.

### 2.1.13 Lambda výrazy

Lambda výrazy, také nazývané anonymní funkce, jsou jazyková konstrukce umožňující definovat funkce, které nemají identifikátor. Aby tento koncept fungoval, funkce v daném jazyce musí být tzv. *first-class citizens* – musí s nimi být možné nakládat jako s hodnotami proměnných, tedy ukládat je do proměnných, předávat je jako argumenty nebo je navracet při volání funkcí (to je dnes naprosto běžné).

Mnoho moderních programovacích jazyků lambda výrazy podporuje, mezi nimi i naše vzorové jazyky C++ a D. Realizace základních lambda výrazů je triviální (pomocí ukazatele na funkci), návrh se ale komplikuje, pokud chceme v anonymní funkci používat nestatické proměnné z rámce, ve kterém byla funkce definována. Bližší popis lambda výrazů je mimo rámec této práce. Specifikace jazyka Beast lambda výrazy zahrnuje, z časových důvodů ale v demonstračním kompilátoru zahrnuty nejsou.

### 2.1.14 Rysy (traits)

Koncept *traits* (rysy) se napříč jazyky různí jak syntaxí, tak funkčností. V některých jazycích nemá syntaktickou podporu a je spíše používán jako návrhový vzor. Všeobecně se jedná o způsob přidávání funkčnosti třídám a zároveň standardizace této funkčnosti mezi třídami (třída, která implementuje daný rys, musí poskytovat jeho funkčnost) mimo konvenční systém dědičnosti. Uveďme si několik různých přístupů k tomuto konceptu:

- Jazyk Rust[11] na konceptu *traits* zakládá celé objektově orientované paradigma. V Rustu neexistují třídy; existují pouze struktury (*struct*), které mohou obsahovat pouze proměnné (tedy žádné funkce), *traits*, které obsahují prototypy funkcí, a implementace *traits* pro dané struktury. Definice struktur, *traits* a implementací mohou být v různých souborech.

```rust
// Rust
struct Circle {
  x: f64,
  y: f64,
  radius: f64,
}

trait HasArea {
  fn area(&self) -> f64;
}

impl HasArea for Circle {
  fn area(&self) -> f64 {
    std::f64::consts::PI * (self.radius * self.radius)
  }
}
```

(2.14) Příklad *traits* v jazyce Rust, převzato z https://doc.rust-lang.org/book/traits.html

- Jazyk Scala[12] využívá rysy k napodobení vícenásobné dědičnosti (kterou jazyk nepodporuje). *Traits* zde rozšiřují koncept rozhraní (*interfaces*) o možnost implicitní implementace funkcí a přidání proměnných (které jsou ale „abstraktní“ – v rysu jsou obsaženy pouze jako reference a musí být definovány ve třídě, která rys zavádí).

```scala
trait Person {
  var name: String
  var gender: Gender
  def sendEmail(subject: String, body: String): Unit
}

class Student extends Person {
  var name: String // "Implementation" of name
  var gender: Gender
  def sendEmail(subject: String, body: String): Unit = {
    // ...
  }
}
```

[11] Více informací o *traits* v jazyku Rust na adrese https://doc.rust-lang.org/book/traits.html

[12] Více informací o *traits* v jazyce Scala na adrese http://joelabrahamsson.com/learning-scala-part-seven-traits/

(2.15) Příklad *traits* v jazyce Scala, převzato
z https://en.wikibooks.org/wiki/Scala/Traits

- Jazyk C++ nemá přímou syntaktickou podporu pro *traits*, ale díky šablonám se dají použít jako návrhový vzor.

```
1  // ---------------- Area.h
2  template< typename T >
3  struct hasTrait_area { static const bool value = false; };
4
5  template<typename T>
6  double area_calculateArea( const T &t ) {
7    throw;
8  }
9
10 // ----------------  Circle.h
11 class Circle {
12   public: double x = 0, y = 0, radius = 1;
13 };
14
15 // Definition of trait 'area' for class Circle, possibly in another
     module
16 template<>
17 struct hasTrait_area<Circle> { static const bool value = true; };
18
19 template<>
20 double area_calculateArea<Circle>( const Circle &c ) {
21   return c.radius * c.radius * PI;
22 }
23
24 // ----------------  main.c
25 #include <Circle.h>
26 #include <iostream>
27
28 template< typename T >
29 void printVariableProperties( const T& var ) {
30   std::cout << "Variable properties:\n";
31
32   if( hasTrait_area<T>::value )
33     std::cout << "  area: " << area_calculateArea( var ) << "\n";
34 }
35
36 void main() {
37   C c;
38   printVariableProperties( c );
39 }
```

(2.16) Aplikace konceptu *traits* v jazyku C++

- Jazyk D nabízí širší možnosti. *Uniform Function Call Syntax*[12] (dále UFCS), která pro statické funkce s alespoň jedním parametrem, které jsou standardně volány pomocí syntaxe `foo( arg1, arg2, arg3 )`, připouští alternativní syntaxi volání `arg1.foo( arg2, arg3 )` (funkce se pak jeví jako třídní), umožňuje využití konceptu *traits* se syntaktickým zápisem, který je bližší standardnímu využití:

```
1  // area.d
2  enum bool hasTrait_area( T ) = __traits( hasMember, T, "hasTrait_area"
     );
```

```
 3
 4 // circle.d
 5 struct Circle {
 6
 7 public:
 8   // Trait 'area' for Circle defined inside the object
 9   enum hasTrait_area = true;
10   double x = 0, y = 0, radius = 1;
11   double area() const {
12     return radius * radius * 3.14159165358979;
13   }
14
15 }
16
17 // rectangle.d
18 struct Rectangle {
19   public double x = 0, y = 0, w = 1, h = 1;
20 }
21
22 // Trait 'area' for Rectangle defined outside the object (possibly
      different module)
23
24 /// 'Override' default hasTrait_area with more specific template
      parameter
25 template hasTrait_area( T : Rectangle ) if( is( T == Rectangle ) ) {
26   enum hasTrait_area = true;
27 }
28 double area( const ref Rectangle r ) {
29   return r.w * r.h;
30 }
31
32
33 // main.d
34 void printVariableProperties( T )( const ref T var ) {
35   import std.stdio;
36   writeln( "Variable properties:\n" );
37
38   static if( hasTrait_area!T )
39     writeln( "  area: ", var.area(), "\n" );
40 }
41
42 void main() {
43   Circle c;
44   Rectangle r;
45   int x;
46
47   printVariableProperties( c );
48   printVariableProperties( r );
49   printVariableProperties( x );
50 }
```

(2.17) Aplikace konceptu *traits* v jazyce D

V jazyce D se dá díky jeho možnostem metaprogramování implementovat například i kontrola, zda daný typ, který o sobě tvrdí, že má zavedený daný rys (`hasTrait_area!T == true`), skutečně nabízí veškerou funkčnost rysem požadovanou.

Jazyk Beast bude podporovat vícenásobnou dědičnost, která obsahuje funkčnost *traits* v takové podobě, jak je zavádí jazyky Rust nebo Scala; *traits* tedy nejsou potřeba.

### 2.1.15 Uniform function call syntax (UFCS)

Jak již bylo zmíněno v předchozím oddíle, *uniform function call syntax* (dále UFCS) je přístup, kdy se volání funkcí ve formátech `arg1.foo( arg2, arg3 )` a `foo( arg1, arg2, arg3 )` považuje za ekvivalentní. Je zavedený například v jazycích D [12] nebo Rust.

Výhoda zavedení UFCS spočívá v tom, že programátorovi je umožněno přidávat funkčnost typům i mimo jejich definice (mimo tělo **`class`** `C { ... }`):

```
1  // D
2  struct Circle {
3    double x = 0, y = 0, radius = 1;
4  }
5
6  double area( const ref Circle c ) {
7    return c.radius * c.radius * PI;
8  }
9
10 void main() {
11   import std.stdio;
12
13   Circle c;
14   writeln( area( c ) );
15   writeln( c.area() );
16 }
```

(2.18) Příklad využití UFCS v jazyce D

Bez zavedení UFCS by kód na řádku 15 nebyl platný, bylo by možné používat pouze variantu na řádku 14. To programátora při použití funkce nutí uvažovat, jestli je definovaná v těle třídy, nebo mimo. Pokud se jedná o funkci, která má s typem jasnou souvislost (jako v našem příkladu), dá se rozdílná syntaxe považovat za nežádoucí. Přesunutí funkce do těla třídy ale nemusí být vždy vhodné řešení, když:

- Funkce pracuje nad typem, jehož definici programátor nemůže upravit – typy definované kompilátorem (`Int`, `String`) nebo knihovnami třetích stran.
- Funkce pracuje nad více typy současně – pro jazyk Beast to zde znamená, že první parameter je typu **`auto`**, například funkce

  ```
  1 Double area( auto? entity )
  2   if( entity.#type in [ Circle, Rectangle ] )
  3 {
  4   // ...
  5 }
  ```

  (2.19)

  je schopná pracovat nad typy `Circle` a `Rectangle`. Přesun funkce do těl tříd by vyžadoval duplikaci stejného kódu (toto by se v dalo řešit pomocí třídní dědičnosti).
- Funkce se používá pouze ve specifických případech; například se nám v některém z modulů hodí funkce `Bool isDividableByTwo(Int x)`, zavádět ji ale všude by bylo nepraktické.

UFCS nicméně povoluje volání funkce dvěma způsoby, což narušuje jednotnost kódu. V úvahu připadá zavést direktivu kompilátoru (formou dekorátoru, viz oddíl 2.1.8), která by říkala, že místo „klasické“ syntaxe využívá funkce k volání syntaxi pro třídní metody.

Pro každou funkci by vždy byla povolena jen jeden ze způsobů volání. Podobně fungují například *extension methods* v C#.

Jazyk Object Pascal řeší tento problém pomocí tzv. *helper objects*. Objekt, nad kterým funkce pracuje, pak není předáván jako první parametr, ale napodobuje se objektový přístup (**self** a implicitní přístup do jmenného prostoru objektu).

```
1  TObjectHelper = class helper for TObject
2    function Test(): String;
3  end;
4
5  function TObjectHelper.Test(): String;
6  begin
7    Result := 'This is a test';
8  end;
9
10 var
11   o: TObject;
12 begin
13   Writeln( o.Test() );
14 end.
```

(2.20)

Tento koncept se podobá *traits* popsaným v předchozím oddílu, je tu však rozdíl: zatímco v *traits* je zavedeno jednotné rozhraní napříč typy a to je potom pro každý typ implementováno, *helper objects* nezavádí žádné jednotné rozhraní, pouze funkčnost.

Jazyk Beast cílí zavést obdobnou funkčnost, z časových důvodů je však její zavedení a určení přesné finální podoby odloženo.

### 2.1.16 Standardní knihovna

Standardní knihovna je velice důležitý prvek jazyka. Standardizace nejpoužívanějších programovacích vzorů, struktur a funkcí přispívá jak ke kompatibilitě knihoven v jazyce napsaných, tak k celkovému komfortu psaní. Vyhotovení dostatečně obsáhlé standardní knihovny je dlouhodobým cílem vývoje.

### 2.1.17 Kompatibilita s jinými jazyky

Pro jazyky všeobecně a pro nové jazyky obzvláště je důležitá možnost využití kódu napsaného v jiném jazyku. Pro kompilované jazyky to znamená zejména poskytnout možnost deklarovat funkce v různých volacích konvencích [6] a s různým *name mangling* [5] a poskytnout typy kompatibilní napříč jazyky. Rozbor tohoto tématu je však mimo rámec této práce.

## 2.2 Nové a netypické koncepty

Jazyk Beast také zavádí několik inovativních prvků; některé z nich jsou zmíněny v této podkapitole.

### 2.2.1 Konstrukce `:ident`

Výčty (*enums*) jsou v programovaní běžně rozšířené a poměrně hojně používané. V jazyce C neměly výčty vlastní jmenný prostor, což vedlo ke kolizím jmen. To se řešilo například

přidáním prefixu před identifikátory položek, čímž nabývaly na velikosti, znepřehledňovaly kód a zpomalovaly jeho psaní. Prefixy, které odpovídaly zkratkám názvů výčtů (například `enum Month -> mJanuary`), zase často kolidovaly s prefixy jiných výčtů (například `mJanuary` a `mRead` z `enum MemoryAccess`). C++ nabízí možnost vytvořit výčet s vlastním jmenným prostorem (`enum class X`), ale to je prakticky ekvivalentní s přidáním prefixu.

Za příklad si vezměme C++ kód na zápis do souboru pomocí standardní knihovny:

```
1 // C++
2 #include <fstream>
3 int main () {
4   std::fstream fs;
5   fs.open( "test.txt", std::fstream::in | std::fstream::out |
      std::fstream::app );
6   fs << " more lorem ipsum";
7   fs.close();
8   return 0;
9 }
```

(2.21) Příklad použití výčtů v jazyce C++, převzato z http://www.cplusplus.com/reference/fstream/fstream/open/, upraveno

Zde na řádku 5 musí programátor napsat prefix `std::fstream::` dokonce třikrát. Přitom funkce `std::fstream::open` přijímá právě jenom hodnoty z tohoto výčtu (sémanticky to zcela zřejmé není, protože C++ standardně nenabízí typ, který by dokázal bezpečně pracovat s prvky výčtu jako s příznaky; jazyk Beast to však bude umožňovat).

Jazyk Beast nabízí elegantní řešení tohoto problému. Zavádí konstrukci `:ident`, která vyhledává zadaný identifikátor ne v aktuálním jmenném prostoru, nýbrž ve jmenném prostoru typu, jehož hodnotu kompilátor očekává. Za předpokladu, že funkce `open` přijímá jako druhý parametr typ `Flags(OpenMode)`, kde `OpenMode` je výčet, a že prvky výčtu jsou z příznakového typu dostupné (toho je docíleno pomocí importu rámců, viz oddíl 2.2.2), může ekvivalentní kód v jazyce Beast vypadat takto:

```
1 Void main() {
2   File( "test.txt", :write | :append ).write( " more lorem ipsum" );
3 }
```

(2.22)

### 2.2.2 Import rámce

Tento prvek jazyk Beast převzal z jazyka D, kde je znám jako **`alias this`**[13]. Typ, v jehož těle je tato konstrukce použita, se pak efektivně stává subtypem typu hodnoty `expr`, která je vyhodnocena výrazem v konstrukci **`alias`** `expr` **`this`** – je do něj implicitně přetypovatelný, má přístup k jeho členům, atp.

```
1 // D
2 struct S
3 {
4   int x;
5   alias x this;
6 }
7
8 int foo(int i) { return i * 2; }
```

[13] Více o **`alias this`** v jazyce D na adrese http://dlang.org/spec/class.html#AliasThis

```
 9
10 void test()
11 {
12   S s;
13   s.x = 7;
14   int i = -s;  // i == -7
15   i = s + 8;   // i == 15
16   i = foo(s);  // implicit conversion to int
17 }
```

(2.23) Příklad konstrukce **`alias this`** v jazyce D, převzato z http://dlang.org/spec/class.html#AliasThis, upraveno

Jazyk Beast tuto konstrukci také cílí zavést, její přesná syntaktická podoba ale ještě není rozhodnutá a zavedení nebude v rámci této práce.

### 2.2.3 Znak `#` v identifikátorech

Jazyk Beast zavádí relativně velké množství kompilátorem definovaných vlastností a členů tříd (funkce pro přetěžování operátorů, konstruktor/destruktor, funkce pro přetypování, prvky poskytující reflexi jazyka, ...). Aby se neomezovala množina identifikátorů, které programátor může použít, všechny identifikátory, které využívá kompilátor nebo které slouží k reflexi jazyka, začínají znakem `#`. Tento prostor identifikátorů je dostupný i programátorovi, nicméně jeho doporučené využití se omezuje na výše uvedené.

```
 1 class C {
 2
 3 @public:
 4   Void #ctor() {
 5     // Constructor
 6   }
 7
 8   Void #operator( Operator.binPlus, C? other ) {
 9     // Overload of a + b
10   }
11
12   Int foo() {
13     foo.#returnType result = 3; // Language reflection example
14     return result;
15   }
16
17 }
```

(2.24)

### 2.2.4 Ternární operátor

Narozdíl od jazyků C++ a D Beast neposkytuje ternární operátor `a ? b : c`. Hlavním důvodem byla potřeba využít lexémy `?` a `:` jinde. Místo toho existuje funkce

```
1 auto select(@lazy Bool cond1, @lazy auto val1, @lazy auto defVal)
```

(2.25)

která má stejnou funkčnost (`cond1` odpovídá parametru `a`, `val1` parametru `b` a `defVal` parametru `c`). Tato funkce je navíc rozšiřitelná o libovolný počet podmínek a odpovídajících hodnot – tedy například

```
auto select(
  @lazy Bool cond1, @lazy auto val1,
  @lazy Bool cond3, @lazy auto val2,
  @lazy Bool cond3, @lazy auto val3,
  @lazy auto defVal
  )
```

(2.26)

Návratová hodnota této funkce odpovídá první hodnotě parametru `valX`, jehož podmínka `condX` je vyhodnocena jako pravda. Vyhodnocování podmínek i výrazů je líné, funkčnost tedy odpovídá zřetězenému použití ternárního operátoru.

Protože pokročilé prvky (líné vyhodnocování výrazů, variadické funkce, **`auto`** deklarace), které tato funkce vyžaduje pro implementaci, nebudou v rámci tohoto projektu implementovány (ačkoli v hotové verzi jazyka se s nimi počítá), ani samotná funkce `select` nebude v demonstračním kompilátoru obsažena.

### 2.2.5 Vnořovatelné komentáře

Na rozdíl od jazyků C++ a Java jazyk Beast umožňuje vnořování víceřádkových komentářů `/* comment */`. Nebyly nalezeny žádné důvody, které by mluvily v neprospěch této vlastnosti, zatímco přínos je zřejmý – víceřádkové komentáře se hojně používají při „zakomentovávání“ kusů kódu (většinou pro testovací účely). Pokud chce programátor „zakomentovat“ kód, který již obsahuje zakomentovaný kód, naráží na problém a musí odstranit zavírací „závorky“ `*/` vnořených komentářů. Při odkomentování vnějšího bloku je situace ještě horší, protože programátor musí přemýšlet, na jaká místa má znovu přidat zavírací „závorky“ vnitřních komentářů.

Jazyk D vnořené komentáře podporuje, ale mají vlastní syntaxi (`/+ comment +/`).

### 2.2.6 Předdefinované typy

Jazyk Beast se snaží přisuzovat co největší sémantiku typům, které v základu nabízí. Z tohoto důvodu odděluje `IntXX` a `BinaryXX` (kde `XX` označuje velikost typu v bitech), kde typy `IntXX` nepodporují bitové operace. Pro bitová pole (příznaky) jazyk nabízí typ `Flags(@ctime Type SourceEnum)`, který umí ideálně pracovat nad konkrétním výčtem; práce s bity by tedy měla být potřeba jen v minimu případů.

Jako náhradu za `size_t` nabízí jazyk bezznaménkové typy `Index` a `Size`. `Index` je zamýšlen k užití pro ukládání pozic prvků a `Size` pro ukládání informace o velikosti (např. seznamů). V souladu s touto sémantikou byla zavedena následující pravidla:

- `Size`, `Index` a `IntXX` jsou mezi sebou explicitně přetypovatelné.
- `Size * IntXX = Size`, obdobně pro `Index`, operátory `- * /` a prohozené operandy.
- `Size + Size = Size`, totéž pro `Index` a operátory `- * /` (indexy však nelze násobit a dělit jiným indexem).
- `Index - Size = Index`, totéž pro prohozené operandy.
- `Size - Index` nelze.
- `Index - Index = Size`

- `Pointer(Void) - Pointer(Void) = Size`
- `Pointer(X) - Pointer(X) = Index` pro ostatní typy ukazatelů
- `null` je implicitně přetypovatelné jak do typu `Index`, tak do `Size` a odpovídá hodnotě `-1`.

### 2.2.7 Konstrukce `while-else`

Tuto konstrukci přejímá Beast z Pythonu[14]. Konstrukce umožňuje za příkaz cyklu (**`while`**, **`for`**, ...) přidat větev **`else`**, která se vykoná, pokud cyklus není přerušen příkazem **`break`** (případně výjimkou, návratem z fuknce, atp.). Toto je užitečné například pokud cyklus implementuje vyhledávání – v případě nálezu se použije příkaz **`break`**, který cyklus přeruší (prvek byl nalezen, další vyhledávání není potřeba) a větev **`else`** se nevykoná. Pokud cyklus prvek nenalezl, větev **`else`** poskytuje alternativní chování. Bez existence této konstrukce by se po cyklu musela použít podmínka, což je výpočetně dražší – s konstrukcí **`else`** je to zadarmo, pouze se upraví adresy skoků.

```
1  Void main() {
2    String[] list = [ "the", "quick", "brown", "fox", "jumped" ];
3    foreach( String item; list ) {
4      if( item == "quick" ) {
5        console.write( "Found!" );
6        break;
7      }
8    } else
9      console.write( "Not found!" );
10 }
```

(2.27)

[14] https://docs.python.org/3/reference/compound_stmts.html#the-while-statement

# Kapitola 3

# Koncept líhnutí kódu (*code hatching*)

Koncept líhnutí kódu je nový koncept vytvořený v rámci návrhu jazyka Beast a spojuje do koherentního celku funkčnost několika již zavedených konceptů – například šablonového metaprogramování, reflexe jazyka, vykonávání funkcí za doby kompilace.

Koncept zavádí jednu jednoduchou myšlenku, ze které pak vyplývá celá řada důsledků. Tou myšlenkou je **zavedení klasifikátoru pro proměnné, jejichž hodnota se dá zjistit** bez nutnosti spouštět program. Tímto klasifikátorem je v jazyce Beast dekorátor `@ctime`.

Vykonávání kódu tak probíhá ve dvou fázích – hodnoty proměnných označených dekorátorem `@ctime` jsou odvozeny již za doby překladu, zbytek je vypočítáván za běhu samotného programu. **Toto se dá připodobnit k líhnutí vajec**, kdy se zárodek vyvíjí za skořápkou, ukryt před světem, a světlo světa spatří až jako vyvinutý jedinec.

## 3.1 Dedukce axiomů

Máme vyřčenou základní myšlenku – hodnoty proměnných označených dekorátorem `@ctime` musíme být schopni odvodit již během kompilace; v následujícím textu tuto myšlenku budeme rozvádět. Začneme jednoduchým příkladem:

```
1| @ctime Int x = 8;
```

(3.1)

Zde je vše jasné. Proměnná je konstantní, takže se nemůže měnit; po celou dobu její existence je její hodnota osm.

### 3.1.1 Datové závislosti

Další jednoduchý příklad:

```
1| @ctime Int x = console.read( Int );
```

(3.2)

Zde je zřejmé, že proměnná `x` nesplňuje naše požadavky. Příkaz `console.readNumber()` čte data z konzole a návratová hodnota této funkce se nedá zjistit bez spuštění programu (mohli bychom požádat o vstup již během kompilace, pro demonstraci konceptu ale toto

nyní neuvažujme). Z příkladu vyplývá, že `@ctime` proměnná nemůže být datově závislá na volání alespoň některých funkcí.

```
1 Int add( Int x, Int y ) {
2   return x + y;
3 }
4
5 Void main() {
6   @ctime Int a = add( 5, 3 );
7 }
```

(3.3)

V tomto příkladě lze hodnotu proměnné `a` určit. `@ctime` proměnné tedy můžou být datově závislé na volání funkcí, ale jen některých.

```
 1 Int foo( Int x ) {
 2   if( x < 3 )
 3     return console.read( Int );
 4
 5   return x + 1;
 6 }
 7
 8 Void main() {
 9   @ctime Int a = foo( 5 );
10   @ctime Int b = foo( 3 );
11 }
```

(3.4)

Tento příklad ukazuje, že to, zda funkci lze použít pro výpočet hodnot `@ctime` proměnných, může záležet na předaných parametrech.

**Funkce tedy k výpočtu hodnot `@ctime` proměnných všeobecně mohou být použity.** To, jestli funkce opravdu lze použít, se zjistí až během vykonávání. Nelze použít žádné funkce, u kterých kompilátor nezná chování nebo ktré závisí na externích datech (soubory, čas, vstup uživatele, ...).

Z tohoto úhlu pohledu nemá funkce smysl označovat dekorátorem `@ctime`; níže v této kapitole je popsáno, že se tento dekorátor na funkce používá, ale s trochu jiným účelem.

Všeobecně se dá říci, že `@ctime` proměnné nemohou být závislé na ne-`@ctime` datech. Hodnoty ne-`@ctime` proměnných mohou sice být odvoditelné bez nutnosti spuštění programu, ale nemusí. Teoreticky by bylo možné klasifikátor vynechat a odvozovat jej pouze z kódu, v praxi by toto však značně zpomalilo kompilaci – kompilátor by musel každou akci s každou proměnnou zkusit při kompilaci vykonat (jak je uvedeno výše, je to jediný možný způsob, jak ověřit, že proměnná může být považována za `@ctime`). Mnoho proměnných by se ukázalo být ne-`@ctime`, tím pádem by bylo časově nákladné ověřování zbytečné. Zavedením klasifikátoru, který `@ctime` proměnné jasně označuje, je za doby kompilace nutné vykonat pouze ty funkce, u kterých se jejich splnitelnost očekává a jejich nesplnitelnost je považována za logickou chybu (což vyvolá chybu kompilátoru).

Předpokládejme tedy, že **hodnoty všech proměnných, které nejsou `@ctime`, se před spuštěním programu nedají odvodit.**

### 3.1.2 Konstantnost

Nyní zauvažujme nad tím, co se stane, když budeme chtít měnit hodnoty `@ctime` proměnných:

```
1 @ctime Int! x = 8;
2 /* code here */
3 x += 2;
4 /* code here */
```

(3.5)

Zde je všechno v pořádku. `@ctime` proměnné tedy nemusí být vždy konstantní.

```
1  @static @ctime Int! x = 5;
2
3  Void foo() {
4    @ctime Int! y = 5;
5    console.write( x, y, '\n' );
6    y += 3;
7    x += 3;
8    console.write( x, y, '\n' );
9  }
10
11 Void foo2() {
12   x += 2;
13 }
14
15 Void main() {
16   while( true ) {
17     if( console.read( Int ) < 2 )
18       foo();
19     else
20       foo2();
21   }
22 }
```

(3.6)

Tady už narážíme na problém. Za doby kompilace nemůžeme zjistit hodnotu proměnné `x`, protože se mění na základě uživatelského vstupu (podle výsledku `console.read(Int)` se totiž volá buď funkce `foo` nebo `foo2`, obě manipulují s proměnnou `x`). Nekonstantní statické proměnné se nedají ohlídat (a vzniká problém s pořadím vykonávání funkcí, které by s nimi manipulovaly), **`@ctime` statické proměnné tedy musí být vždy konstantní**.

### 3.1.3 Podmínky a cykly

Větvení *if-then-else* a cykly sdílejí stejný princip – různé chování na základě hodnoty nějakého výrazu. Pravidla odvozená v tomto oddílu platí pro všechna větvení stejně.

```
1 Void main() {
2   @ctime Int! x = 5;
3
4   while( x < 6 )
5     x += 3;
6
```

```
 7|   Int y = console.read( Int );
 8|   if( y < 2 )
 9|     x += 8;
10| }
```

(3.7)

Z řádku 4 je patrné, že větvení v rámci `@ctime` proměnných je možné. Řádek 8 zase ukazuje, že to není možné vždy. Pokud je větvení datově závislé na výrazu, jehož hodnota se dá odvodit za doby kompilace (dále jen `@ctime` výrazu), mohou být v jeho těle přítomny `@ctime` proměnné. Pro usnadnění práce kompilátoru a zpřehlednění kódu Beast vyžaduje, aby `@ctime` větvení byly označeny dekorátorem (jinak se k nim přistupuje jako k ne-`@ctime`, viz dále).

Problematika je ale trochu složitější:

```
 1| Void main() {
 2|   @ctime Int! y = 6;
 3|
 4|   while( console.read( Int ) < 5 ) {
 5|     @ctime Int! z = 3;
 6|
 7|     console.write( y );
 8|     console.write( z );
 9|
10|     z += 4;
11|     console.write( z + y );
12|
13|     y += 2;
14|   }
15| }
```

(3.8)

Zde máme proměnnou `z`, která je definovaná v těle ne-`@ctime` cyklu. Nicméně takovéto použití proměnné naše požadavky nenarušuje, stejně tak čtení z proměnné `y` na řádku 7. V ne-`@ctime` větveních tedy mohou být použity `@ctime` proměnné a dokonce i definovány nekonstantní `@ctime` proměnné. Pravidlo, které se z tohoto příkladu dá vyvodit, je že **v tělech ne-`@ctime` větvení nelze měnit hodnoty `@ctime` proměnných deklarovaných mimo něj.**

### 3.1.4 Tranzitivita `@ctime`

Je zřejmé, že je-li proměnná `@ctime` programátorem definovaného třídního typu, musí být pro danou proměnnou všechny třídní proměnné toho typu také `@ctime`.

```
1| class C {
2|   @public Int x, y;
3| }
4|
5| Void main() {
6|   @ctime C c;
7|   c.x = 5; // c.x is @ctime
8| }
```

(3.9)

### 3.1.5 Reference a dynamické alokace

Pro praktickou demonstraci tohoto problému je již třeba složitější příklad:

```
1  class BinaryTreeNode {
2
3  @public:
4    String key;
5    Int! value;
6    BinaryTreeNode!?! left, right;
7
8  }
9
10 class BinaryTree {
11
12 @public:
13   BinaryTreeNode!?! root;
14
15 @public:
16   Void insert!( String key, Int value ) { ... }
17
18   /// binaryTree[ "key" ] key lookup
19   Int #operator( Operator.brackets, String key ) { ... }
20
21 }
22
23 BinaryTree sampleTree() {
24   BinaryTree result;
25   result.insert( "beast", 3 );
26   result.insert( "best", 5 );
27
28   result.top.data = 10;
29 }
30
31 @static @ctime BinaryTree tree = sampleTree();
32
33 Void main() {
34   console.write( tree["beast"] ); // the tree is @ctime, so the compiler is
        capable of looking up value in the binary tree at compile time, making
        this code extremely fast when running
35
36   tree.root.value = 7; // tree is const, but tree.root is not
37 }
```

(3.10)

Zde jsme si na řádku 31 definovali `@ctime` proměnnou `tree`. Třída `BinaryTree` potřebuje pro svou správnou funkčnost i bloky dynamicky alokované paměti (listy stromu; v kódu to explicitně uvedeno není, nicméně z jeho sémantiky to vyplývá). Ty pro správnou funkčnost stromu také musí splňovat podmínky `@ctime`. Ačkoli by teoreticky bylo možné mít `@ctime` ukazatel na ne-`@ctime` paměť (za doby kompilace by se vědělo, kam ukazatel odkazuje, ale ne, co v paměti je), výhodnější je udělat `@ctime` tranzitivní i přes ukazatele a reference. Pokud bychom to takto nenastavili, kód příkladu tohoto oddílu by pro fuknčnost vyžadoval zvláštní úpravy, nebo by nemohl fungovat vůbec.

Protože `tree` je statická `@ctime` proměnná, musí být konstantní. Dynamicky alokované bloky (listy stromu) vytvořené při její inicializaci tedy také musí být konstantní. Toto ale neplatí zcela – během inicializace proměnné `tree` (během vykonávání funkce `sampleTree` za doby kompilace) s bloky normálně manipulujeme (řádky 25 a 26). Tyto bloky se tedy stávají

konstantními až *po* dokončení inicializace. Zde vyvstává problém – v naší třídě `BinaryTree` je ukazatel na dynamicky alokovaný blok typu `BinaryTreeItem!?!` – odkazovaná paměť je mutabilní. Řádek 36 je tedy technicky korektní, i když by podle výše uvedených myšlenek neměl být.

Jak tedy zajistit, aby se po inicializaci proměnné `tree` nedalo k jejím listům přistupovat přes mutabilní reference? Tento problém by se dal vyřešit zavedením vynucené tranzitivity konstantnosti referencí a ukazatelů – tedy že při konstantní referenci/ukazateli by se odkazovaná paměť automaticky brala jako také konstantní. V oddílu 2.1.5 jsme ale rozhodli, že jazyk Beast nebude vynucovat propagaci konstantnosti přes reference – programátor si ji musí zajistit sám tam, kde je to potřeba. U binárního stromu je logické, že je-li konstantní strom, měly by jeho listy být také konstantní; v případě programátorem definovaných typů ale nelze zaručit, že je mutabilita korektně ošetřena.

Při kompilaci se konstantní statické proměnné v některých kompilátorech umísťují do tzv. *.text section* ve výsledném binárním souboru. Stránky paměti načtené z těchto sekcí jsou chráněny operačním systémem proti zápisu a pokus o zapsání do nich vyvolává běhovou chybu (*segfault/access violation*). Beast řeší problém obdobným způsobem – při pokusu o zápis do paměti (během kompilace), která byla alokována během inicializace statické `@ctime` (nebo i jen konstantní) proměnné, mimo dobu inicializace zmíněné proměnné vyvolá kompilátor chybu. Dalším prvkem ochrany je zákaz existence mutabilních referencí a ukazatelů na dané bloky mimo rámec inicializované proměnné.

### 3.1.6 Konverze na ne-`@ctime`

Je vhodné objasnit, že dekorátor `@ctime` není modifikátor typu (`typeModifier`[1]), ale modifikátor proměnné/parametru (`parameterModifier`/`variableModifier`). To znamená, že typy proměnných `ref` a `cref` v příkladu 3.11 jsou shodné (`ref.#type == cref.#type == Int!?`). Zákaz přiřazování výrazů závislých na ne-`@ctime` datech do `@ctime` proměnných je řešen jinak než typovou kontrolou.

```
1  Void main() {
2    @ctime Int! x = 3;
3
4    @ctime Int!? cref := x;
5    cref = 9;
6
7    Int!? ref := cref;
8
9    if( console.read( Int ) < 5 )
10     ref := 8; // x = 8 based on console.read!
11
12   @ctime Int y = x;
13 }
```

(3.11)

Výše uvedený příklad neporušuje žádné z pravidel, které jsme si již odvodili, nicméně hodnota proměnné `y` již není odvoditelná během kompilace, protože se hodnota proměnné `x` může změnit na základě uživatelského vstupu. Přiřazení do `cref` na řádku 5 je v pořádku; kontrola nad daty se ztrácí při inicializaci proměnné `ref` na řádku 7. Aby se tomuto

[1] Viz specifikace jazyka Beast (v příloze), oddíl *Decoration contexts*

předešlo, zavedeme pravidlo, že **na `@ctime` data nelze odkazovat ne-`@ctime` referencí/ukazatelem na mutabilní typ**.

Toto pravidlo ale není dostatečné:

```
1  class C {
2    @public Int! x;
3  }
4
5  class D {
6    @public C!?! c = new C;
7  }
8
9  Void main() {
10   @ctime D d;
11
12   D? dref := d;
13   dref.c.x = 5; // Changing @ctime data via non-@ctime reference!
14 }
```

(3.12)

Zde jsme toto pravidlo neporušili, ale přesto se nám opět podařilo porušit axiom pro `@ctime`. Zavedeme tedy další pravidlo: **odkaz na `@ctime` data nelze uložit do ne-`@ctime` reference/ukazatele, pokud všechny třídní proměnné typu reference/ukazatel v odkazovaném typu nejsou konstantní**.

Toto pravidlo se ale také dá obejít:

```
1  class C {
2    @public Int x;
3  }
4
5  class C2 : C {
6    @public Int!? y = new Int;
7  }
8
9  class D {
10   @public C? c = new C2;
11 }
12
13 Void main() {
14   @ctime D d;
15
16   D? dref := d;
17   dref.c.to( C2? ).y = 5;
18 }
```

(3.13)

Řešení tohoto problému už je značně problematické. Kvůli způsobu implementace by zápis do paměti připadající `@ctime` datům vedl k nedefinovanému chování aplikace, takže je třeba mu zabránit. Všem těmto komplikacím by se předešlo vynucením tranzitivity konstantnosti přes referenci a vzniká tak dilema, zda tranzitivitu vynutit i za cenu omezení možností programátora, nebo ponechat programátorovi místo, kde by se nechtěně mohl „střelil do nohy“. Na základě názoru, že „jazyk má sloužit programátorovi, ne programátor jazyku“ se autor přiklání k variantě nevynucovat tranzitivitu.

### 3.1.7 Třídní `@ctime` proměnné

Zvažme možnost existence nestatických `@ctime` proměnných ve třídách:

```
1  class C {
2
3  @public:
4    @ctime Int! y = 0;
5
6  @public:
7    Void foo!() {
8      y += 3;
9    }
10
11 }
12
13 Void main() {
14   C! c, c2;
15
16   C!?! cref = select( console.read( Int ) < 4, c, c2 );
17   cref.foo();
18 }
```

(3.14)

Z příkladu lze odvodit, že myšlenka není uskutečnitelná. Volání funkce `foo` je datově závislé na uživatelském vstup a nedá se tomu zabránit jinak, než označením proměnných `c` a `c2` jako `@ctime` (což by znemožnilo použití `console.read`). I kdybychom zavedli „`@ctime`“ funkce, které jako parametry přijímají pouze `@ctime` data, k ničemu by nám to nebylo. Toto se dá chápat i tak, že funkce `foo` obsahuje skrytý parametr – odkaz na instanci třídy. Protože proměnná `cref` není `@ctime`, právě tento parametr by porušoval axiom `@ctime`.

## 3.2 Přehled odvozených pravidel

1. `@ctime` proměnné nemohou být jakkoli (datově) závislé na proměnných, které nejsou `@ctime`.
2. Statické `@ctime` proměnné musí být konstantní.
3. Větvení může být `@ctime`, pokud je řídící výraz `@ctime`.
   (a) `@ctime` větvení se musí dekorovat.
   (b) Těla `@ctime` větvení nejsou nijak omezena (nejsou `@ctime`).
   (c) V tělech ne-`@ctime` větvení se nemůže měnit data `@ctime` proměnných, které byly deklarovány mimo něj (ale lze je číst).
4. Je-li proměnná `@ctime`, jsou všechny její třídní proměnné také `@ctime`.
5. Je-li reference nebo ukazatel `@ctime`, paměť, na kterou odkazuje, je také `@ctime` (`@ctime` je tranzitivní přes odkaz).
6. Po inicializaci statické `@ctime` proměnné nelze zapisovat do paměti, která byla během její inicializace (dynamicky) alokována.
7. Ukládání `@ctime` proměnných do ne-`@ctime` proměnných má další omezení:

(a) Na `@ctime` data nelze odkazovat ne-`@ctime` referencí/ukazatelem na mutabilní typ.

(b) Odkaz na `@ctime` data nelze uložit do ne-`@ctime` reference/ukazatele, pokud všechny třídní proměnné typu reference/ukazatel v odkazovaném typu nejsou konstantní.

8. Třída (ne-`@ctime`) nemůže obsahovat nestatické `@ctime` proměnné.

V dalších oddílech jsou odvozena další dodatečná pravidla. Jejich přehled je uveden v podkapitole 3.5.

## 3.3 Potenciál a důsledky konceptu

Když už máme odvozená pravidla, můžeme si ukázat, co všechno nám koncept líhnutí kódu umožňuje.

### 3.3.1 Optimalizace kódu

Pokud kompilátor zná hodnoty `@ctime` proměnných již za doby kompilace, logicky může výpočet jejich hodnot vynechat při běhu. Kód

```
1 Void main() {
2   @ctime Bignum myPrime = nthPrimeNumber( 1286345 );
3   console.write( myPrime );
4 }
```

(3.15)

se vyoptimalizuje do

```
1 Void main() {
2   console.write( 20264747 );
3 }
```

(3.16)

Toto samozřejmě umí hodně moderních kompilátorů; dekorátor `@ctime` ale dává programátorovi pevnější kontrolu nad optimalizacemi.

### 3.3.2 Typové proměnné

Díky tomu, že kompilátor zná hodnotu `@ctime` proměnné v každém bodě již během kompilace, může měnit chování `@ctime` proměnné na základě její hodnoty. To umožňuje zavedení mutabilních typových proměnných. Všechny třídy jsou v jazyce Beast instancemi třídy `Type` (i samotná třída `Type` je svou vlastní instancí). Identifikátory tříd jsou ekvivalentní konstantním typovým proměnným.

```
1 Void main() {
2   @ctime Type T = Int16;
3
4   @ctime if( VERSION > 15 )
5     T = Int32;
```

```
6
7     T x = 4;
8   }
```

(3.17)

```
1   class C {
2     @static Int! x;
3   }
4
5   class D {
6     @static Int! y;
7   }
8
9   void Main() {
10    @ctime Type T = C;
11    T.x = 5;
12
13    T = D;
14    D.y = 8;
15  }
```

(3.18)

Typová proměnná může existovat pouze jako `@ctime`.

### 3.3.3 Šablonování a unifikace šablonových a klasických parametrů

Koncept `@ctime` se dá jednoduše využít i pro parametry šablon.

```
1   class TreeNode( @ctime Type Key, @ctime Type Value, @ctime Int childrenCount
       ) {
2
3   @public:
4     Key! key;
5     Value! value;
6     This?!?[ childrenCount ] children;
7
8   }
```

(3.19)

Díky prakticky nulovému rozdílu v syntaxi `@ctime` a ne-`@ctime` proměnných lze šablonové a standardní parametry funkcí zapisovat do stejných závorek a libovolně je míchat:

```
1   T readFromBuffer( Buffer!? buf, @ctime Type T ) {
2     T? result = buf.ptr.to( T? );
3     buf.ptr += T.#size;
4     return result;
5   }
6
7   Void main() {
8     Buffer! buf;
9     buf.readFromFile( File( "in.txt", :read ) );
10    Int i = readFromBuffer( buf, Int );
11  }
```

(3.20)

Dokonce lze vytvořit i speciální dekorátor, v našem jazce pojmenovaný `@autoCtime`, který argument bere jako `@ctime`, pokud to je možné:

```
Index regexFind( String pattern, @autoCtime String regex ) {
  // regex implementation
}

Void main() {
  Index r1 = regexFind( console.read( String ), "Y[a-z]+l" ); // Regex
     argument is known at compile time, so the regex precprocessing is done
     at compile time

  Index r2 = regexFind( console.read( String ), console.read( String ) ); //
     No optimizations here
}
```

(3.21)

Pomocí klíčového slova **`auto`** může funkce přijmout hodnotu jakéhokoli typu

```
// C++
template< typename T1, typename T2 >
inline auto max( T1 a, T2 b ) {
  return ( a  > b ) ? a : b;
}
```

(3.22)

```
// Beast
@inline auto max( auto a, auto b ) = select( a > b, a, b );
```

(3.23)

Jak bylo popsáno v oddílu 2.1.10, jazyk Beast zavádí konstrukci **`if`**`( expr )`, která se vkládá mezi deklarační část a tělo funkce/třídy. Výraz `expr` je vyhodnocen jako `@ctime Bool` a pokud je jeho hodnota `false`, deklarace je ignorována. Výraz v konstrukci může pracovat i s `@ctime` argumenty deklarace.

```
Void test( @ctime Int x )
  if( x < 0 )
{
  console.write( "<0" );
}

Void test( @ctime Int x )
  if( x >= 0 )
{
  console.write( ">=0" );
}

Void main() {
  test( -3 ); // Writes "<0"
  test( 5 ); // Writes ">=0"
}
```

(3.24)

### 3.3.4 `@ctime` funkce a třídy

Pro usnadnění práce programátora se zavádí i podpora dekorátoru `@ctime` pro funkce a třídy.

`@ctime` funkce mají všechny argumenty, návratovou hodnotu, proměnné a příkazy uvnitř automaticky `@ctime` a není tedy třeba všechny prvky takto dekorovat.

Třídní proměnné `@ctime` tříd jsou `@ctime` (funkce mohou být ne-`@ctime`).

### 3.3.5 Reflexe kódu

Koncept také umožňuje reflexi; implicitně se jedná o reflexi za doby kompilace, reflexi za doby běhu si programátor může napsat (případně využít knihovnu). Vezmeme-li v úvahu, že každá třída je instancí `@ctime` třídy `Type`, stačí už jen přidat do těchto tříd další `@ctime` proměnné, které je popisují. Jazyk Beast zavádí `@ctime` konstanty jako `Symbol.#identifier`, `Function.#returnType`, třídy `FunctionMetadata`, `ClassMetadata` a funkce jako `Class.#member( String identifier )`, které reflexi zajišťují. Jsou zdokumentovány v referenci jazyka.

## 3.4 Implementace konceptu

Nyní zvažme, jak by se tento koncept dal implementovat do kompilátoru.

Je zřejmé, že hodnoty `@ctime` proměnných se budou vypočítávat již během kompilace pomocí interpretu zabudovaného do kompilátoru.

Možnosti `@ctime` jsou díky typovým proměnným a reflexi kódu větší, než co je možné udělat za běhu aplikace; v rámci `@ctime` se sémantika příkazů může měnit na základě vykonání jiného `@ctime` kódu. Kdybychom zpracování `@ctime` chtěli realizovat klasickým interpretem, který přijímá bajtkód, zjistili bychom, že takový kód, který by dokázal pojmout veškerou funkčnost konceptu líhnutí kódu, by odpovídal abstraktnímu syntaktickému stromu.

Tento `@ctime` „interpret“ bude nutně pomalejší než konvenční interprety, protože jeho součástí je kompletní sémantická analýza. Jakmile se ale zbavíme `@ctime` proměnných, můžeme program převést do bajtkódu a ten zpracovávat konvenčním interpretem, který je rychlejší. Pro větší rychlost kompilace by tedy kompilátor jazyka Beast měl mít dva interprety: první (dále interpret prvního stupně) pracující přímo nad AST schopný zpracovávat i `@ctime` proměnné a druhý (dále interpret druhého stupně) pracující nad bajtkódem. Tyto interprety se dokonce mohou postupně aplikovat na stejnou funkci:

```
1  Int pow( Int x, @ctime Int exp ) {
2    Int! result = 1;
3
4    @ctime foreach( expTmp; 0 .. exp )
5      result *= x;
6
7    return result;
8  }
9
10 @ctime Int powed = pow( 5, 3 );
```

(3.25)

Výše uvedený kód by po interpretaci funkce `pow( 5, 3 )` pomocí `@ctime` interpretu vypadal (bez optimalizací) takto:

```
1 Int pow( Int x, 3 ) {
2   Int! result = 1;
3   result *= x;
4   result *= x;
5   result *= x;
6   return result;
7 }
```

(3.26)

Po spuštění druhého stupně interpretace (které by předcházela generace bajtkódu) by funkce už navrátila hodnotu 125, která by se uložila do proměnné `powed`. V tomto konkrétním případě by mohlo být vhodnější celou funkci vykonat pomocí pomalejšího `@ctime` interpretu (protože generování bajtkódu a jeho následné vykonání může být ve výsledku pomalejší), nicméně v případech, kdy by se funkce `pow` se stejnou hodnotou druhého parametru (který je `@ctime`) volala vícekrát, je výhodnější použít dvoustupňovou interpretaci.

Bližší studium této problematiky je mimo rámec této práce. V demonstračním kompilátoru je z časových důvodů implementován pouze interpret prvního stupně. I přes faktickou existenci pouze jednoho interpretu zaveďme termíny „interpretace prvního stupně," kterým je myšleno zpracování `@ctime` kódu, a „interpretace druhého stupně," který referuje na vykonávání standardních funkcí za doby kompilace.

### 3.4.1 `@ctime` proměnné a ne-`@ctime` reference

Nelze tvrdit, že `@ctime` proměnné jsou čistě záležitostí kompilace. V momentě, kdy uložíme adresu `@ctime` proměnné do ne-`@ctime` reference/ukazatele (toto může nastat například i při předávání `@ctime` dat jako argumentů typu reference při volání ne-`@ctime` funkcí), tak abychom zajistili funkčnost reference, musíme data této proměnné (a všech dalších proměnných, na které odkazuje) umístit do paměti i při běhu programu. Data referencovaných nestatických `@ctime` proměnných ve funkcích musí být uložena na zásobníku. Ačkoli je veškerý `@ctime` kód vykonáván interpretem prvního stupně, výsledky a dokonce i některé mezistavy musí být kopírovány při běhu aplikace.

Uvažme příklad:

```
1  Void write( Int? val ) {
2    console.write( val );
3  }
4
5  Void main() {
6    @ctime Int! x = 3;
7    write( x );
8    x += factorial( 10 );
9    x += 5;
10   write( x );
11 }
```

(3.27)

Řádky 6, 8 a 9 jsou obsluhovány interpretem prvního stupně a řádky 7 a 10 se vykonávají až za běhu aplikace. Vzhledem k tomu, že parametr funkce `write` je předáván odkazem, musí existovat místo v paměti, které je funkci předáno a které obsahuje aktuální hodnotu proměnné `x`. Hodnota nemusí být aktualizována vykováváním samotného výpočtu – mašinerii pro výpočet faktoriálu lze nahradit jednoduchou *mov* nebo *xor* instrukcí – a nemusí

být aktualizována vždy – například mezi řádky 8 a 9 není žádný ne-`@ctime` kód, který by s daty pracoval a není tedy třeba promítat změny z řádku 8 okamžitě do paměti.

### Paměťové prostory

Je třeba také brát v potaz existenci dvou adresových prostorů – virtuálního, se kterým se pracuje během kompilace, a skutečného, který je zaveden při běhu programu. Interpret v kompilátoru nutně nemusí být implementován s klasickým adresovým prostorem, pro funkčnost ukazatelové aritmetiky v `@ctime` se ale jedná o nejintuitivnější řešení a demonstrační kompilátor je tímto způsobem implementován.

```
1 Void foo() {
2   @ctime Int! x = 3;
3   @ctime Int? ctXRef := x;
4   Int? xRef := x;
5 }
```

(3.28)

Ve výše uvedeném příkladu je proměnná `cXRef` `@ctime`, kompilátor tedy musí vědět, kam odkazuje. To při našem návrhu interpretu znamená, že proměnné `x` musí být přiřazena adresa již za doby kompilace. Nedá se předpokládat, že adresa proměnné `x` bude za běhu stejná jako za doby kompilace. Kromě toho za doby kompilace bude proměnná `x` existovat pouze v jedné instanci – protože funkce `foo` není `@ctime`, hodnoty lokálních `@ctime` proměnných budou mít vždy ve stejném místě v kódu stejnou hodnotu; interpretu prvního stupně stačí vykonat `@ctime` část funkce pouze jednou.

Je tedy třeba zavést mapování z adresového prostoru interpretu prvního stupně do adresového prostoru aplikace, případně interpretu druhého stupně. Toto mapování může mít tři formy:

1. Mapování adres statických dat (pevná adresa)
2. Mapování adres lokálních proměnných (na zásobníku – *base pointer offset*, takto je řešená například proměnná `x` z příkladu 3.28)
3. Mapování adres `@ctime` bloků, které byly dynamicky alokovány při vykonávání ne-`@ctime` funkce

Třetí forma se dá demonstrovat tímto příkladem:

```
1 Void foo() {
2   @ctime Int!?! x := new Int!( 5 );
3   console.write( x );
4   x += 5;
5   console.write( x );
6   delete x;
7 }
```

(3.29)

Zde se při každém volání funkce `foo` dynamicky alokuje proměnná typu `Int`. Protože dynamická alokace je součástí inicializace `@ctime` proměnné, je obsluhována interpretem prvního stupně. Interpret tím pádem musí zajišťovat (generováním vhodných instrukcí) promítání změn proměnné odkazované z `x` do paměti běžící aplikace. V tomto konkrétním případě je jednoduché odvodit, že adresa je uložena v proměnné `x`, vezmeme-li ale v úvahu

vícenásobnou indirekci v `@ctime` datech a optimalizaci promítání změn (takže změny dat jsou promítány v dávkách, až když je to potřeba – kompilátor tedy v době promítání již „neví," přes které odkazy se k danému bloku dat dostal), problém na triviálnosti ztrácí.

Naivním řešením tohoto problému je uchovávat adresy všech bloků, které byly dynamicky alokovány při vykonávání `@ctime` kódu, jako skryté proměnné na zásobníku – to ale může při větším počtu dynamických alokací způsobovat potíže s místem na zásobníku. Zkoumání efektivnějších řešení této problematiky je ale mimo rámec této práce. Demonstrační kompilátor to řeší právě uvedeným naivním způsobem.

Demonstrační kompilátor kvůli mapování adres uchovává informace o tom, na kterých místech paměti interpretu jsou ukazatelé a reference (podle volání konstruktorů a destruktorů).

Aby se skryly rozdíly mezi adresovým prostorem interpretu a adresovým prostorem za běhu a předešlo se tak případným problémům (například by hašovací tabulka v `@ctime` generovala hashe na základě hodnot ukazatelů, ale za běhu by adresy byly jiné a tabulka by tím pádem byla neplatná), Beast dává `@ctime` ukazatelům a referencím následující omezení:

1. Přetypování mezi referenčním a nereferenčním typem není možné (nelze přetypovat ukazatel/referenci na ordinální typ ani naopak).

   Toto se dá obejít přetypováním adresy ukazatele na Pointer( Void ) a pak na ordinální typ pomocí kódu `pointer.addr.to( Pointer( Void ) ).to( Pointer( Index ) ).data`, ale tomu se dá jen těžce zabránit. Takovéto přetypování programátor používá na vlastni nebezpečí.

2. Provnávání (`< > <= >=`, lze `== a !=`) a rozdíl dvou ukazatelů je možný pouze pro ukazatele, které ukazují na stejný blok paměti (v jiných případech kompilátor zobrazí chybu). Toto omezení je kontrolováno interpretem při vykonávání.

### 3.4.2 `@ctime` větvení a sémantická kontrola

Uvažme následující příklad:

```
1  class C {
2    Int! x;
3  }
4
5  class D {
6    Int! y;
7  }
8
9  Void main() {
10   @ctime Type T = C;
11   @ctime T a;
12
13   @ctime if( a.#type == C )
14     a.x = 5;
15   else
16     a.y = 5;
17 }
```

(3.30)

Aby kód správně fungoval, sémantická kontrola v **`else`** větvi podmínky (řádek 16) nesmí proběhnout, protože proměnná `a` nemá člen `y`. Tento případ není okrajovou záležitostí – v reálném `@ctime` kódu by obdobné problémy nastávaly na spoustě míst. **Sémantická**

**kontrola tedy neprobíhá ve větvích `@ctime` větvení, které neproběhnou.** Na druhou stranu ale také musí probíhat pro každou iteraci `@ctime` cyklů:

```
1  class C {
2    Int! x;
3  }
4
5  Void main() {
6    @ctime Type[] arr = [ Int16, Int32, C ];
7    @ctime foreach( Type T; arr ) {
8      T var;
9
10     @ctime if( T == C )
11       var.x = 5;
12     else
13       var = 5;
14   }
15 }
```

(3.31)

### 3.4.3 Šablonování

Následující dva příklady mají stejný význam a podobnou funkčnost:

```
1  // C++
2  template< int i >
3  int foo( int j ) {
4    return i * i * j;
5  }
```

(3.32)

```
1  // Beast
2  Int foo( @ctime Int i, Int j ) {
3    return i * i * j;
4  }
```

(3.33)

`@ctime` proměnné v tomto příkladu nahrazují parametry šablon. Po doplnění hodnoty do parametru `i` (který je `@ctime`, a tedy je třeba znát jeho hodnotu již za doby kompilace; v příkladu níže `i = 5`), kompilátor vygeneruje funkci v podobě:

```
1  Int foo( 5, Int j ) {
2    return 25 * j;
3  }
```

(3.34)

V případě, že se někde jinde v kódu volá funkce `foo` se *stejnými* `@ctime` argumenty, zdánlivě není třeba generovat další funkci, protože by se chovala naprosto stejně; kompilátor může využít již vygenerovaný kód. Stejnými `@ctime` argumenty se myslí bitově shodná data; nelze uvažovat shodu podle operátoru `==`, protože bitově rozdílná data mohou i přes sémantickou shodu vést k rozdílným výsledkům výpočtu (například i když jsou dva binární

stromy shodné, jejich hloubka může být různá a výsledky výpočtů pracujících s hloubkou se nemusí shodovat).

Tvrzení, že bitově stejné `@ctime` argumenty generují stejné funkce, ale není ve všech případech platné. Uvažme tento příklad:

```
1  Int calc( Int x, @ctime Int!? y ) {
2    y ++;
3    return x + y;
4  }
5
6  Void main() {
7    @ctime Int! y = 1;
8    Int f = 0;
9    f += calc( 5, y );
10   f += calc( 5, y );
11 }
```

(3.35)

Ačkoli zde funkce `calc` při volání na řádcích 9 a 10 přijímá bytově stejné `@ctime` argumenty, v prvním případě navrací `x + 2` a ve druhém `x + 3`. Nedá se tedy zaručit, že dvě funkce generované ze stejných `@ctime` argumentů budou totožné, a nejjednodušším způsobem je funkci pokaždé generovat. Na tento problém lze aplikovat řadu optimalizačních algoritmů, jejich rozvedení je ale mimo rámec této práce.

### 3.4.4 Typové proměnné

Typové proměnné lze z formálního hlediska považovat za typické `@ctime` proměnné. Jediným případem, kdy se typová proměnná liší od ostatních `@ctime` proměnných, je konstrukt pro deklaraci proměnné.

### 3.4.5 Možné konflikty referencí, `@ctime` a CTFE

Uvažme následující kód:

```
1  Void foo( Int!? x, @ctime Int!? y ) {
2    @ctime while( y < 2 ) {
3      while( x < 10000 )
4        x++;
5      y++;
6    }
7  }
8
9  Void main() {
10   @ctime Int! a = 0;
11   foo( a, a );
12 }
```

(3.36)

Funkce `foo` je zde na řádku 11 volána za doby kompilace (protože se jí předává `@ctime` proměnná přes nekonstantní referenci). Uvážíme-li, že se funkce může vykonávat ve dvou stupních (v tomto případě to má smysl, protože řádek 3 představuje relativně velký cyklus a interpret druhého stupně je rychlejší), vyvstává tu problém: kód po interpretaci prvního stupně bude vypadat takto:

```
1  Void foo( Int!? x ) {
```

```
2   @ctime while( y < 4 ) {
3     while( x < 10000 )
4       x++;
5     // y = 1
6
7     while( x < 10000 )
8       x++;
9     // y = 2
10
11    while( x < 10000 )
12      x++;
13    // y = 3
14  }
```

(3.37)

Chování funkce neodpovídá tomu, jak by se chovala při jednostupňové interpretaci. Jelikož v demonstračním kompilátoru dvoustupňová interpretace nebude, není třeba tento problém bezprostředně řešit. Řešení problému by ale mohlo vést k zavedení dalších limitujících pravidel, tedy k regresi.

## 3.5 Přehled doplňujících pravidel

Tento přehled pravidel doplňuje ten uvedený v podkapitole 3.2.

1. Je zakázáno přetypovávat `@ctime` ukazatel na jiný než referenční `@ctime` typ a naopak.
2. Nelze porovnávat (`< > <= >=`, lze `== a !=`) ani vypočítat rozdíl `@ctime` ukazatelů, které patří do různých alokačních bloků paměti.
3. Pro nevykonané `@ctime` větve se neprovádí sémantická kontrola. U `@ctime` cyklů se provádí sémantická kontrola pro každou iteraci zvlášť.

# Kapitola 4

# Implementace kompilátoru

Tato kapitola se věnuje implementaci demonstračního kompilátoru jazyka.

**Volba implementačního jazyka**

S ohledem na autorovu znalost programovacích jazyků připadaly jako jazyky pro implementaci demonstračního kompilátoru v úvahu dva kandidáti – C++ a D. Zvítězil jazyk D, zejména kvůli obsáhlé a dobře zdokumentované standardní knihovně a kvůli odpadnutí nutnosti psát hlavičkové soubory. Nevýhodou je menší nabídka podporovaných vývojových prostředí, obzvláště pak s podporou ladění v OS Windows.

## 4.1 Lexikální a syntaktická analýza

Při psaní kompilátoru stojí za to uvažovat o využití generátoru parseru. Existuje mnoho generátorů parserů, jejich průzkum a srovnávání by sám o sobě vydal na knihu; mezi ty nejznámější z nich patří například LALR generátor Bison[1], a LL generátor ANTLR[2]. Srovnání LL a LR parserů je mimo rámec této práce a snadno dohledatelné na internetu v přehledné podobě (například [9]).

Bylo rozhodnuto, že demonstrační kompilátor bude založen na LL parseru, především kvůli větší intuitivnosti návrhu a lepším možnostem generování chybových zpráv. Dalším předmětem k úvaze bylo, zda opravdu použít generátor parseru, nebo parser napsat ručně. Napsání vlastního syntaktického a sémantického analyzátoru není obtížné a nejde o nic neobvyklého – takto jsou řešeny například kompilátory Microsoft C#[3], DMD[4], GCC[5] nebo Clang[6]. Autor tohoto projektu zvolil variantu napsání vlastního parseru a lexeru metodou rekurzivního sestupu (modifikovanou tak, aby si poradila i s částmi gramatiky, které nejde zpracovat LL(1) parserem) ve víře, že časová náročnost bude srovnatelná s osvojením si a aplikací nové technologie s tím, že takto napsaný parser bude lépe modifikovatelný a všeobecně více „pod kontrolou“.

[1] https://www.gnu.org/software/bison/

[2] http://www.antlr.org/

[3] Dle [13], oddíl 8.5.2

[4] Kompilátor jazyka D; zřejmé z kódu parseru, viz https://github.com/dlang/dmd/blob/master/src/parse.d

[5] Uvedeno zde: https://gcc.gnu.org/wiki/New_C_Parser

[6] Uvedeno zde: http://clang.llvm.org/features.html, první odstavec předposledního oddílu

# Literatura


[1] The Java™ Tutorials. 1995.
URL https://docs.oracle.com/javase/tutorial/index.html

[2] Specification for the D Programming Language. 1999.
URL https://dlang.org/spec/spec.html

[3] The Rust Programming Language. 2010.
URL https://doc.rust-lang.org/nightly/book/

[4] C++ reference. 2011.
URL http://en.cppreference.com/w/

[5] Name Mangling. 2016.
URL https://en.wikipedia.org/wiki/Name_mangling

[6] Calling convention. 2017.
URL https://en.wikipedia.org/wiki/Calling_convention

[7] Decorator pattern. 2017.
URL https://en.wikipedia.org/wiki/Decorator_pattern

[8] Const Correctness. c2017.
URL https://isocpp.org/wiki/faq/const-correctness

[9] What are the main advantages and disadvantages of LL and LR parsing? c2017.
URL http://softwareengineering.stackexchange.com/questions/19541

[10] Abrahamsson, J.: Learning Scala. 2010.
URL http://joelabrahamsson.com/programming/scala/

[11] Allain, A.: Const Correctness. c1997-2011.
URL http://www.cprogramming.com/tutorial/const_correctness.html

[12] Bright, W.: Uniform Function Call Syntax. 2002.
URL http://www.drdobbs.com/cpp/uniform-function-call-syntax/232700394

[13] Campbell, B.; Iyer, S.; Akbal-Delibas, B.: *Introduction to compiler construction in a Java world*. Boca Raton, FL: CRC Press, 2013, ISBN 978-143-9860-885.

[14] Grönlund, H.-E.: Functional D: Is Transitive Const Fundamental? c2006-2013.
URL http://www.hans-eric.com/2008/07/30/functional-d-is-transitive-const-fundamental/

[15] Mortoray, M.: Why Garbage Collection is not Necessary and actually Harmful. 2011.
URL https://mortoray.com/2011/03/30/why-garbage-collection-is-not-necessary-and-actually-harmful/

[16] Schveighoffer, S.: Fully transitive const is not necessary. 2008.
URL http://lists.puremagic.com/pipermail/digitalmars-d/2008-April/034868.html

# Přílohy