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 JAZYKA

COMPILER OF NEW MODULAR LANGUAGE

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE **DANIEL ČEJCHAN**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ZBYŇEK 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

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Citace

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

Překladač nového modulárního 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
19. prosince 2016

Poděkování

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 část z mnoha rozhodnutí, která byla vykonána při procesu návrhu programovacího jazyka NATI a vzorového překladače pro něj. Rámcově prozkoumává syntaktické i sémantické prvky moderních programovacích jazyků a popisuje i nové koncepty a prvky.

Programovací jazyk se hodně inspiruje jazyky C++ a D[1]. Tyto jazyky budou používány pro srovnávání syntaxe a efektivity psaní kódu.

1.1 Motivace

Jazyk C++ je stále jedním z nejpoužívanějších jazyků[2].

1.2 Cíle

[1] http://dlang.org/

[2] Podle http://www.tiobe.com/tiobe-index/

Kapitola 2

Vlastnosti a syntaxe jazyka

Nejdříve musíme určit základní rysy jazyka, které potom budeme dále rozvíjet.

Použití jazyka Naším cílem je navrhnout tzv. *general purpose language*, tedy jazyk nezaměřený na konkrétní případy užití. Cílíme vytvořit „nástupce“ jazyka C++, který by se dal použít ve všech případech, kde se dá využít C++ – tedy i třeba na mikroprocesorových systémech. Složitější struktury se tím pádem budeme snažit řešit spíše vhodnou abstrakcí než zaváděním prvků, které kladou zvýšené nároky na výkon a paměť.

Kompilovaný vs. interpretovaný jazyk Ačkoli interpretované jazyky mají jisté výhody, platí se za ně pomalejším kódem a nutností zavádět interpret. V rámci této práce budeme navrhovat kompilovaný jazyk. Pro zjednodušení práce při psaní překladače nebudeme ale překládat přímo do strojového kódu, ale do jazyka C.

Syntaxe jazyka Abychom maximálně usnadnili přechod případných programátorů k našemu jazyku a zkrátili učební křivku, je vhodné se co nejvíce inspirovat již existujícími jazyky. Vzhledem k tomu, že se jedná o kompilovaný jazyk, je nejrozumnější vycházet se syntaxe rodiny jazyků C, které jsou hojně rozšířené a zažité.

Programovací paradigmata Jelikož vycházíme z jazyků C++ a D, přejímáme i jejich paradigmata a základní koncepty. Náš jazyk bude tedy umožňovat strukturované, funkcionální i objektově orientované programování.

2.1 Existující prvky programovacích jazyků

2.1.1 Systém dedičnosti tříd

Jazyky C++ a D mají různě řešené třídní systémy. 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 má systém spíše podobný Javě – třídy (*class*) mohou mít maximálně jednoho rodiče, navíc ale existují rozhraní (*interface*), které však nemohou obsahovat proměnné.

Ačkoli vícenásobné dědění není třeba často, občas potřeba je a jen těžko se nahrazuje. Rozhraní nemohou obsahovat proměnné, což omezuje jejich možnosti. D nabízí ještě jedno

řešení – tzv. *template mixins*[1] – které funguje velice podobně jako kopírování bloků kódu přímo do těla třídy. Toto řešení rozbíjí model dědičnosti (na *mixiny* se nelze odkazovat, nefungují jako rozhraní); navíc, protože jsou vložené funkce prakticky součástí třídy, nefunguje v určitých případech kontrola přepisování (*overridingu*).

Ačkoli je implementace systému vícenásobné dědičnosti tak, jak je například v C++, složitější, její implementace je uskutečnitelná, kód nezpomaluje a fakticky rozšiřuje možnosti jazyka. C++ model dědičnosti umí vše, co umí Javovský model, a ještě víc.

Jazyk NATI tedy bude umožňovat třídní dědičnost, a to způsobem podobným C++. Nicméně implementace systému třídní dědičnosti není primárním cílem projektu, a tak je možné, že v rámci bakalářské práce nebude plně implementován.

Dá se přemýšlet i o zavedení různých direktiv, které umožňují dále nastavovat tento systém – tedy například přidat možnost nastavení zamezení generování tabulky virtuálních metod, atp.

2.1.2 Automatická správa paměti – garbage collector

Spousta moderních jazyků (mezi nimi i D) obsahuje garbage collector (dále GC) v základu. Jeho zavedení nenabízí pouze výhody – programy mohou být pomalejší, GC zvyšuje nároky na CPU a paměť (těžko se zavádí v mikroprocesorech); navíc efektivní implementace GC je velice složitá.

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

2.1.3 Implicitní konstantnost proměnných a propagace konstantnosti

Koncept konstantnosti proměnných byl zaveden jednak jako prvek kontroly při psaní kódu, jednak zvětšil potenciál kompilátorů při optimalizování kódu. Označení proměnné za konstantní ale u jazyku C++ (i D, Java, ...) vyžaduje ale 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.

Některé jazyky (například Rust[2]) přišly s opačným přístupem – všechny proměnné jsou implicitně konstantní a programátor musí použít nějakou syntaktickou konstrukci k tomu, aby to změnil. NATI tento přístup také zavede. V rámci koherence syntaxe jazyka (která je rozvedena v dalších kapitolách tohoto textu), připadají v úvahu dvě možnosti:

1. Vytvoření dekorátoru v kontextu `typeWrapper`[3], 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 zase asociovatelné s mutabilitou.

Při designu NATI byla zvolena druhá možnost, především kvůli „upovídanosti“ kódu a ještě kvůli jednomu důvodu, který je popsán níže.

[1] https://dlang.org/spec/template-mixin.html
[2] https://doc.rust-lang.org/nightly/book/mutability.html
[3] Viz specifikace jazyka NATI (v příloze), oddíl *Decoration contexts*

Propagace konstantnosti 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. Toto je zbytečné limitování; tranzitivní konstantnost se může hodit, ale měla by být volitelná (v NATI má toto opět potenciál pro řešení v dekorátorech).

V NATI tedy nebude vynucená propagace konstantnosti; můžeme mít konstantní ukazatel na nekonstantní data. Chceme-li mít nekonstantní ukazatel na nekonstantní data, musíme tedy specifikovat mutabilitu dvakrát. Ve dříve navržených dvou případech by to tedy vypadalo takto (syntaxe pro referenci je `Typ?`, viz 2.1.5):

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

Jednou z alternativ by bylo zcela využít gramatiku C++, nicméně ta je všeobecně považována za velice matoucí – dochází tam ke kombinování suffixových a prefixových modifikátorů podle neintuitivních pravidel. Čistě suffixový zápis je jasný a jednoduchý a značně snižuje potřebu využívat závorek.

Srovnání syntaxe C++, D a NATI

```
1  // C++
2  int a, b; // Mutable integer
3  const int c, d; // Const integer
4  int *e, *f; // Mutable pointer to mutable integer
5  const int *g, *h; // Mutable pointer to const integer
6  int * const i, * const j; // Const pointer to mutable integer
7  const int * const k, * const l; // Const pointer to const integer
```

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

```
1  // NATI
2  Int32! a, b; // Mutable integer
3  Int32 c, d; // Const integer
4  Int32!?! e, f; // Mutable reference to mutable integer
5  Int32?! g, h; // Mutable reference to const integer
6  Int32!? i, j; // Const reference to mutable integer
7  Int32? k, l; // Const reference to const integer
```

2.1.4 Typová kontrola a konverze typů

NATI 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`)[4]. Implicitní konverze se provádí sama v případě potřeby[5].

- Typy mohou být explicitně konvertibilní do jiných typů (funkce `#explicitCastTo` a `#explicitCastFrom`). K explicitnímu přetypování se používá výhradně funkce `a.to( Type )` (lze použít i k implicitnímu přetypování).

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

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

```
1 // NATI
2 Int32 x = voidRef.to( Int32? );
```

2.1.5 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.

V C++ reference neumožňují měnit adresu odkazované paměti. V praxi ale programátor poměrně často potřebuje měnit hodnotu odkazu a 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 referencí má ale několik nevýhod:

[4] Viz specifikace jazyka NATI (v příloze), oddíl *Type casting*

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

- Programátor musí zajišťovat konstrukci (případně i destrukci) objektů
- Způsobuje více alokací a dealokací
- Indirekce bývá i tam, kde by být nemusela

Řešení NATI NATI 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 náš jazyk nabízí dva ukazatelové typy, kterým říkáme ukazatel a reference.

Způsob deklarace ukazatelů je v C++ i D problematický z hlediska parsování. 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 NATI je tímto znakem otazník (`?`). V C++ a D je používán pouze v ternárním operátoru (`cond ? expr1 : expr2`), kteréhož funkčnost NATI obstarává jiným způsobem[6]; tím pádem je znak otazníku „postradatelný“.

Ukazatel funguje skoro stejně jako v C++; podporuje ukazatelovou aritmetiku. Na dereferenci se nepoužívá prefixový operátor `*ptr` kvůli gramatickým konfliktům a protože může způsobovat nepřehlednost kódu a nejasnosti v prioritě operátorů (například u výrazu `*p++`). Místo toho se používá `ptr.data`. Přístup k prvkům odkazované hodnoty je možný pouze pomocí dereference. 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(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) a reference je explicitně konvertibilní na odkazovaný typ.

Reference v NATI 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 )`.

[6] Viz 2.2.3

2.1.6 Dekorátory

2.1.7 Vykonávání fukncí za doby kompilace

2.1.8 Metaprogramování

Deklarace s `if` v D

2.1.9 Mixiny

2.1.10 Unified function call syntax (UFCS)

2.1.11 Systém výjimek

2.1.12 Lambda výrazy

2.1.13 Traits

2.1.14 Standardní knihovna

2.2 Nové a netypické koncepty

2.2.1 Konstrukce `:ident`

2.2.2 Znak `#` v identifikátorech

2.2.3 Ternární operátor

2.2.4 Konstrukce `switch`

2.2.5 Vnořovatelné komentáře

2.2.6 Řetězcové literály

2.2.7 Typy IntXX, BinaryXX a Index

Kapitola 3

Koncept líhnutí kódu (*code hatching*)

Toto je nový koncept navržený pro jazyk NATI. Zasahuje do několika již známých konceptů – například šablonové metaprogramování, reflexe jazyka, vykonávání funkcí za doby kompilace; všechny tyto koncepty spojuje do jednoho koherentního celku.

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 NATI 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; nyní tuto myšlenku budeme rozvádět. Začneme jednoduchým příkladem:

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

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 );
```

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 tohoto 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  }
```

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 }
```

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 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 to pouze z kódu, v praxi by toto však značně zpomalilo kompilaci. 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 */
```

Zde je také 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 }
```

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`, kde každá manipuluje s proměnnou `x`). Nekonstantní statické proměnné se nedají ohlídat, **`@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 }
```

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 NATI 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 }
```

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

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" ] lookup
19   Int #operator( Operator.brackets, String key ) { ... }
20
21 }
22
23 BinaryTree sampleTree() {
24   BinaryTree result;
25   result.insert( "nati", 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["nati"] ); // 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 }
```

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 na té 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 z odkazu 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 odka-

zovaná paměť automaticky brala jako také konstantní. V oddílu 2.1.3 jsme ale rozhodli, že jazyk NATI 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*). NATI 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. Ochrana se dá například ještě rozšířit o zákaz existence mutabilních referencí a ukazatelů na dané bloky; v současné fázi návrhu jazyka ještě ale není jisté, zda by toto omezení nemohlo způsobit nějaké komplikace.

3.1.6 Přetypování na ne-`@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 výraz v nich datově závislý pouze na `@ctime` proměnných
   (a) Těla `@ctime` větvení nejsou nijak omezena
   (b) 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`.
6. Po inicializaci statické `@ctime` proměnné nelze zapisovat do paměti, která byla během její inicializace alokována.

3.3 Potenciál a důsledky konceptu

3.3.1 Vykonávání funkcí za doby kompilace

3.3.2 Šablonování a unifikace šablonových a klasických parametrů

3.3.3 Typové proměnné

3.3.4 `@ctime` funkce a třídy

3.3.5 Reflexe kódu

3.4 Implementační detaily

Přílohy

Příloha A

Jak pracovat s touto šablonou

V této kapitole je uveden popis jednotlivých částí šablony, po kterém následuje stručný návod, jak s touto šablonou pracovat.

Jedná se o přechodnou verzi šablony. Nová verze bude zveřejněna do konce roku 2016 a bude navíc obsahovat nové pokyny ke správnému využití šablony, závazné pokyny k vypracování bakalářských a diplomových prací (rekapitulace pokynů, které jsou dostupné na webu) a nezávazná doporučení od vybraných vedoucích. Jediné soubory, které se v nové verzi změní, budou projekt-01-kapitoly-chapters.tex a projekt-30-prilohy-appendices.tex, jejichž obsah každý student vymaže a nahradí vlastním. Šablonu lze tedy bez problémů využít i v současné verzi.

Popis částí šablony

Po rozbalení šablony naleznete následující soubory a adresáře:

bib-styles Styly literatury (viz níže).

obrazky-figures Adresář pro Vaše obrázky. Nyní obsahuje placeholder.pdf (tzv. TODO obrázek, který lze použít jako pomůcku při tvorbě technické zprávy), který se s prací neodevzdává. Název adresáře je vhodné zkrátit, aby byl jen ve zvoleném jazyce.

template-fig Obrázky šablony (znak VUT).

fitthesis.cls Šablona (definice vzhledu).

Makefile Makefile pro překlad, počítání normostran, sbalení apod. (viz níže).

projekt-01-kapitoly-chapters.tex Soubor pro Váš text (obsah nahraďte).

projekt-20-literatura-bibliography.bib Seznam literatury (viz níže).

projekt-30-prilohy-appendices.tex Soubor pro přílohy (obsah nahraďte).

projekt.tex Hlavní soubor práce – definice formálních částí.

Výchozí styl literatury (czechiso) je od Ing. Martínka, přičemž anglická verze (englishiso) je jeho překladem s drobnými modifikacemi. Oproti normě jsou v něm určité odlišnosti, ale na FIT je dlouhodobě akceptován. Alternativně můžete využít styl od Ing. Radima Loskota

nebo od Ing. Radka Pyšného[1]. Alternativní styly obsahují určitá vylepšení, ale zatím nebyly řádně otestovány větším množstvím uživatelů. Lze je považovat za beta verze pro zájemce, kteří svoji práci chtějí mít dokonalou do detailů a neváhají si nastudovat detaily správného formátování citací, aby si mohli ověřit, že je vysázený výsledek v pořádku.

Makefile kromě překladu do PDF nabízí i další funkce:

- přejmenování souborů (viz níže),
- počítání normostran,
- spuštění vlny pro doplnění nezlomitelných mezer,
- sbalení výsledku pro odeslání vedoucímu ke kontrole (zkontrolujte, zda sbalí všechny Vámi přidané soubory, a případně doplňte).

Nezapomeňte, že vlna neřeší všechny nezlomitelné mezery. Vždy je třeba manuální kontrola, zda na konci řádku nezůstalo něco nevhodného – viz Internetová jazyková příručka[2].

Pozor na číslování stránek! Pokud má obsah 2 strany a na 2. jsou jen „Přílohy“ a „Seznam příloh“ (ale žádná příloha tam není), z nějakého důvodu se posune číslování stránek o 1 (obsah „nesedí“). Stejný efekt má, když je na 2. či 3. stránce obsahu jen „Literatura“ a je možné, že tohoto problému lze dosáhnout i jinak. Řešení je několik (od úpravy obsahu, přes nastavení počítadla až po sofistikovanější metody). **Před odevzdáním proto vždy překontrolujte číslování stran!**

Doporučený postup práce se šablonou

1. **Zkontrolujte, zda máte aktuální verzi šablony.** Máte-li šablonu z předchozího roku, na stránkách fakulty již může být novější verze šablony s aktualizovanými informacemi, opravenými chybami apod.

2. **Zvolte si jazyk**, ve kterém budete psát svoji technickou zprávu (česky, slovensky nebo anglicky) a svoji volbu konzultujte s vedoucím práce (nebyla-li dohodnuta předem). Pokud Vámi zvoleným jazykem technické zprávy není čeština, nastavte příslušný parametr šablony v souboru projekt.tex (např.: `documentclass[english]{fitthesis}` a přeložte prohlášení a poděkování do angličtiny či slovenštiny.

3. **Přejmenujte soubory.** Po rozbalení je v šabloně soubor projekt.tex. Pokud jej přeložíte, vznikne PDF s technickou zprávou pojmenované projekt.pdf. Když vedoucímu více studentů pošle projekt.pdf ke kontrole, musí je pracně přejmenovávat. Proto je vždy vhodné tento soubor přejmenovat tak, aby obsahoval Váš login a (případně zkrácené) téma práce. Vyhněte se však použití mezer, diakritiky a speciálních znaků. Vhodný název tedy může být např.: „xlogin00-Cisteni-a-extrakce-textu.tex“. K přejmenování můžete využít i přiložený Makefile:

   ```
   make rename NAME=xlogin00-Cisteni-a-extrakce-textu
   ```

[1] BP Ing. Radka Pyšného http://www.fit.vutbr.cz/study/DP/BP.php?id=7848

[2] Internetová jazyková příručka http://prirucka.ujc.cas.cz/?id=880

4. Vyplňte požadované položky v souboru, který byl původně pojmenován projekt.tex, tedy typ, rok (odevzdání), název práce, svoje jméno, ústav (dle zadání), tituly a jméno vedoucího, abstrakt, klíčová slova a další formální náležitosti.

5. Nahraďte obsah souborů s kapitolami práce, literaturou a přílohami obsahem svojí technické zprávy. Jednotlivé přílohy či kapitoly práce může být výhodné uložit do samostatných souborů – rozhodnete-li se pro toto řešení, je doporučeno zachovat konvenci pro názvy souborů, přičemž za číslem bude následovat název kapitoly.

6. Nepotřebujete-li přílohy, zakomentujte příslušnou část v projekt.tex a příslušný soubor vyprázdněte či smažte. Nesnažte se prosím vymyslet nějakou neúčelnou přílohu jen proto, aby daný soubor bylo čím naplnit. Vhodnou přílohou může být obsah přiloženého paměťového média.

7. Nascanované zadání uložte do souboru zadani.pdf a povolte jeho vložení do práce parametrem šablony v projekt.tex (`documentclass[zadani]{fitthesis}`).

8. Nechcete-li odkazy tisknout barevně (tedy červený obsah – bez konzultace s vedoucím nedoporučuji), budete pro tisk vytvářet druhé PDF s tím, že nastavíte parametr šablony pro tisk: (`documentclass[zadani,print]{fitthesis}`). Barevné logo se nesmí tisknout černobíle!

9. Vzor desek, do kterých bude práce vyvázána, si vygenerujte v informačním systému fakulty u zadání. Pro disertační práci lze zapnout parametrem v šabloně (více naleznete v souboru fitthesis.cls).

10. Nezapomeňte, že zdrojové soubory i (obě verze) PDF musíte odevzdat na CD či jiném médiu přiloženém k technické zprávě.

Pokyny pro oboustranný tisk

- Zapíná se parametrem šablony: `\documentclass[twoside]{fitthesis}`
- Po vytištění oboustranného listu zkontrolujte, zda je při prosvícení sazební obrazec na obou stranách na stejné pozici. Méně kvalitní tiskárny s duplexní jednotkou mají často posun o 1–3 mm. Toto může být u některých tiskáren řešitelné tak, že vytisknete nejprve liché stránky, pak je dáte do stejného zásobníku a vytisknete sudé.
- Za titulním listem, obsahem, literaturou, úvodním listem příloh, seznamem příloh a případnými dalšími seznamy je třeba nechat volnou stránku, aby následující část začínala na liché stránce (\cleardoublepage).
- Konečný výsledek je nutné pečlivě překontrolovat.

Užitečné nástroje

Následující seznam není výčtem všech využitelných nástrojů. Máte-li vyzkoušený osvědčený nástroj, neváhejte jej využít. Pokud však nevíte, který nástroj si zvolit, můžete zvážit některý z následujících:

MikTeX LaTeX pro Windows – distribuce s jednoduchou instalací a vynikající automatizací stahování balíčků.

TeXstudio Přenositelné opensource GUI pro LaTeX. Ctrl+klik umožňuje přepínat mezi zdrojovým textem a PDF. Má integrovanou kontrolu pravopisu, zvýraznění syntaxe apod. Pro jeho využití je nejprve potřeba nainstalovat MikTeX.

JabRef Pěkný a jednoduchý program v Javě pro správu souborů s bibliografií (literaturou). Není potřeba se nic učit – poskytuje jednoduché okno a formulář pro editaci položek.

InkScape Přenositelný opensource editor vektorové grafiky (SVG i PDF). Vynikající nástroj pro tvorbu obrázků do odborného textu. Jeho ovládnutí je obtížnější, ale výsledky stojí za to.

GIT Vynikající pro týmovou spolupráci na projektech, ale může výrazně pomoci i jednomu autorovi. Umožňuje jednoduché verzování, zálohování a přenášení mezi více počítači.

Overleaf Online nástroj pro LaTeX. Přímo zobrazuje náhled a umožňuje jednoduchou spolupráci (vedoucí může průběžně sledovat psaní práce), vyhledávání ve zdrojovém textu kliknutím do PDF, kontrolu pravopisu apod. Zdarma jej však lze využít pouze s určitými omezeními (někomu stačí na disertaci, jiný na ně může narazit i při psaní bakalářské práce) a pro dlouhé texty je pomalejší.

Užitečné balíčky pro LaTeX

Studenti při sazbě textu často řeší stejné problémy. Některé z nich lze vyřešit následujícími balíčky pro LaTeX:

- `amsmath` – rozšířené možnosti sazby rovnic,
- `float, afterpage, placeins` – úprava umístění obrázků,
- `fancyvrb, alltt` – úpravy vlastností prostředí Verbatim,
- `makecell` – rozšíření možností tabulek,
- `pdflscape, rotating` – natočení stránky o 90 stupňů (pro obrázek či tabulku),
- `hyphenat` – úpravy dělení slov,
- `picture, epic, eepic` – přímé kreslení obrázků.

Některé balíčky jsou využity přímo v šabloně (v dolní části souboru fitthesis.cls). Nahlédnutí do jejich dokumentace může být rovněž užitečné.

Sloupec tabulky zarovnaný vlevo s pevnou šířkou je v šabloně definovaný „L“ (používá se jako „p“).