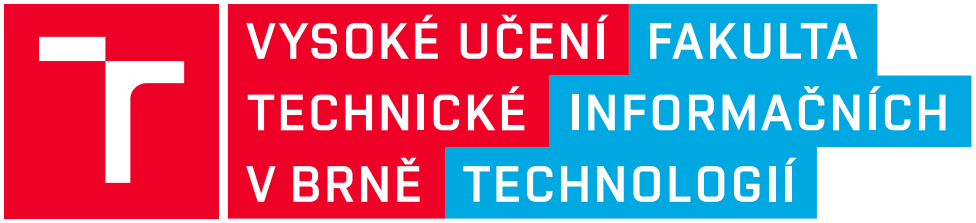

Prekladač nového modulárního jazyka

Semestrální projekt

Daniel Čejchan | xcejch00

ZS 2016/2017

1 | Obsah

2 | 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.

2.1 Motivace

Mou hlavní motivací je nespokojenost se stávajícími programovacími jazyky. Programování mám jako koníčka už od nějakých dvanácti let, a tak jsem ještě před nastoupením na FIT získal nějaké zkušenosti z psaní aplikací v Object Pascalu (Delphi), C++ (Qt, SDL + OpenGL) a PHP (webové aplikace). FIT pak můj repertoár (i když jen rámcově) rozšířil na valnou většinu dnes používaných jazyků. Bohužel jsem ale nenarazil na žádný, který by splňoval mé požadavky. Rozhodl jsem se tedy využít nutnost napsání bakalářské práce k uskutečnění mého dlouholetého snu.

Jazyk D Mému srdci nejbližší jazyk, na který jsem ve svém pátrání narazil, byl jazyk D. Jedná se o kompilovaný jazyk vycházející z C++ (binárky jsou do jisté míry kompatibilní) s velice podobnou syntaxí. Největší rozdíly jsou modulový systém[2] (zdrojový kód je rozdělen do modulů, které se vzájemně, i rekurzivně, importují; odpadá nutnost psát hlavičkové soubory), značně rozšířená funkčnost metaprogramování s šablonami a rozšířená schopnost vykonávat funkce za doby kompilace.

D mi byl velikou inspirací při navrhování mého jazyka. Bohužel i v D jsem narazil na strop možnosti (ačkoli byl značně výš než třeba v C++), kdy některé věci nešly napsat tak jednoduše, jak bych chtěl. Tento jazyk však dokazuje, že tato bariéra může být mnohem dál. Já ji chci ve svém jazyku ještě více posunout.

[1] http://dlang.org/

[2] http://dlang.org/spec/module.html

3 | 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 (předmětem projektu je jazyk, ne překladač) 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.

3.1 Prvky moderních programovacích jazyků

3.1.1 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.

3.1.2 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 jakožto knihovny; GC by tedy mohla být jedna ze základních knihoven jazyka. Tvorba této knihovny ale není předmětem tohoto projektu.

[1] https://dlang.org/spec/template-mixin.html

3.1.3 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 nepraktikuje, 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.

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).

Výsledky 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 3.1.4):

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

Alternativou by bylo třeba ještě zcela využít gramatiku C++, nicméně ta je všeobecně považována za velice matoucí. Ačkoli seskupení operátorů `?!?` (mutabilní reference na mutabilní hodnotu) může bezesporu v některých programátorech zpočátku vyvolat zděšení, první varianta je na tom ještě hůř. Po pochopení syntaktických pravidel, která jsou průhledná a jasná, se tato konstrukce ukáže být veskrze jednoduchá. Nicméně je třeba přiznat, že tento případ může pravděpodobně být nejhůře přijímanou věcí v jazyku.

Porovnání syntaxí C++, D a NATI

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

(3.1)

```
1 // D
2
3 int a, b; // Mutable integer
4 const int c, d; // Const integer
5 int* e, f; // Mutable pointer to mutable integer
```

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

[3] Viz specifikace jazyka NATI, oddíl *Decoration contexts*

```
6 const( int )* g, h; // Mutable pointer to const integer
7 // const pointer to mutable integer not possible
8 const int* k, l; // Const pointer to const integer
```

(3.2)

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

(3.3)

3.1.4 Reference, ukazatelé a jejich syntaxe

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). Očividným 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 (viz 3.2.1).

Ukazatel vs. reference Typ ukazatel umožňující ukazatelovou aritmetiku je v dnešní době považován za potenciálně nebezpečnou věc, která by se měla užívat jen v nutných případech. Kromě toho umožnění ukazatelové aritmetiky by mohlo způsobovat konflikty při přetěžování operátorů, což se řeší zavedením dereferencí (oddělí se jmenný prostor funkcí ukazatele od jmenného prostoru odkazovaného typu; místo `ptr.var` je `(*ptr).var`), případně speciálním operátorem pro přístup k prvkům odkazované hodnoty (`ptr->var`). Jako odpověď na toto C++ nabízí pseudotyp reference a D to řeší rozhodnutím, že třídy jsou vždy předávané referencí.

Problematika přístupu C++ Reference v C++ jsou konstantní (mohou ale odkazovat na nekonstantní data). V praxi ale programátor často potřebuje měnit, kam reference ukazuje, a

3.2 Další změny oproti C++/D

3.2.1 Ternární operátor