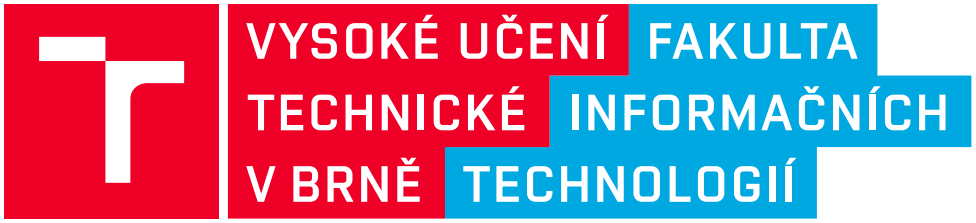

Prekladač nového modulárního jazyka

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 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