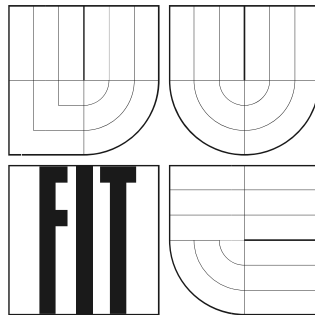


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Sémantické programování

Semestrální projekt

Sémantické programování

© Daniel Rozsnyó, 2006.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Daniel Rozsnyó
4. ledna 2006

Abstrakt

Práce si klade za cíl realizovat myšlenku zápisu a organizace zdrojových kódů programů v sémanticky čistější a také přenositelnější formě než je tomu u současných jazyků. K dosažení těchto cílů se používá abstrakce která osvobozuje programátora od implementačních detailů a omezení. Přenositelnost je zajištěna pomocí standardního formátu dat a vlastním návrhem jazyka, který je jak sjednocením tak zobecněním dnešních jazyků.

Klíčová slova

Programovací jazyk, syntaktický strom, překladač, generování kódu, datové typy, procedurální jazyk, objektově orientované programování, *Intentional Programming*, *Generative programming*, LOP – *Language oriented programming*, XML – *Extensible markup language*

Obsah

1	Úvod	1
2	Důvody a současná situace	2
2.1	Nezávislost.....	2
2.2	Rozšiřitelnost.....	4
2.3	Snížení chybovosti.....	5
2.4	Správa projektů.....	5
3	Návrh nového jazyka	7
3.1	Objektový návrh.....	7
3.2	Zápis v XML.....	7
3.3	Identifikátory.....	8
3.4	Datové typy.....	8
3.5	Objektově orientované programování.....	18
3.6	Zápis kódu.....	23
4	Závěr	30
	Literatura a jiné zdroje	31

1 Úvod

Programování bylo mým koníčkem již od základní školy a jak čas plynul, vyměnil jsem několik počítačů, programovacích jazyků i vývojových prostředí a bohužel to dospělo do stádia, že v současnosti žádné prostředí nebo jazyk nedokáže splnit moje požadavky týkající se jednoduchosti a síly. Z tohoto důvodu vznikla tedy myšlenka navrhnout si vlastní jazyk a příslušné vývojové prostředí, které bude v budoucnu pro programátora pohodlnějším nástrojem, než jsou ty dnešní.

Pokud chceme psát programy, potřebujeme znát alespoň jeden programovací jazyk. Znalost konkrétního jazyka nám ale nepostačuje, protože také potřebujeme další vybavení, jakým je např. překladač zvoleného jazyka nebo nástroj pro složitější projekty jménem integrované vývojové prostředí (IDE¹). Přestože existuje několik takovýchto nástrojů, situace není ideální. Volba jazyka závisí většinou na druhu projektu (pro web jiný, pro operační systémy jiný) a pokud už daný jazyk známe, potřebujeme znát také vlastní vývojové prostředí nebo nástroje (*toolchain*²) kvůli efektivnímu programování. To ale pořád není vše, protože je nutno znát prostředí kde program poběží a jaké knihovny funkcí jsou dostupné. Situace je dnes tedy taková, že existuje několik jazyků, několik prostředí, několik systémů a knihoven, jejichž konečným výsledkem je přitom jediná, stejná věc - program, který dělá přesně to, co po něm chceme.

Z uvedeného vyplývá, že pokud si chceme zjednodušit život, je nutno smazat rozdíly, které nám ho komplikují. Logicky prvním krokem je snížení počtu jazyků (ideálně na jeden) a to je také předmětem mé diplomové práce, v jejíž první části je podrobněji popsána situace se současnými programovacími jazyky a důvody, které vedou k potřebě vytvořit další, společný, programovací jazyk. Následně je tento jazyk popsán, od definic úplných základů jako datové typy, přes zápis konstant a řídicích konstrukcí až po organizaci kódu do větších celků. Druhá část se pak zabývá realizací překladače z nově navrženého jazyka do jednoho ze současných jazyků – jazyka C. Bez přímé možnosti použití je totiž jakýkoliv nový jazyk jenom teorie, která může, ale nemusí přinést očekávané zlepšení.

¹ Integrated Development Environment – nástroj umožňující jak překlad tak pohodlné ladění programů

² Sada nástrojů nebo utilit pro překlad programu, např. *make*, *gmake*, *gcc*, *g++*, *ld*, *strip*, atd..

2 Důvody a současná situace

Jak bylo naznačeno v úvodu, současná situace není ideální. Z toho důvodu je snaha o vytvoření nového jazyka. V případě, že zavedeme novou nevýhodu, bude muset být vyvážena výraznými výhodami, což by v konečném výsledku mělo současnou situaci zlepšit..

2.1 Nezávislost

2.1.1 Na druhu projektu

Pokud se cítíte v dnešní době univerzálním programátorem, musíte poznat několik jazyků. Píšete-li webové aplikace, použijete PHP, ASP, JSP nebo některý jazyk z rodiny .NET, když programujete systémové součásti, použijete pravděpodobně C, jestliže aplikace s grafickým rozhraním, tak C++, Visual Basic anebo Object Pascal (Delphi). Navíc ještě můžete znát assembler několika různých procesorů a výčet aplikací a jazyků neuzavírá ani popisný jazyk pro hardware, VHDL protože ještě existují speciální jazyky typu UML.

Seznam, i když neúplný, je dost rozsáhlý a postačuje pro účel ilustrace situace, že programování je dnes velice komplikované. Alespoň navenek. Pokud totiž rozdělíme jazyky na několik skupin, můžeme sice vidět rozdíl mezi jednotlivými skupinami (např. kategorie assembler a vyšší jazyky jako C), ale v rámci dané skupiny budou rozdíly minimální.

Jako příklad může posloužit programová konstrukce - cyklus, který provádí blok příkazů opakovaně dle podmínky. Ve všech jazycích vyšší úrovně se skládá ze stejných částí – výrazu , který se provede před cyklem, výrazu který rozhoduje o tom jestli se iterace provede nebo nikoliv, výrazu, který se provede na konci každé iterace a vlastního těla cyklu.

Pokud tedy víme, že všechny jazyky dělají téměř totéž, můžeme dané konstrukce nahradit společným zápisem a situace se tím zjednoduší. Zde ale nesmíme opomenout, že důvodem existence různých jazyků mohlo být také různé předurčení – hlavně co se týče prostředí.

Otázku uživatelského rozhraní (web, konzole, grafika) lze vyřešit jednoduše pomocí definice programového rozhraní, přes které bude program, aplikace, komunikovat s okolním světem. Některá tyto rozhraní jsou již analyzována a někdy i standardizována (SOAP³). Příklady známých rozhraní:

- standardní vstup a výstup
- textová obrazovka
- parametry na příkazové řádce
- HTTP požadavek a odpověď
- SOAP požadavek a odpověď – webové služby
- grafické uživatelské rozhraní

V praxi se setkáváme i s kombinací některých rozhraní, např. grafické aplikace, které pracují s parametry z příkazové řádky.

³ Simple Object Access Protocol – způsob volání webových služeb, <http://www.w3.org/TR/soap/>

2.1.2 Na velikosti projektu

Některé jazyky jsou předurčeny na tvorbu velkých systémů a některé nikoliv. Jeden z rozdílů, který o tom rozhoduje je například podpora jmenných prostorů.

Jmenné prostory jsou rozšířením systému pro pojmenovávání tříd (proměnných) v rámci projektu. V případě, že některému jazyku jmenné prostory schází, můžeme se při velkých projektech setkat s poměrně obskurními identifikátory, které jsou jak dlouhé, tak neoptimální, protože všechny jsou na stejné (jedné) úrovni. V jazyce který jmenné prostory obsahuje jsou identifikátory na první pohled kratší protože ve zdrojovém kódu jednoho modulu používá jen poslední (lokální) komponenta identifikátoru. Celý jednoznačný identifikátor - i když delší - lze pak logicky odvodit ze stromové hierarchie jmenných prostorů. Výhodou existence jmenných prostorů je pak snazší orientace ve velkém projektu, případně jednoznačnější členění na části.

Jazyky, které jmenné prostory mají jsou např. *Java*, *C++*, *C#*, *Perl*, a příklady jazyků bez jmenných prostorů: *C*, *PHP*, *Pascal*.

Jiná metoda jak zvládnout velký projekt je použití modelovacích technik, např. pomocí *UML*, které spočívá v zavedení abstrakce a tím i zjednodušení pochopení složitého problému. Nevýhoda *UML* spočívá v tom, že je dnes určeno jen pro modelování, pro konkrétní algoritmy je nutno sáhnout po jiném programovacím jazyku, většinou *Java* nebo *C++*, pro které dokáže modelovací program vytvořit kostru řešení. Pokud se ale v modelu něco změní, nastává pracné nebo méně pracné předělávání zbytku projektu psaného v jiném jazyce dle možností nástroje, který používáme. V případě, že v nově navrženém systému nebudeme rozlišovat mezi modelem a implementací, lze se uvedeným činnostem vyhnout.

Uvedené dva způsoby řešení problému s velkými projekty je možno zakomponovat do návrhu nového jazyka a to tím způsobem, že nabídneme hierarchickou strukturovanost. A to jak konkrétního kódu – jmenné prostory, tak několik vrstev abstrakce - modelování s tím, že model nebude sloužit jenom jako pomocný nástroj, ale bude přímo realizací řešení. Náš model bude reprezentovat celý program, od nejvyšší úrovně abstrakce po nejmenší detail.

2.1.3 Na cílovém prostředí

Při závislosti na prostředí se myslí většinou na programové prostředí jako operační systém, druh služby, způsob běhu. Také se ale může jednat o druh počítače – jestli je to normální PC nebo jednoúčelové zařízení na bázi procesorů *ARM* nebo *XScale*. Tyto zařízení jsou např. IP kamery, bezdrátové přístupové body (*Wifi Access Point*), počítače do dlaně (*PDA*) nebo průmyslové a vestavěné systémy.

Při vývoji nového programovacího jazyka bylo ale myšleno i na další možnosti, které nám poskytnou až technologie budoucnosti. Jedná se o součástky typu *FPGA*⁴, kde lze jednotlivé funkce realizovat s vysokou mírou efektivnosti, ale jako náhrada počítačů se tato technologie v dnešní době nepoužívá.

⁴ Field Programmable Gate Array – programovatelné hradlové pole

Důvod, proč je pro toto použití potřeba nového jazyka je ten, že dnešní programy byly psány a optimalizovány pro počítače s procesorem a pamětí a ne pro hradlové pole. Pokud se ale zapíše sémantika původních algoritmů ve formě srozumitelné pro počítač, bude možné vytvořit si konfiguraci hradlového pole, která realizuje příslušný algoritmus.

2.1.4 Na programovacím jazyku

Vedlejším efektem vysoké úrovně abstrakce a nezávislosti na implementaci je také to, že ze zdrojových kódů zapsaných v novém jazyce lze vygenerovat zdrojové kódy programu ve většině dnešních programovacích jazyků.

Využití vlastnosti je jednoduché – stačí napsat kód jednou a ihned ho budeme moci využít v různých projektech – např. použít cílový jazyk *PHP* pro webové aplikace a jazyk *C* pro aplikaci na příkazové řádce. Jiným příkladem je možnost použít verifikaci webových formulářů jak na straně klienta (*JavaScript*), tak na straně serveru (*PHP*, *C*) pomocí jednoho kódu, bez nutnosti psaní verifikačních knihoven zvlášť pro každý jazyk.

Pokud se vytvoří programy, které analyzují dnešní zdrojové kódy, bude možno překládat automaticky mezi různými programovacími jazyky. Tato funkcionality má také svoje opodstatnění – můžeme mít programátora, který používá volnější jazyk (*PHP*) a v případě potřeby většího výkonu si necháme přeložit aplikaci do jazyka *C*.

2.2 Rozšiřitelnost

Určitě si již mnoho lidí kladlo stejnou otázku jako my - proč existuje tolik programovacích jazyků když všechny dělají téměř totéž. Odpověď na tuto otázku může být dvojího druhu – buď je to důsledek komerce, komerčního boje firem o kousek na trhu s nástroji pro programování (např *Java*, *.NET*) anebo to je jen důsledek nemožnosti rozšiřovat stávající jazyky, takže je pokaždé nutné navrhnout nový, sice založený na starém základu, ale nekompatibilní.

Pokud se podíváme na situaci kolem jazyků *C* a *C++*, kde ten druhý je rozšířením prvního o prvky objektového programování, jmenné prostory a ještě kdoví co všechno, ale zároveň by měl být zpětně kompatibilní se svým předchůdcem, můžeme si položit další otázku – proč ten *nedokonalý* předchůdce nezapadl prachem a nadále se používá? Možná jsou to standardy, které má *C* v podobě *ANSI C* (a *ISO C99*) a v *C++* zůstávají nejasnosti co se týče některých detailů.

Pokud tedy navrhujeme nový jazyk, nebudeme ho zakládat na starém základu, protože přínos by byl minimální. Raději tedy zvolíme cestu od nuly a vytvoříme něco zcela nového, prvně jen v minimální verzi, ale již v základu s úmyslem pozdějšího rozšiřování.

Dle mého názoru existuje jen jeden způsob jak jednoduše dosáhnout uvedeného, a to použitím *XML*⁵. Někteří se můžou domnívat, že binární nebo i specializovaný textový formát by byl lepší, ale my se snažíme zvolit nejlepší variantu. Kromě toho, že nám *XML* poskytuje prostředky na neomezené rozšiřování také existují nástroje, které s ním dokážou pracovat nezávisle na oblasti použití - *XSLT*⁶.

⁵ Extensible Markup Language – rozšiřitelný značkovací jazyk

⁶ Extensible Stylesheet Language Transformations – rozšiřitelný jazyk pro transformace

2.3 Snížení chybovosti

Lidé obecně dělají chyby. Nevyhnou se jim ani programátoři a protože počet programátorů ve světě stoupá, absolutní počet chyb také narůstá. Řešení chyb se pak ponechává na později a vznikají mohutné systémy pro správu chyb a hlášení o chybách (svobodné programy) anebo se kritické chyby opraví a na zbytek se jednoduše nereaguje (komerční sféra).

Chybám se ale částečně lze vyhnout a to dokonce velice jednoduše – přenecháme některé činnosti stroji, který bude provádět operace s absolutní přesností a bez chyb (samozřejmě je podmínkou správné naprogramování tohoto stroje).

Dnes se nad nasazením tohoto přístupu neuvažuje, protože komerční podniky se orientují na okamžitou dosažení nejvyššího zisku, mnohokrát na úkor kvality produktu a lidé z protějšího konce – pracující na svobodných a otevřených programech – těch je počtem hodně a nepociťují potřebu měnit svůj způsob práce nebo vývoje.

Přínos, který přinese strojové zpracování pro programátora analytika, jeho zaměstnavatele a také pro jejich klienty, je pozitivní – jednotvárné práce bude méně a vývoj nebo dodávka řešení bude rychlejší. Udržovatelnost bude také na vyšší úrovni než dnes, protože o přepsání programu se postará stroj, jediné co je potřeba upravit je výchozí model.

2.4 Správa projektů

Správa projektů je důležitou součástí programování, protože čím je projekt větší, tím víc je potřeba v něm udržovat pořádek. V dnešní době je tento druh spravování ponechán na vývojářích a vývojové nástroje poskytují jen možnost automatizovat vybrané úkoly.

2.4.1 Snížení redundance

Programy obsahují značné množství společných částí. Pokud se nalezne několik funkcí, které má smysl oddělit a sdílet, tak se vytvoří knihovna (*DLL*⁷ nebo *SO*⁸). Tento způsob dělení a sdílení je ale už příliš pozdě, protože během psaní programu lze najít mnoho více společných částí – některé jsou poměrně krátké, takže vytvořit knihovnu se nevyplatilo, nebylo by to vůbec efektivní.

Způsob, který by byl ideální a je také navržen v této práci se týká vytvoření systému pro sdílení zdrojových kódů – globálního repositáře, kde budou na jednom místě dostupné a přístupné pro další použití veškeré algoritmy, funkce, třídy, metody, konstanty.

Jazyk nebo systém, který již takovouto vlastnost obsahuje je *Smalltalk* – všechny třídy a objekty jsou uloženy v objektové databázi, která obsahuje obraz aktuálního stavu systému, takže kdykoliv je možno práci přerušit a později pokračovat od stejného stavu dále. Tato vlastnost je pak v komerčních verzích i víceuživatelská, ale pořád má pro nás jednu nevýhodu – obsah

⁷ Dynamic Link Library – dynamicky linkovaná knihovna, svět operačního systému Windows

⁸ Shared Object – sdílený objekt, kód, obdoba DLL pro operační systémy Linux, Unix a jeho klony

databáze je určen primárně pro virtuální stroj a pro programování systémových součástí (ovladačů, sdílených knihoven) se nehodí.

Dále pak existují webové stránky (např. <http://www.phpclasses.org/>) zabývající se seskupováním zdrojových kódů, ale to je na velice nízké - manuální úrovni. Také existují centralizované systémy pro knihovny – pro *PHP* je to *PEAR*⁹, pro *Perl* zase *CPAN*¹⁰ - zde je situace lepší, ale pořád to má daleko k dokonalosti, protože jak *PEAR* tak *CPAN* je určeno na sdílení celých tříd implementujících určitý problém.

Námi navržený systém musí a i bude umožňovat sdílet části s jemnějším dělením, což jsou právě jenom konkrétní algoritmy nebo konstanty, bez nutnosti používat celou třídu. Navíc tento systém bude existovat online a v jediné verzi – aktuální. Základy v podobě hierarchie identifikátorů - jmenné prostory - již položeny jsou.

V případě, že všechny zdrojové kódy budou na jednom místě, naskytne se šance analyzovat tyto data a optimalizovat je. Jednou z aktivit, které lze provést je automatické hledání shodných konstrukcí a nahrazování je odkazem na společný zápis – v době překladu je pak chování obdobné *inline* funkcím známých z jazyka *C*. Podobné možnosti poskytuje nástroj *Intentional Programming* od firmy Microsoft, kde se nahrazování děje explicitně na povel programátora a vytvořená náhrada se jmenuje *enzyme*.

2.4.2 Dokumentace a vizualizace

Častým problémem u programových projektů je dokumentace. Některé projekty to řeší vytvořením obarvené hypertextové verze zdrojových kódů, nebo generováním dokumentace z poznámek, které jsou napsány určitým, jednoznačně identifikovatelným způsobem.

Nebylo by lepší mít tvorbu dokumentace integrovanou přímo do nástroje pro programování? Odpadne tím zmatek ohledně komentářů, přibude možnost přiložit obrázek jako obrázek a ne jako *ascii-art*¹¹. Nebo také jiné formáty jako tabulky a videa, obecně multimédia, budou reprezentovány nativně, bez potřeby konverze na čistý text. Jednoduchost použití těchto nových poznámek pak také záleží na vývojovém prostředí.

Primárním účelem dokumentace je také seznámit nového programátora se starým systémem, aby mohl pokračovat ve vývoji produktu. Kromě poznámek nebo psané dokumentace existuje také efektivnější metoda – a tou je vizualizace projektu nebo zdrojových kódů, protože jeden obrázek je častokrát více než několik slov.

⁹ PHP Extension and Application Repository - <http://pear.php.net/>

¹⁰ Comprehensive Perl Archive Network - <http://www.cpan.org/>

¹¹ Kreslení obrázků za pomoci písmen a jiných znaků v textovém editoru

3 Návrh nového jazyka

Tato kapitola se zabývá návrhem první, základní, verze nového programovacího jazyka. Jelikož se jedná o první návrh, je možné, že některé detaily budou po implementaci překladače blíže specifikovány, pozměněny nebo i zrušeny.

3.1 Objektový návrh

Objektový návrh znamená, že to, co se vytvoří (zapiše, naprogramuje) v novém jazyku si lze představit jako objekt. Navíc každá dílčí součást komplexnějšího zápisu je pochopitelně znova jakoby objekt. Objektem je tedy např. funkce, její parametry, typy těchto parametrů, návratový typ a tělo, stejně jako vše ostatní zapsané ve zdrojovém kódu v novém jazyce.

Důsledkem objektového návrhu je, že pro efektivní ukládání objektových dat, v našem případě zdrojových kódů, by se hodila objektově-relační databáze. Jediná nevýhoda takovéto databáze je, že není jednoduše rozšiřitelná, takže pokud se objeví ve zdrojovém kódu nějaká nová vlastnost nebo parametr, tak ji nebude možné uložit do databáze beze změny schématu příslušné databáze.

Abychom tomuto předešli, nebudeme prozatím používat objektově-relační databáze, ale zvolíme si jednodušší cestu – ukládání do souborů a vytvoříme si příslušnou přístupovou vrstvu, která zabezpečí načítání správných objektů dle identifikátoru nebo jiných požadavků.

3.2 Zápis v XML

Protože cokoliv, co se naprogramuje v novém jazyce bude tvořit *XML* dokument, je nutno specifikovat podrobnosti formátu – *XML* totiž není „všelák“.

Jednou z metod určení přesného formátu je použití *DTD*¹² souboru pro validaci. Tato možnost ale byla zavržena, protože *DTD* nepodporuje jmenné prostory *XML* a ty budou využívány v téměř každém dokumentu. *DTD* je ale také příliš jednoduché a na ověření komplexní struktury jakou je syntaktický strom nepostačuje. Další systémy pro ověření formátu jako *W3C XML Schema*, *RelaxNG*, nebo *Schematron* by na validaci byly pravděpodobně dostatečně silné, ale jejich aplikace by byla zdlouhavá a také není zajištěno, že poskytují možnosti, které tento projekt potřebuje, takže to může být zbytečná práce.

Problém s formátem dokumentů bude vyřešen vlastními silami - pomocí „agentů“, kteří budou dohlížet na aktuální stav repozitáře a nesrovnalosti budou hlásit, případně upravovat tak, aby byl stav znova konzistentní.

Pro účely popisu jazyka nám ale bude stačit vědět, že v uvedených *XML* zápisech se používají různé jmenné prostory pro značky, které se týkají typů, kódu a organizačních pravidel. Použité značky uvedené v dalším popisu jsou nezkrácené verze značek, což napovídá k existenci krátkých verzí – *aliasů*. Tyto zkrácené verze je pak možno použít při ručním psaní kódu na zjednodušení. V případě uložení dokumentu se ale použijí nezkrácené značky.

¹² Document Type Definition – jednoduchá definice obsahu dokumentu sloužící za účelem jeho validace

3.3 Identifikátory

Identifikátory v dnešních jazycích jsou poměrně omezené (co se týče použitelných znaků) a navíc pro první znak platí ještě víc omezení. V novém jazyku nic takového není a nebude, protože identifikátorem může být jakákoliv sekvence znaků UNICODE včetně znaku mezera. To také pak znamená, že je možno bezpečně použít lokalizované identifikátory, i když se jejich použití z důvodu vývoje na mezinárodní úrovni neodporučuje.

Identifikátory jsou pak hierarchicky organizovány obdobně jako adresářová struktura na disku. Pokud identifikátor nezačíná znakem pro kořenový adresář, tak se považuje automaticky za relativní, což ulehčí práci při ručním zápisu. Pokud se ale relativní objekt nenajde, bude se zkoušet načíst objekt z cesty absolutní, případně se budou prohledávat předem určené cesty (definována zvlášť pro každou XML značku). Několik příkladů na jednoduché identifikátory:

```
Počet
math/Pi
iso-8859-2
```

Aby se předešlo zbytečným komplikacím, je nutno říci, že zpětné lomítko v identifikátoru je uváděcí znak víceznakové sekvence, která pak umožňuje použít i znak „/“ (dopřední lomítko) v identifikátoru. Jinak se tento znak považuje za oddělovač hierarchické struktury.

Pokud již existuje tato svoboda v použití identifikátorů, může se navrhnout doporučení používat identifikátory podobné k URL. Příkladem:

```
example.com/Project1/Module2/Class3/Property4
```

3.4 Datové typy

Problematika datových typů je v novém jazyce řešena pomocí typových výrazů, popisným způsobem. Typové výrazy se skládají z typových operátorů a atomických typů, které jsou dále nedělitelné. První návrh pokrývá tyto atomické typy:

- Pravdivostní hodnota
- Číslo
- Znak
- Výčet

Ke konstrukci složených typů lze pak použít operátory:

- Volitelnost
- Struktura
- Množina
- Řetězec
- Pole
- Reference
- Funkce

3.4.1 Atomické datové typy

Základním a absolutním atomickým typem je typ pro uložení binární pravdivostní hodnoty, nebo také informace o rozsahu jednoho bitu. Protože jde o velice jednoduchou záležitost, zápis v XML je teda obdobně jednoduchý:

```
<boolean />
```

Ne všechno je ale tak triviální, čehož důkazem jsou typy pro ukládání čísel. Čísla obecně můžeme rozdělit na několik kategorií, jako jsou celá čísla, čísla s pohyblivou řádovou čárkou, komplexní čísla. Všechny tyto kategorie pak mají svá specifika, které se u programování v novém jazyce projeví jako parametry datových typů.

Celočíselné typy se vyznačují primárně dvěma parametry. První se týká znaménka, přesněji jeho existence a určuje, zda bude možno ukládat jenom kladné nebo kladné i záporné hodnoty. Druhým parametrem je pak počet bitů, reprezentující rozsah hodnot, které půjdou uložit do daného typu a také paměťovou náročnost. Jako sekundární parametry lze uvést minimální a maximální uchovatelnou hodnotu. Tyto dva parametry jsou zcela ekvivalentní primární, ale jejich specifikace je v některých případech složitější.

Příklad na obecný celočíselný typ:

```
<integer />
```

Příklad na specifikace typu, která uschová jeden bajt:

```
<integer size="8" signed="0" />
```

Příklad na specifikace typu, která uschová jeden bajt pomocí rozsahu hodnot:

```
<integer min="0" max="255" />
```

Celočíselné typy jak byly uvedené výše jsou po překladu nahrazeny optimálním typem, který splňuje požadavky kladené v parametrech. Pokud nechceme aby bylo při překladu provedeno toto nahrazování, což se vyžaduje např. při psaní ovladačů a mapování signálů, je nutno použít alternativní značku:

```
<signal size="32" signed="0" />
```

Ekvivalent k pravdivostní hodnotě:

```
<signal size="1" signed="0" />
```

Celočíselné typy jsou tedy definovány pomocí sady parametrů které jsou:

- Velikost dat v bitech (minimální, maximální, přesně)
- Znaménko (ano/ne)
- Informace, zda se započítává znaménko do uvedeného počtu bitů (ano/ne)
- Alternativna k uvedenému rozsah hodnot (minimum a maximum)
- Dovolení optimalizace nebo změny typu na větší, vícebitový

Další kategorií čísel jsou čísla v pohyblivé řádové čárce, kde je znova možné a někdy i potřebné upřesnit velikost. Tato informace o velikosti se pak při překladu nebo použití mapuje na jednu z možných velikostí čísel definovaných dle standardu IEEE. Pokud je definován větší datový typ než je procesor schopný zpracovat, naskytuje se možnost zpracovávat uvedený typ pomocí emulace. Na místech kde ale na velikosti nezáleží, je možno použít obecnou definici:

```
<float />
```

Pokud ale požadujeme definovanou přesnost, je možno použít zápisu:

```
<float size="80" />
```

U desetinných čísel je alternativním parametrem k bitové délce přesnost na číslice:

```
<float precision="6" />
```

Zde se již začínají projevovat přednosti volného zápisu a možnosti upravit si překladač dle libosti, takže alternativou k definování přesnosti může být také:

```
<float precision="1e-6" />
```

Poslední kategorií čísel jsou komplexní čísla, které jsou v definici jen kvůli sémantické podstatě jazyka. I když je implementace komplexních čísel závislá na knihovnách, tj. emulaci výpočtů, pro programátora je důležité, že to co zpracovává je komplexní číslo a ne struktura o dvou prvcích. Příkladem:

```
<complex />
```

Pokud se vyžaduje přesnost, je možno použít stejné parametry jako u typu s plovoucí desetinnou čárkou.

Dalším datovým typem je znak. V původních programech se za znak považoval vždy jeden bajt, ale doba ukázala, že to nebylo dostatečné, protože byla nutnost zpracovávat texty celosvětově, s různými znaky a diakritickými znaménky. V novém jazyce je možno přímo určit, jestli je znak jako jeden nebo více bajtů. Nejprve ale příklad pro obecný zápis:

```
<char />
```

Který lze parametrem upravit dle libosti na 1, 2, nebo 3 bajtové znaky:

```
<char size="8" />
```

```
<char size="16" />
```

```
<char size="24" />
```

Speciálním příkladem jsou znaky pro čisté ASCII a sadu UNICODE:

```
<char size="7" />
```

```
<char size="31" />
```

Při použití těchto typů dostáváme přidanou hodnotu v podobě možnosti kontroly mezí, která nám může odhalit problematické části programu.

Problematika znaků se ale tímto neuzavírá, protože pro znakový typ jde specifikovat znaková sada. Tímto se automaticky zabezpečí správný překlad mezi různými znakovými sadami, což sníží náročnost implementace projektu. Definice formátu pro specifikaci znakové sady ale není součástí práce, pro informaci jen tolik, že používá stejné identifikátory a systém zpřístupnění jako celý projekt. Operace které lze pak nad znakem provést je určení znakové sady bez konverze a konverze do jiné znakové sady.

Výčtový typ je posledním definovaným atomickým typem. Jeho konstrukce je složitější v porovnání s předešlými typy, protože jde o strukturovanou záležitost. Příkladem definice výčtu pro dny v týdnu:

```
<enumeration>
  <item id="Pondělí" />
  <item id="Středa" />
  <item id="Čtvrtek" />
  <item id="Pátek" />
  <item id="Úterý" />
  <item id="Sobota" />
  <item id="Neděle" />
</enumeration>
```

Parametry výčtového typu slouží na očíslování jednotlivých položek, pokud chceme aby hodnota pro pondělí byla jedna, použijeme zápis:

```
<enumeration>
  <item value="1" id="Pondělí" />
  <item value="3" id="Středa" />
  <item value="4" id="Čtvrtek" />
  <item value="5" id="Pátek" />
  <item value="2" id="Úterý" />
  <item value="6" id="Sobota" />
  <item value="7" id="Neděle" />
</enumeration>
```

Stejného efektu dosáhneme vynecháním explicitního číslování a zavedením automatického:

```
<enumeration from="1">
  :
```

Jen pro úplnost - číslovat je možno nejen lineárně:

```
<enumeration from="1" step="+1">
  :
```

Ale i exponenciálně, třeba po bitech:

```
<enumeration from="1" step="*2">
  :
```

3.4.2 Typové operátory

Typové operátory slouží pro tvorbu komplexnějších datových struktur. I když je v stávajících jazycích podporováno všechno, co bude uvedeno dále, většina z nich nerozlišuje např. mezi polem a řetězcem. Toto chování v sémantickém programování není přípustné, protože musí být jednoznačné s jakými typy se pracuje. Různé typy také umožňují různé operace mezi kterými by se také měl dělat rozdíl, za účelem jemnější kontroly na vyšší úrovni.

Prvním operátorem je operátor volitelnosti, který určitému datového typu přidá vlastnost, že jeho hodnota nemusí být definovaná. Tato funkcionalita je známá především z databází, kde lze definovat sloupec tabulky jako volitelný. Aplikace operátoru je následovná:

```
<optional>
  <boolean />
</optional>
```

Zde je názorně vidět jednu z výhod XML – možnost uložit strukturovaná data při zachování jednoduchosti zápisu případně ručních úprav.

Vnější značky jsou typovým operátorem volitelnosti – *optional* – aplikovaným na jeho vnitřní část – typ *boolean*. Proměnná tohoto typu může uchovávat celou množinu hodnot, která je odvozena od vnitřního typu, v tomto případě hodnoty *true* a *false* a protože jsme aplikovali operátor volitelnosti, k této množině přibude další hodnota – *null* – označující stav nedefinované hodnoty.

Další typový operátor slouží na tvorbu struktur. Struktura je datový typ, který sdružuje několik pojmenovaných položek různých typů. Identifikátory položek musí být unikátní v rámci jedné struktury. Jednoduchý příklad, jak vytvořit strukturu, která je podobná komplexnímu číslu:

```
<structure>
  <element id="Re"><float /></element>
  <element id="Im"><float /></element>
</structure>
```

Předešlé dva operátory bylo možno aplikovat na libovolný typ, což u následujícího ale neplatí – jedná se totiž o operátor pro tvorbu množin, které lze vytvořit jen z výčtového typu. Příkladem typ, který má vlastně tři bity a osm stavů:

```
<set>
  <enumeration>
    <item id="Vlevo" />
    <item id="Vpravo" />
    <item id="Rovně" />
  </enumeration>
</set>
```

Praktické použití takového typu je na reprezentaci jednoduché křížovky sestávající z cest do tří směrů.

Další operátor aplikovatelný jenom na určitý základní typ slouží pro tvorbu řetězců. Tyto jsou předurčeny na ukládání textových informací, takže typem, nad kterým se řetězce zakládají je znak. Operátor pro řetězec je ale speciální – dovoluje nám vytvořit svůj složený typ i bez specifikace základního typu. Tento zápis je tedy nejobecnější:

```
<string />
```

Pokud chceme ukládat jen základní znaky z tabulky ASCII, můžeme použít zápisu:

```
<string>  
  <char size="7" />  
</string>
```

Popisovaný operátor pro řetězce je tedy parametrickým operátorem – má jeden až tři parametry, které určují jeho chování. První parametr určuje typ řetězce, tj. jedna volba z následujícího seznamu:

- nulou ukončený řetězec (*asciiz*)
- dynamicky alokovaný řetězec (*dynamic*)
- staticky alokovaný řetězec (*static*)

Další parametry mají význam jen pro staticky alokované řetězce. Jeden určuje počet znaků které se alokují v paměti a druhý označuje způsob práce s řetězcem – jestli je možné ukládat i kratší řetězce, nebo jenom řetězce délky specifikované počtem znaků.

Příkladem pro tyto volby budou reprezentace řetězců v jazycích C:

```
<string type="asciiz" />
```

a *Pascal*:

```
<string type="static" size="255" dynamic="1" />
```

Operátor pro tvorbu polí je podobný operátoru pro řetězce v tom, že umožňuje provádět operaci indexování, ale tím shody končí, protože jednak pole nelze postavit bez specifikace základního typu a dále tento typ může být libovolný (na rozdíl od textových řetězců).

Obecné pole čísel tedy definujeme jako:

```
<array>  
  <integer />  
</array>
```

V praxi je tento zápis příliš obecný, takže je zde možnost parametrizovat typ – určit, zda je pole statické velikosti, nebo dynamické a také je-li číslováno od 0 nebo 1 případně jinak. U parametru je tedy možno určit statickou velikost, což je vhodné na číselné sekvence známé velikosti. Příkladem reprezentace IPv4 adresy po jednotlivých číslech:

```
<array size="4">  
  <integer min="0" max="255" />  
</array>
```

Pokud známe počátek i konec číslování, můžeme použít kombinace parametrů *min* a *max*, pokud známe jen počátek a velikost, tak použijeme *min* a *size*. Typ vhodný na ukládání informací pro každý den v kalendářním měsíci je tedy:

```
<array min="1" max="31">
:
</array>
```

Jak bylo uvedeno, je možno vytvořit i dynamické pole. Např. pro nula až 1024 položek:

```
<array size="1024" dynamic="1">
:
</array>
```

Toto pole ale má nevýhodu, protože při každé změně velikosti je nutno alokovat paměť. Výhodnější je tedy použít staticky alokované pole o 1024 prvcích:

```
<array type="static" size="1024" dynamic="1">
:
</array>
```

Reference je operátor známý z programovacích jazyků, které dovolují používat ukazatele. Operátor reference v našem případě vytvoří z datového typu takový typ, který může obsahovat jako svou hodnotu ukazatel. Zápis reference třeba na pole je:

```
<reference>
<array>
:
</array>
</reference>
```

Praktické použití pro referenci je pak v případě programovacích jazyků bez podpory předávání parametrů referencí – zde můžeme definovat parametr jako referenci a tuto referenci předat hodnotou. V našem novém programovacím jazyce se i pole jako parametr funkce bude předávat hodnotou, a existují jen dvě možnosti jako tomu zabránit – použít typ s referencí, nebo nadefinovat argument funkce jako předávaný odkazem.

Zde jsme se dostali až k funkcím, které i když přímo datovým typem nejsou, typ mají. Každý typ funkce je totiž určen pomocí typu návratové hodnoty a typu parametrů. Pro začátek příklad na zápis jednoduché funkce vracející číselnou hodnotu:

```
<function>
<return>
<integer />
</return>
</function>
```

Ne všechny funkce jsou ale takto jednoduché, protože obsahují parametry. Tyto parametry jsou určeny svým pořadím, typem a také unikátním jménem, které je pak dále použito jako jméno argumentu. Definice funkce s parametrem je tedy např.:

```
<function>
  <return>
    <integer signed="0" />
  </return>
  <parameter id="text">
    <string />
  </parameter>
</function>
```

Na základě tohoto typu lze stavět funkce akceptující řetězec a vracející nezáporné číslo – např. funkci pro spočtení počtu znaků. V případě, že funkce nemění obsah parametru, což uvedené spočtení znaků nedělá, můžeme parametr předávat také odkazem:

```
<parameter id="text" pass="reference">
  <string />
</parameter>
```

Výhodou tohoto přístupu je větší efektivita funkce, protože bude pracovat přímo nad zdrojovými daty, které se nemusí kopírovat na dočasné paměťové místo. Pokud ale naopak vyžadujeme předávání parametru pomocí hodnoty, třeba z důvodu použití argumentu funkce jako další proměnné při výpočtu, uvedeme to jako:

```
<parameter id="text" pass="value">
  :
</parameter>
```

Návratová hodnota se pak předává vždy hodnotou pokud existuje. V případě funkcí bez návratové hodnoty, označovaných většinou jako procedury (např. v jazyku *Pascal*) je část *return* nepovinná. Když tato procedura nemá ani parametry, definujeme ji triviálně, jako:

```
<function />
```

Mezi parametry, které mohou ovlivňovat zpracování funkcí lze zařadit vlastnost označovanou jako *inline funkce*. V jazyce *C* se např. definuje pro každou funkci, v našem programovacím jazyce ji ale bude možno odvolat na místě volání, takže se použije standardní způsob volání. Zapnutí vlastnosti provedeme parametrem:

```
<function inline="1">
  :
</function>
```

Speciální vlastnosti, jako např. proměnlivý počet parametrů nejsou v této době součástí specifikace jazyka.

3.4.3 Speciální datové typy

Mezi speciální datové typy lze zařadit následující dva typy, ve své podstatě komplementární. Jeden z nich reprezentuje libovolný datový typ, se zápisem:

```
<any />
```

Druhý naopak nereprezentuje nic a zapisuje se jako:

```
<void />
```

Existence těchto typů není jen z důvody svobody programovacího jazyka, existují i praktické situace které dané typy vyžadují – např. import stávajícího kódu, kde beztypové programovací jazyky jako *PHP* mají vlastně všechno libovolného typu a programovací jazyk *C* obsahuje značné množství typu žádný typ.

Tyto speciální typy ale nejsou přímo všeobecně použitelné, takže existují způsoby jak je eliminovat. Jeden z nich je založen na ruční úpravě, což je dost neefektivní způsob v dnešní době. Z tohoto důvodu existuje vylepšení, poloautomatická úprava zakládající se na faktu, že jak operátory, tak knihovní funkce, i v beztypovém jazyce jako je *PHP*, logicky očekávají vstupy určitých typů a z této informace si lze odvodit minimálně typovou kategorii (číslo, text, pole, struktura). Zbylé detaily jako velikost typů je nutno specifikovat ručně, nebo provést simulaci programu pro statistické zjištění limitů (také součást strojové optimalizace programů).

Dalším speciálním typem v obdobném duchu je typ pro obecné číslo – tj. sdružující čísla jak celá, tak v pohyblivé řádové čárce a také komplexní. Význam typu je ve výše uvedeném procesu určování typů v původně beztypovém zdrojovém kódu a zapisuje se jako

```
<number />
```

3.4.4 Definice uživatelských typů

Pro definici uživatelských datových typů jsou vyhrazeny dvě značky. První definuje typ a druhá je referencí na typ takto definovaný. Příklad definice typu nazvaného *myComplex*:

```
<define id="myComplex">
  <structure>
    <element id="Re"><float /></element>
    <element id="Im"><float /></element>
  </structure>
</define>
```

Reference na typ je pak zapsána jako:

```
<type>myComplex</type>
```

Tato reference se při zpracování zdrojového kódu nahradí obsahem definice, v našem příkladě složeným typem struktura.

3.4.5 Pravidla pro kontrolu chyb v jednoduchých typech

Protože definice datových typů je poměrně volná, je nutno zavést kontrolní mechanismy, které nám buď nahlásí nesrovnalosti nebo v lepším případě i sami udrží zdrojové kódy validní. Tyto kontroly za pomoci agentů budou tedy prováděny již v době editace zdrojových kódů, takže hlášení o chybách budou na vyšší úrovni než u dnešních programovacích jazyků, kde je seznam chyb dostupný až při překladu.

Tato kapitola tedy obsahuje popis několik chyb, které mohou vzniknout v době editace:

- při použití typového operátoru volitelnosti na typ, který již volitelný je, se musí zobrazit hlášení o možné chybě. Stav tady ještě není kritický a možná mohou existovat i odůvodněné případy, kdy je potřeba struktury jako je tato:

```
<optional>
  <optional>
    :
  </optional>
</optional>
```

- nejednoznačnost elementů v rámci struktury, je již nepřekonatelným problémem a pokud projekt obsahuje takovou chybu tak není validní. Příklad konstrukce:

```
<structure>
  <element id="A"> .. </element>
  <element id="A"> .. </element>
</structure>
```

- operátor pro tvorbu množin musí obsahovat výčet. Pokud je prázdný nebo aplikován na jiný typ, hlásí se chyba a překlad není možný.
- operátory pro řetězce a pole nesmí definovat parametry, které se neberou v potaz při existenci jiného parametru, protože to vyvolává varování. Také parametry nesmí být protichůdné, např. operátor pro tvorbu pole nemůže definovat všechny tři parametry *min*, *max* i *size* protože pak se hlásí chyba, jen v případě, že tyto velikosti souhlasí, tak se vyvolá varování.
- funkce také nesmí obsahovat dva různé parametry se stejným názvem a názvy musí být uvedeny (to platí ale také i pro strukturu)
- ve funkci se nesmí nacházet více sekcí definujících návratový typ.

Seznam pravidel není úplný, jelikož proces hledání a definice chyb je kontinuální a další chyby se jistě objeví až po nasazení jazyka do praxe.

3.5 Objektově orientované programování

V dnešní době je nejsilnější pomůckou pro tvorbu programů objektově orientovaný návrh. Protože nový jazyk nechce zavádět zcela nové pravidla pro programování tak staví na osvědčených základech, které zahrnují nejenom procedurální programování ale také zmiňovaný objektově orientovaný přístup.

Pokud se podíváme na situaci v implementaci těchto vlastností v současných programovacích jazycích, zjistíme, že různé jazyky poskytují různé úrovně možností. Tyto se také liší od verze jazyka a také i od verze nebo dodavatele překladače. Nejvíce možností v tomto směru poskytuje dle mého názoru modelovací jazyk *UML*, ale bohužel to není skutečný programovací jazyk.

Objektově orientovaný přístup je založen na několika pojmech, jako jsou třída, objekt, zapouzdření, dědičnost, abstrakce a polymorfismus. Všechny tyto prvky obsahuje dále uvedený návrh objektově orientovaného rozšíření nového programovacího jazyka.

3.5.1 Třídy, rozhraní a zapouzdření

Úplným základem objektově orientovaného návrhu je rozložení problému na objekty, které přísluší do různých tříd. Objekty těchto tříd komunikují pomocí předem určených kanálů nazývaných rozhraní. Jedna třída může implementovat zároveň několik rozhraní. Definice třídy:

```
<class id="obdélník">
:
</class>
```

Zde je postřehnutelný rozdíl oproti dříve uvedeným datovým typům – třídy jsou pojmenované. Když je pak potřeba se odkazovat na třídu ve smyslu datového typu, použijeme následující značení:

```
<object>obdélník</object>
```

Většina dnešních programovacích jazyků nepracuje přímo s objekty, ale jen s odkazem na objekt. Tyto odkazy je možno vytvořit pomocí typového operátoru reference, definovaného dříve:

```
<reference>
  <object>obdélník</object>
</reference>
```

Rozhraní se definuje obdobně jako třída, rozdíl je jenom v značce:

```
<interface id="operace s útvary">
:
</interface>
```

Rozdíl mezi třídou a rozhráním je ten, že rozhraní neimplementuje své metody, jen definuje jejich typy. V novém programovacím jazyce ale nic nebrání mít některou z metod rozhraní implementovanou, tudíž rozhraní a třída je téměř to samé, rozdíl je jen v primárním účelu.

Když chceme, aby naše třída implementovala nějaké rozhraní, tak musíme toto rozhraní uvést v definici třídy:

```
<class id="obdélník">
  <implements interface="operace s útvary">
  :
  </implements>
  :
```

Tento způsob zápisu byl zvolen z důvodu, že nastane kolize mezi názvy prvků dvou různých rozhraní. Implementace metod jsou totiž uvnitř značek, které nesou odkaz na rozhraní, takže je pak jednoznačné kam přísluší. U volání metody je pak nutno explicitně uvést rozhraní, přes které chceme dané volání provést.

Do rozhraní se metoda dostane následovně:

```
<interface id="operace s útvary">
  <method id="plocha">
    <function>
      <return>
        <float />
      </return>
    </function>
  </method>
  <method id="obvod">
  :
  </method>
</interface>
```

Pokud jsou funkce stejného typu, doporučuje se vytvořit si vlastní (pojmenovaný) datový typ a v definici rozhraní použít odkaz na tento typ.

Vlastnost zvaná zapouzdření označuje druh přístupu, kde ke změně obsahu objektu může docházet jen za pomoci veřejných metod třídy nebo přes rozhraní. Abychom mohli provést tyto operace, musíme si definovat obsah objektu a viditelnost metod.

Obsahem objektu jsou převážně vlastnosti (z angl. *properties*), které v jednodušším případě připomínají prvky datového typu struktura. Složitější případ je ten, když vlastnost není takto materializována a existuje jen virtuálně pomocí speciálních metod pro zápis a čtení. Definice jednoduché vlastnosti:

```
<class id="obdélník">
  <property id="a"><float /></property>
  <property id="b"><float /></property>
  :
```

Tyto vlastnosti jsou skutečné, protože jejich obsah je uložen v datové části objektu. Můžeme ale také definovat vlastnost, která nebude uložena, ale definována pomocí funkcí. Praktickým příkladem může být úhlopříčka – její velikost je odvoditelná ze stran obdélníku:

```
<class id="obdélník">
  <property id="úhlopříčka">
    <float />
    <get>
      :
    </get>
  </property>
```

Zde musí být funkcionalita pro výpočet implementována v části *get*. Vlastnost úhlopříčky nemusí být jenom pro čtení, můžeme předpokládat i zápis do této vlastnosti za účelem změny velikosti obdélníku se zachováním poměru stran. V tomto případě implementujeme kód do části *set*.

O tom, jestli bude vlastnost materializována můžeme rozhodnout algoritmem, ale když chceme mít jistotu, použijeme parametr:

```
<property id="úhlopříčka" virtual="1">
  :
```

Takto můžeme definovat vlastnosti, které implementuje až třída specifikovaná. Viditelnost vlastností se řeší také parametrem:

```
<class id="obdélník">
  <property visibility="public" id="a"> ..
  <property visibility="public" id="b"> ..
  :
  <property visibility="private" id="cacheO"> ..
  <property visibility="private" id="cacheS"> ..
  :
```

Hodnota tohoto parametru může být zatím jen jedna ze tří základních:

- Veřejné prvky – *public*
- Chráněné prvky - *protected*
- Soukromé prvky – *private*

K veřejným prvkům přístup není omezován, chráněné lze použít jen v aktuální třídě a jejich potomcích, soukromé pak jen v aktuální třídě. Viditelnost lze aplikovat také na metody, které jsou dalším druhem prvků třídy a jejich definice je stejná jako u metod v rozhraních:

```
<class id="obdélník">
  <method id="normalizovat obvod">
    <function />
  </method>
  :
```

Implementace metody je možná v rámci definice jejího typu, nebo také později – stačí uvést identifikátor metody.

Další vlastnost, která se týká prvků třídy je staticnost, možnost definice vlastnosti nebo metody jako statické, čímž dosáhneme toho, že bude možno na ně odkazovat a volat je i v případě, že neexistuje žádná instance třídy jako objekt. Příklad:

```
<class id="obdélník">
  <property static="1" id="počet objektů">
    <integer />
  </property>
  :
```

Každá třída má také několik speciálních metod, které se volají ve speciálních případech, např. při vytváření instance třídy (*konstruktor*), nebo zániku instance (*destruktor*). Také můžeme definovat vlastní chování pro případ přiřazení (*přiřazovací konstruktor*) nebo vytvoření kopie (*klonování*). Všechny tyto speciální metody mají svoje značky, které jsou:

```
<class id="obdélník">
  <construct>
    :
  </construct>
  <destruct>
    :
  </destruct>
  <copy>
    :
  </copy>
  <clone>
    :
  </clone>
  :
```

Kromě vlastností prvků tříd jako jsou metody a vlastnosti existují také vlastnosti, které se týkají celé třídy. Jednou z těchto vlastností je definice třídy jako abstraktní, což zabrání vytváření instancí:

```
<class id="tvar" abstract="1">
  :
```

Třída je za abstraktní prohlášena také v případě, když existují metody (vlastní nebo z rozhraní), které nejsou implementované.

Druhá vlastnost, která zabraňuje tvorbě specifikovaných tříd, je prohlášení třídy za finální. Tato vlastnost se ale musí používat s uvážením, protože může zabraňovat znovupoužitelnosti kódu:

```
<class id="čtverec" final="1">
  :
```

3.5.2 Dědičnost

Tvorbu komplexnějších tříd v objektově orientovaném návrhu ulehčuje možnost dědit vlastnosti a metody. Tato dědičnost je ve většině dnešních programovacích jazyků jen jednoduché určení třídy, kterou rozšiřujeme, ze které dědíme. Omezení na jednu třídu bylo zavedeno z jednoduchosti tvorby překladače, programovacího jazyka nebo teorie, ale my se toho nebudeme držet a definujeme pravidla pro vícenásobné dědění.

Příkladem:

```
<class id="obdélník">
  <inherits class="tvar">
  :
</inherits>
:
```

Vícenásobnou aplikací uvedených značek dosáhneme vícenásobné dědičnosti s jednoznačnou prioritou metod. Metody zděděné pak můžeme nahradit vlastním kódem, jediné co musíme udělat je vybrat si metodu a dopsat vlastní kód. To samé platí pro implementaci abstraktní metody.

Pomocí vhodného parametru pak můžeme docílit ignoraci metody při dědění, což je jeden z nástrojů pro řešení konfliktů souvisejících s vícenásobným děděním:

```
<class id="obdélník">
  <inherits class="tvar">
  <method ignore="1" ..
  :

```

Dalším nástrojem pro změnu pořadí zavedení metod je priorita, metody s vyšší prioritou jsou aplikovány jako poslední, takže ta, která má prioritu nejvyšší je také chápána jako výchozí implementace metody:

```
<class id="obdélník">
  <inherits class="tvar">
  <method priority="1" ..
  :

```

Ve většině případů stačí označit jednu metodu parametrem pro prioritu protože ty, které prioritu uvedenou nemají, se považují za metody s prioritou nižší než je nejmenší uvedená.

3.6 Zápis kódu

Značky pro zápis kódu přísluší jinému jmennému prostoru XML a rozlišovací znaky (prefix) bude uveden jen v případě, že se jedná o část zdrojového textu, kde se tyto značky míchají s jinými, např. pro definici typů.

Na místě, kde dochází k potřebě specifikovat přímo kód se nachází úvodní značka, sloužící pro zjednodušení ručního zápisu, jelikož je to vhodné místo na změnu výchozího jmenného prostoru XML:

```
<class id="obdélník" xmlns="TYPE">
  <property id="úhlopříčka">
    <float />
    <get>
      <code xmlns="CODE">
        :
      </code>
    </get>
  </property>
</class>
```

Jazyk je postaven nad výrazy, které se vyhodnocují v pořadí jakém jsou uvedeny. Změna pořadí je možná pomocí řídicích konstrukcí – větvení a cyklů. Nejprve si ale definujeme výrazy a jejich skladbu – atomické výrazy a operátory.

3.6.1 Atomické výrazy

Výrazy jsou na nejnižší úrovni složené z několika základních prvků jako konstanty a proměnné. Konstanty představují kapitolu samou o sobě, protože pro ně byl vyčleněn další jmenný prostor, aby bylo možné odlišit zejména jestli myslíme typ číslo, nebo číselnou konstantu. Některé značky pro konstanty jsou jiné než jejich typy:

```
<true />
<false />
<null />
```

Většina značek je ale podobná typům, které mají jejich hodnoty, např.:

```
<integer>2006</integer>
<float>3.14159265</float>
<string>Hello world!</string>
```

Dalším atomickým výrazem je přístup k proměnné. Abychom mohli definovat přístup k proměnné, musíme definovat nejprve proměnnou a jejich typy, protože jde o sémantické programování, kde má všechno svůj význam. V praxi to znamená, že se rozlišují minimálně lokální proměnné a argumenty funkcí, ke kterým se přistupuje následovně:

```
<variable>index</variable>
<parameter>text</parameter>
```

Definice parametru byla součástí definice typu funkce, proměnná se ale definovala jako:

```
<local id="index">
  <integer />
</local>
```

Vhodné umístění této definice je před prvním použitím v dané funkci.

Globální proměnné nejsou prozatím definovány, ale můžeme použít statické vlastnosti tříd, které plně nahradí uvedenou funkcionalitu. Pro přístup k vlastnosti třídy se použije:

```
<property>a</property>
```

V případě přístupu k statickému prvku z jiné třídy se musí specifikovat o kterou třídu jde, zápis je tedy nepatrně komplikovanější:

```
<property class="obdélník">počet</property>
```

Značky pro přístup k proměnným nebo vlastnostem přísluší do jmenného prostoru pro kód.

3.6.2 Operátory

Tvorba výrazů v novém jazyce je umožněna pomocí operátorů. Existují různé druhy operátorů:

- aritmetické
- přiřazovací
- bitové
- relační
- operátory pro inkrementaci/dekrementaci
- logické
- řetězcové

Protože v novém jazyce je formátem pro ukládání XML, odpadá nutnost specifikace priority operátorů – pořadí vyhodnocení je totiž dáno strukturou zdrojového kódu, kterého formu při přenesení do dnešních způsobů zápisů jazyků možno označit za povinné závorkování.

Použití operátorů v kódu příslušející metodě třídy:

```
<math:sqrt>
  <math:sum>
    <math:sqr><property>a</property></math:sqr>
    <math:sqr><property>b</property></math:sqr>
  </math:sum>
</math:sqrt>
```

Z uvedeného zdrojového kódu jsou zřejmé dvě věci – za prvé, že operátory jsou rozděleny také do různých jmenných prostorů a za druhé – druhá mocnina a druhá odmocnina se nepovažuje za funkci, ale operátor, víceméně proto, protože jde o všeobecně známou operaci.

Kompletní seznam operátorů a jejich značek nelze prozatím shrnout, takže alespoň pár příkladů různých druhů operátorů. Přiřazení, operátor příslušející jmennému prostoru pro kód se provádí jako:

```
<assign>
  <property>a</property>
  <parameter>b</parameter>
</assign>
```

Bitové operátory (uvedený kód je ekvivalentní výrazu $(input >> 4) \& 0x0F$ v jazyce C):

```
<bit:and>
  <bit:shift direction="right">
    <variable>input</variable>
    <const:integer>4</const:integer>
  </bit:shift>
  <const:integer>15</const:integer>
</bit:and>
```

Relační operátor zapíšeme (výraz je ekvivalent $a > b$):

```
<rel:greater>
  <property>a</property>
  <property>b</property>
</rel:greater>
```

Operátor pro inkrementaci, známý z jazyků podobných C, zapsaný jako $i++$ lze v novém jazyce zapsat jako:

```
<math:postIncrement>
  <variable>i</variable>
</math:postIncrement>
```

Logické operace provádíme pomocí logických operátorů:

```
<log:and>
  <variable>isGreen</variable>
  <variable>isSquare</variable>
</log:and>
```

Řetězcové operátory, jsou založeny na současných metodách pro práci s objekty řetězcového typu a zahrnují tedy zjištění délky řetězce, podřetězec, indexaci, atd. Příklad operátoru:

```
<string:length>
  <parameter>text</parameter>
</string:length>
```

Pro objektově orientovaný návrh jsou k dispozici speciální operátory, příslušející znova do základní sady značek ve jmenném prostoru pro psaní kódu. Vytvoření objektu – instance třídy:

```
<instantiate class="obdélník">  
  <const:integer>16</const:integer>  
  <const:integer>9</const:integer>  
</instantiate>
```

Vytvoření kopie (klonu) objektu:

```
<clone>  
  <parameter>source</parameter>  
</clone>
```

Zrušení objektu

```
<destroy>  
  <variable>source</variable>  
</destroy>
```

Pro doplnění celkového přehledu operátorů musíme uvést ještě operátor pro indexaci:

```
<index>  
  <variable>pole</variable>  
  <variable>index</variable>  
</index>
```

A také operátor pro dereferenci z reference – hodnotou následujícího výrazu je to, na co se reference odkazovala:

```
<dereference>  
  <variable>ref</variable>  
</dereference>
```

Zůstává už jen uvést příklad na přístup do prvku struktury nebo objektu – pro statické typy jako:

```
<element id="a">  
  <variable>Obdélník1</variable>  
</element>
```

Pro dynamický přístup za běhu se používá odlišná konstrukce:

```
<element>  
  <variable>Obdélník1</variable>  
  <const:string>a</const:string>  
</element>
```

Konstantní řetězec je zde uvedený jen ilustrativně, aby byly poslední dvě ukázky zdrojových kódů ekvivalentní.

3.6.3 Řízení běhu

Konstrukce pro řízení běhu můžeme rozdělit na dvě větší skupiny – konstrukce pro větvení a konstrukce pro cykly. Kromě těchto ještě existují speciální řídicí konstrukce – pro návrat z bloku kódu a konstrukce pro návrat hodnoty.

Základní větvicí konstrukcí je všeobecně známé *IF-THEN-ELSE*. V našem novém jazyce má podobu:

```
<if>
:
<then>
:
</then>
<else>
:
</else>
</if>
```

Podmínka se uvádí v rámci značek *if* a kód pro jednotlivé větve v příslušných sekcích.

Dalším způsobem větvení – tentokrát na více než dvě větve – je konstrukce *SWITCH* nebo *CASE* (dle toho jestli známe jazyk *C* nebo *Pascal*). Toto větvení sestává z výrazu, jehož výsledek určuje správnou větví. V uvedených jazycích se tato konstrukce nepatrně liší – zatímco v *Pascal*-u se u *case* nepokračuje v další větvi, u jazyka *C* a jemu podobných (*Java*, *PHP*) se pokračuje a pro ukončení větvení je nutno používat explicitně vyskočení z větve (příkaz *break*). Nový jazyk umí zpracovat oba způsoby – pomocí parametru:

```
<switch implicitBreak="1">
:           výraz
<case>
:           hodnota
:           kód
</case>
:           další <case> sekce
<default>
:           výchozí kód
</default>
</switch>
```

Hodnota uvedeného parametru určuje, zda se pokračuje v další větvi (při hodnotě 0) nebo nepokračuje (parametr musí mít hodnotu 1).

Uvedenou konstrukci pro větvení lze použít také v inverzím režimu, tj. jako výraz uvést konstantní hodnotu a na místě hodnot uvést výrazy. V tomto případě se vyhodnocují jednotlivé výrazy za sebou a chování je ekvivalentem konstrukce *IF-THEN(-ELSEIF-THEN)*-ELSE*.

Co se týče cyklů, v novém programovacím jazyce je možno zapsat jak základní tři formy cyklů jazyků podobných C, tak některé další formy, zejména tu z jazyka *Pascal* a pak je zde ještě konstrukce obecné opakování kódu.

Tři základní cykly jsou *WHILE*, *DO-WHILE* a *FOR* a zapisují se jako:

```
<while check="before">
:           podmínka
:           kód
</while>
```

```
<while check="after">
:           podmínka
:           kód
</while>
```

```
<for>
  <initialize>
  :
  </initialize>
  <preCondition>
  :
  </preCondition>
  <afterIteration>
  :
  </afterIteration>
  :
</for>
```

Forma známá z *Pascal-u*:

```
<count>
  <variable> .. </variable>
  <from> .. </from>
  <to> .. </to>
  <step> .. </step>
</count>
```

A zde je zápis slíbené obecné formy pro konstantní opakování:

```
<repeat count="..">
:
</repeat>
```

Obecné opakování je exemplárním prvkem sémantického opakování, protože říká, co se má provést v sémantické rovině, ale neříká jak. Implementace tedy může být libovolná za dodržení podmínky, že se kód zopakuje v přesně určeném počtu.

Mezi cykly pak také patří procházení přes prvky polí nebo seznamů (resp. kolekcí, které nejsou v této práci ale zahrnuty).

```
<foreach>
  <source>
    :                pole, seznam, kolekce
  </source>
    :                kód
</foreach>
```

V cyklech, které jsou tvořeny značkami *count* a *foreach* je možnost odkazovat se na aktuální prvek cyklu pomocí následovní konstrukce:

```
<current />
```

Dále existují řídicí konstrukce, které mají svoje opodstatnění jen v cyklech – jako je příklad pro další iteraci:

```
<next />
```

nebo příkaz pro zopakování aktuální iterace:

```
<restart />
```

Pro cykly a zároveň i větvení existuje možnost explicitního ukončení provádění příkladu:

```
<break />
```

Posledními řídicími konstrukcemi jsou konstrukce pro ukončení bloku kódu nebo funkce. Blok kódu, který není funkcí lze ukončit pomocí příkazu, který se zapisuje jako:

```
<exit />
```

Blokem kódu se implicitně rozumí nejbližší hranice v rámci větvení nebo explicitně definovaný blok kódu pomocí značek *code*.

Návrat z funkce je pak možný dvěma způsoby, buď přímo vrácením návratové hodnoty přes příkaz *return*:

```
<return> .. <return>
```

nebo pomocí dvou kroků - přiřazení výsledku do speciálního odkazu zapsaného jako

```
<result />
```

a následným zavoláním ukončení, teď již bez hodnoty:

```
<return />
```

Když nebyla před voláním *return* přiřazena hodnota do výsledku funkce, považuje se to za méně kritickou chybu, pro které se generuje jen varování. V případě zapnutí striktnějších kontrol nebude projekt obsahující uvedenou chybu přeložen.

4 Závěr

Semestrální projekt, jehož výsledkem je tento dokument slouží především na ilustraci možností nově vznikajícího jazyka a tudíž všechny uvedené příklady jsou jenom nejlepším odhadem ideálního zápisu. Pokud ale uvedené zápisy nebudou vhodné a bude zapotřebí detailnějšího značení, je možno takovéto značení přidat k definici, nebo změnit stávající definice a kód, který již byl napsán konvertovat do nové verze jazyka. Protože formát souborů je XML tak tato konverze by měla být jednoduchou operací, např. za pomoci XSLT.

Z důvodu scházející zpětné vazby nejenom od uživatelů (které nový jazyk nemá), ale také z implementační fáze, nebylo možno definovat některé z plánovaných vlastností jako jsou abstraktní datové struktury – seznamy, zásobníky a kolekce. Konkrétní podoba není zatím určena ani pro operátor volání funkce. V objektově orientované části existuje požadavek pro umožnění implementace metody pomocí metody jiné třídy, což je vlastnost vhodná pro zvýšení produktivity, jelikož jde o znovupoužití kódu. Další vlastnosti jako jsou makra a jejich protějšek v sémantickém programování, tzv. enzymy (z *intentional programming*) také jen čekají na svoji definici.

Přínos práce je zejména v tom, že dnes neexistuje programovací jazyk, který by umožňoval větší možnosti abstrakce současně s detailní možností popsání implementace. Můj jazyk k těmto vlastnostem přidává ještě neomezenou rozšiřitelnost, protože zápis je ve formě XML.

Na tuto práci pak bude navazovat můj diplomový projekt který dodefinuje chybějící části a upřesní sporné body. Výsledek diplomového projektu nebude jenom pouhá definice programovacího jazyka po teoretické části, ale také přinese možnost praktického seznámení se tímto novým jazykem pomocí překladu do přeložitelných zdrojových kódů v jazyce C. Po praktickém ověření výhod by mělo následovat vytvoření grafického rozhraní pro tvorbu a úpravu zdrojových kódů.

Literatura a jiné zdroje

- [1] *Intentional programming*. Wikipedia – the free encyclopedia.
Dokument dostupný* na URL
http://en.wikipedia.org/wiki/Intentional_programming

- [2] Microsoft Research. *Intentional programming – promotional video*
Audiovizuální záznam dostupný na URL
<http://www.cse.unsw.edu.au/~cs3141/ip.asf>

- [3] Dmitriev S. *Language Oriented Programming: The Next Programming Paradigm*.
JetBrains onBoard Online Magazine, 11, 2004
Dokument dostupný na URL
<http://www.onboard.jetbrains.com/is1/articles/04/10/lop/>

* Uvedené online zdroje byly v době psaní této práce veřejně dostupné (prosinec 2005)