VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DIPLOMOVÁ PRÁCE

2006

Daniel Rozsnyó

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

SÉMANTICKÉ PROGRAMOVÁNÍ

Vedoucí: Křivka Zbyněk, Ing., UIFS FIT VUT

Obor: Výpočetní technika a informatika

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 diplomovou práci na téma *Sémantické programování* vypracoval samostatně pod vedením Ing. Zbyňka Křivky, s využitím vědomostí získaných na přednáškách a s použitím literárních pramenů, které jsem uvedl v seznamu literatury.

……………………………………..
Daniel Rozsnyó
25. května 2006

Poděkování

Na tomto místě bych chtěl poděkovat panu Ing. Zbyňku Křivkovi za jeho odborné vedení, podnětné připomínky a náměty pro zpracování diplomové práce. Rovněž bych rád poděkoval mé rodině, přátelům a také všem ostatním, kteří v mé práci viděli nějaký smysl, nebo alespoň věřili, že má práce smysl má.

Abstrakt

Rozsnyó, D., *Sémantické programování*. Diplomová práce. Brno, 2006.

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 vyplývajících ze syntaxí použitých programovacích jazyků. Přenositelnost je zajištěna pomocí standardního formátu dat – XML a vlastním návrhem jazyka, který je jak sjednocením, tak zobecněním dnešních jazyků. Pro praktickou aplikaci byl následně vytvořen překladač nového jazyka, sémantické notace, do zdrojových kódů v jazyce C. Tento překladač zahrnuje i několik optimalizací.

Klíčová slova

Programovací jazyk, syntaktický strom, překladač, generování kódu, datové typy, procedurální jazyk, objektově orientované programování, cílené programování, generativní programování, jazykově zaměřené programování (*LOP*), rozšiřitelný vyznačovací jazyk (*XML*).

Abstract

Rozsnyó, D., *Semantic programming*. Diploma thesis. Brno, 2006.

The thesis is about an idea of a new way of source code notation. The proposed notation is semantically cleaner and offers better portability, than the most of the current programming languages. Some of these features are achieved by abstractions which liberate the programmer from solving implementation details. They also help to overcome the limitations arising mostly from the syntaxes of the current programming languages. The question of portability is ensured by using a standard data format (*XML*) and own design of the programming language. This language not only unifies the features of today programming languages, but also generalizes some of them. For the application in praxis, a compiler was created. This compiler basically transforms the constructions from the new language to C language source code, which is then compiled into executables.

Keywords

Programming language, syntactical tree, compiler, code generation, data types, procedural language, object oriented programming, intentional programming, generative programming, language oriented programming (*LOP*), extensible markup language (*XML*).

Obsah

1 Úvod 1

2 Sémantické programování 2

2.1 Myšlenka a její definice ... 2
2.2 Vlastnosti ... 3
2.2.1 Jednotnost ... 3
2.2.2 Jednoznačnost ... 4
2.2.3 Jednoduchost ... 5
2.2.4 Univerzálnost ... 6
2.2.5 Variabilnost ... 7
2.2.6 Abstrakce ... 8
2.2.7 Přenos nativní sémantické informace ... 8
2.3 Aplikace v praxi ... 9

3 Návrh sémantické notace 10

3.1 Syntaxe ... 10
3.1.1 Globální identifikátor ... 10
3.1.2 Domény a jmenné prostory ... 11
3.1.3 Značky ... 11
3.1.4 Hello world! ... 12
3.2 Elementární datové typy ... 13
3.2.1 Pravdivostní hodnota ... 13
3.2.2 Celé číslo ... 13
3.2.3 Číslo s pohyblivou řádovou čárkou ... 14
3.2.4 Komplexní číslo ... 14
3.2.5 Znak ... 15
3.2.6 Řetězec ... 16
3.2.7 Výčet ... 17
3.2.8 Speciální typy ... 18
3.3 Typové operátory ... 18
3.3.1 Volitelná hodnota ... 19
3.3.2 Struktura ... 20
3.3.3 Pole ... 21
3.3.4 Množina ... 21
3.3.5 Reference ... 22
3.3.6 Funkce ... 22

3.4 Příkazy a operátory ... 23
3.4.1 Organizační pravidla ... 23
3.4.2 Definice funkce ... 24
3.4.3 Návrat z funkce ... 25
3.4.4 Příkaz větvení ... 25
3.4.5 Příkazy cyklu ... 26
3.4.6 Příkaz výstupu ... 27
3.4.7 Logické operátory ... 28
3.4.8 Relační operátory ... 29
3.4.9 Aritmetické operátory ... 30
3.4.10 Volání funkce ... 31
3.4.11 Indexace polí ... 32
3.4.12 Přístup k prvkům struktury ... 32
3.4.13 Práce s komplexními čísly ... 33
3.4.14 Definice a používání proměnných ... 34

4 Realizace překladače **35**

4.1 Vnější prostředí ... 36
4.2 Vnitřní struktura ... 36
4.3 Obecné vlastnosti ... 37
4.3.1 Podporované prvky sémantického programování ... 37
4.3.2 Simultánní překlad ... 38
4.4 Volby překladače ... 38
4.5 Implementované optimalizace ... 39
4.5.1 Vyhodnocování konstantních výrazů ... 39
4.5.2 Eliminace mrtvého kódu ... 40
4.5.3 Rozvinutí smyček ... 41
4.5.4 Spojení příkazů výstupu ... 42

5 Závěr **43**

5.1 Výhody ... 43
5.2 Obecné nevýhody ... 43
5.3 Chyby v návrhu a jiné nedostatky ... 44
5.4 Budoucí vývoj ... 45

Seznam použité literatury **46**

Seznam obrázků

1. Stejný algoritmus, ale různá syntaxe 2
2. Sjednocení jazyků 3
3. Použití metadat pro zvýšení jednoznačnosti 4
4. Zápis číselných konstant 6
5. Zjednodušení statických regulárních výrazů 6
6. Analogie k variabilnosti 7
7. Nasazení sémantického programování v praxi 9
8. Přehled všech datových typů 18
9. Způsoby implementace typového operátoru volitelná hodnota 19
10. Příklad struktury 20
11. Ilustrace reference 22
12. Rozhraní a implementace funkce 24
13. Vytvoření spustitelné aplikace 35
14. Vnitřní struktura překladače 36

1 Úvod

Programování bylo mým koníčkem již od základní školy a jak čas plynul, vyměnil jsem několik počítačů, programovacích jazyků i vývojových prostředí. Bohužel to dospělo do stádia, kdy žádné prostředí nebo jazyk nedokáže splnit moje požadavky týkající se jednoduchosti a síly. Z tohoto důvodu tedy vznikla myšlenka navrhnout si vlastní jazyk a příslušné vývojové prostředí, které bude v budoucnu pro programátora pohodlnějším nástrojem, než jsou ty dnešní. Motivací byl také směr nazvaný *Intentional programming*, jehož bližší popis je k nalezení v [1] a donedávna existovalo i demonstrační video, [7].

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 a nástroj pro složitější projekty jménem integrované vývojové prostředí (IDE – *Integrated Development Environment*). Přestože existuje několik takovýchto nástrojů, situace není ideální. Volba jazyka závisí většinou na druhu projektu (jiný pro web, jiný pro operační systémy) a pokud už daný jazyk známe, potřebujeme znát také vlastní vývojové prostředí nebo nástroje (angl. *toolchain*) kvůli efektivnímu programování. To ale pořád není vše, protože je nutno znát prostředí, ve kterém program poběží a dostupné knihovny funkcí. 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 mého semestrálního projektu a diplomové práce. Opak je rozebírán v [2], kde je cíl (fungující program dle zadání projektu) dosažen za pomocí programování v specializovaných jednoúčelových jazycích vytvořených právě pro daný projekt.

Součástí semestrálního projektu byla analýza situace se současnými programovacími jazyky a důvody, které vedou k potřebě vytvořit další, společný, programovací jazyk. Zde byl prezentován návrh jazyka, který ale nerespektoval podstatu inkrementálního vývoje – byla to pouhá rozsáhlá definice. Z tohoto důvodu byl tedy realizován experimentální překladač nového jazyka a také identifikovány vlastnosti, kterými se vyznačuje sémantické programování.

První kapitola této diplomové práce, *Sémantické programování*, popisuje vlastnosti nového jazyka po teoretické stránce. V další kapitole, s názvem *Návrh sémantické notace,* se přechází od teoretických vlastností k praktickým prvkům – zejména je zde vyčerpávající popis programových konstrukcí použitelných při tvorbě programů v tomto novém jazyce. Poslední kapitola, *Realizace překladače*, popisuje implementaci překladače ze sémantické notace do jazyka C, ze kterého lze následně dalším překladem získat spustitelnou aplikaci.

Práce si za vedlejší cíl klade také ukázat, že má smysl tvořit programy i jinak, než způsobem, který je standardně zažitý, plynoucí buďto z pohodlnosti, nebo jako důsledek vlivu marketingu a reálného prostředí, založeném výhradně na komerci. Dalším cílem, již nad rámec této práce, bude realizace kompletního vývojového prostředí – ideálního, otevřeného a bez všech omezení způsobených vlivem komerce a neochoty zkusit *to dělat* jinak než ostatní.

2 Sémantické programování

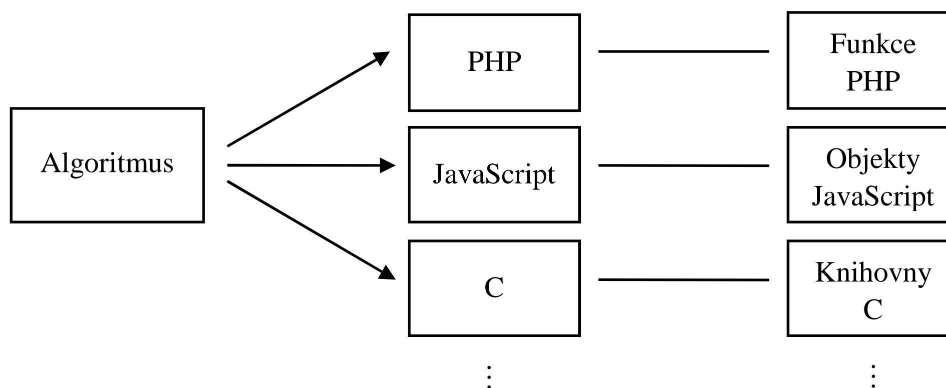
2.1 Myšlenka a její definice

Myšlenka sémantického programování vznikla v průběhu práce na různých projektech používajících různé programovací jazyky (zejména jde o jazyky C, C++, jazyk symbolických instrukcí a skriptovací jazyky *PHP*, *JavaScript*, *bash*) a zjednodušené notace jako jsou regulární výrazy a dotazy SQL. V některých projektech se opakovala nutnost použít stejných algoritmů ve dvou různých prostředích. Jelikož byly použitelné programovací jazyky pro tyto prostředí různé, přímé nasazení stávajícího kódu nebylo možné a nastala tedy fáze přepisování a překládání zdrojových kódů – např. z PHP, které generovalo stránky dynamicky na požádání (*http* dotaz) do jazyka JavaScript, který dokázal informace aktualizovat na straně klienta bez nutnosti komunikace se serverem. Při změně algoritmu v rámci inkrementálního vývoje bylo potřeba změněné části přeložit znova ručně .

Vlastnosti, které zde byly na obtíž, byly identifikovány jako syntaxe a knihovní funkce použitých skriptovacích jazyků. Výpočetní algoritmus byl stejný, nezávisle na místě spuštění či prostředí (webový server resp. klient). Pro zlepšení dané situace by bylo vhodné tyto vlastnosti alespoň z části eliminovat. A tento požadavek dal vzniknout sémantickému programování.

Když tedy chceme identifikovat základní myšlenku sémantického programování, bude jednodušší nahlédnout nejprve na obrázek 1. Zde je zobrazena situace, kdy přestože existuje jediný společný algoritmus na řešení konkrétního problému, jeho implementace v různých jazycích je rozdílná a to hlavně v syntaxi zdrojového kódu. Pak u různých jazyků nalezneme stejnou funkcionalitu na různých místech (funkce, rozšíření, objekty, knihovny apod.).

Tudíž základní myšlenkou sémantického programování je společný zápis programů a algoritmů – nezávisle na použitém programovacím jazyku a prostředí (ve významu dostupných knihoven funkcí).



Obr. 1: *Stejný algoritmus, ale různá syntaxe*

2.2 Vlastnosti

Vlastnosti, které má sémantické programování, nevyplývají jenom z toho, že se vytvoří společný zápis všech populárních procedurálních jazyků, ale také z toho, že byla šance vytvořit něco od základu. Tímto přístupem je možno dosáhnout obsažení vlastností, kterými současné programovací jazyky nedisponují. Také se lze vyvarovat chybám nebo omezením v návrzích jazyků, které plynou z dob, ve kterých byly tyto jazyky navrhnuty.

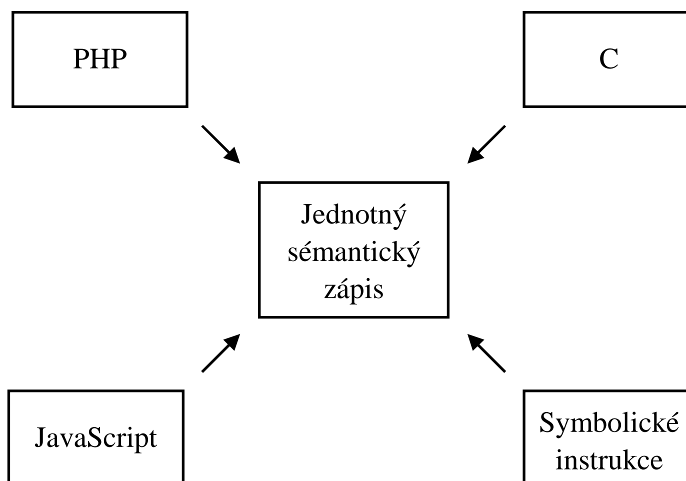
2.2.1 Jednotnost

Některé konstrukce jsou ve stávajících programovacích jazycích syntakticky málo odlišné, přesto se tyto detaily musí naučit každý programátor. Jako vhodný příklad poslouží řádek kódu s komentářem:

```
printf( "Hello world" );          /* C */

write( 'Hello world' );           { Pascal }

echo 'Hello world';               // PHP
echo "Hello world";               // PHP, alternativní zápis
```

Všechny ukázky jsou sémanticky shodné, dosáhne se totiž stejného efektu. Rozdíly jsou jen v použité syntaxi, kde jsou patrné hned dva rozdíly. První ve způsobu zápisu řetězcových konstant a druhý ve způsobu zápisu komentářů. Tyto rozdíly jsou nepraktické pro programátora a z hlediska překladače nemají veliký význam (s výjimkou PHP, které obsahuje obyčejný řetězec v apostrofech a řetězec s proměnnými v uvozovkách).

Jelikož je sémantické programování založeno hlavně na sjednocení, výsledkem bude jediný způsob zápisu daného zdrojového kódu (ilustrováno na obr. 2).



Obr. 2: *Sjednocení jazyků*

2.2.2 Jednoznačnost

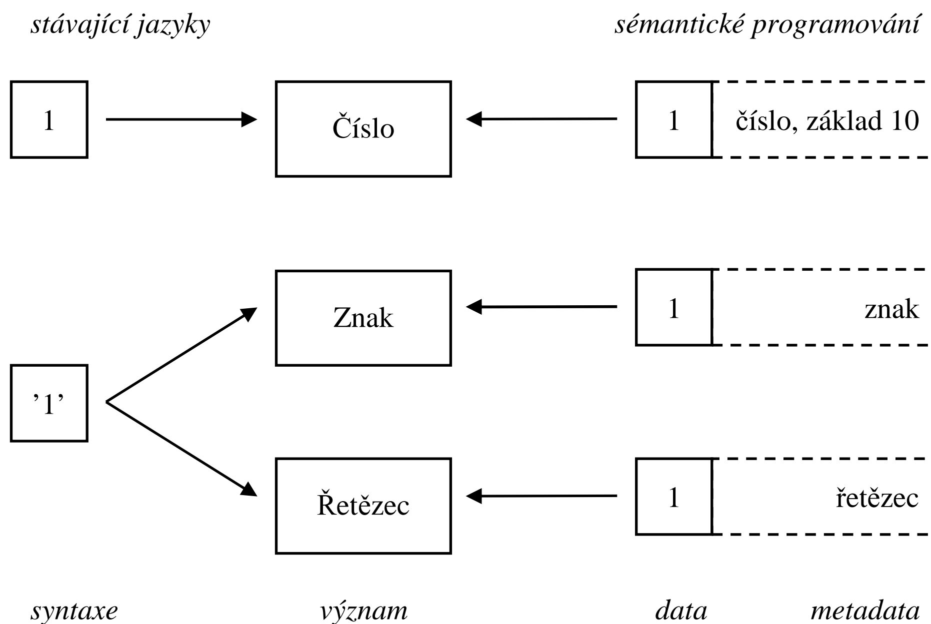
Problém nejednoznačnosti syntaxe existuje jednak z požadavku udržení jednoduché syntaxe a pak také z důvodu složitého rozšiřování syntaxe zaběhnutého jazyka. V praxi se tak můžeme setkat např. se zápisy:

```
1                                    { číslo   }
'1'                                  { znak    }
'1'                                  { řetězec }
```

Původem jsou tyto fragmenty ze staršího programovacího jazyka Pascal, ale obdobný problém existuje i v současném jazyku PHP. Tímto problémem je nejednoznačnost při určení, zda se jedná o konstantu typu znak nebo řetězec.

V sémantickém programování se takovéto nejednoznačnosti nepřipouští. Typ dat se nesmí dodatečně určovat, i když by to bylo možné pomocí heuristiky nebo dle kontextu. Příslušný typ dat se zkrátka musí znát předem. Tento požadavek souvisí se silně typovaným přístupem.

Metodou, kterou se v sémantickém programování řeší problém jednoznačnosti, je důsledné označování (angl. *tagging*) fragmentů zdrojového kódu pomocí *metadat*. Vše je ilustrováno na obr. 3, kde si lze také všimnout, že mezi metadata patří i informace o základu číselné soustavy. U znaků a řetězců může být také naznačení kódové stránky a aktuální stav ošetření speciálních znaků (angl. *character escaping*), míněno jako pořadí provedení daných ošetření.



Obr. 3: *Použití metadat pro zvýšení jednoznačnosti*

2.2.3 Jednoduchost

Programování nikdy nebylo a ani nebude jednoduché, přesto jsou některé konstrukce až zbytečně složité, např. pro zápis číselných konstant v různých číselných soustavách se používají speciální sekvence. Například jazyk C poskytuje následující možnosti:

```
0101                                    /* osmičkově    */

101                                     /* desítkově    */

0x101                                   /* šestnáctkově */
```

Pomineme-li, že jazyk C neposkytuje možnost zápisu binárních čísel, přestože se jedná o jazyk určený pro systémové programování, jsou pravidla pro zápis v různých soustavách složitá a nejednotná (osmičková: číslo začíná nulou, desítková: číslo nezačíná nulou, šestnáctková: číslo začíná nulou a znakem x).

Zjednodušení se v sémantickém programování dosáhne tím, že kromě uvedení čísla (ve tvaru řetězce) se uvede také *základ číselné soustavy*. Tímto se přibližujeme rozumnější formě zápisu, který používají zejména matematici – 101_8, 101_{10}, 101_{16}.

Jiným příkladem je ošetření speciálních znaků, které se používá při spojování dvou různých syntaktických domén. Obecně je můžeme označit jako datová a řídící doména. Nejjednodušším příkladem je zápis řetězců, které ve svých datech obsahují znak ukončení řetězce z řídící domény:

```
"Následující text je \"v uvozovkách\""        /* C */

'Znak apostrofu ('') se zdvojuje'             { Pascal }
```

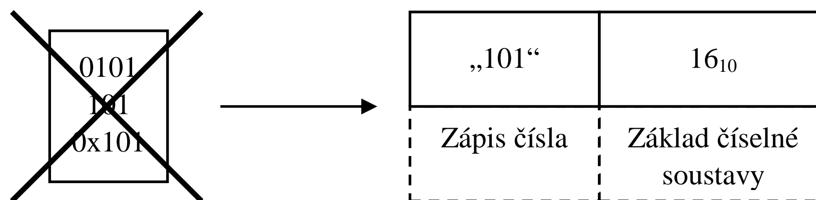
V jazyce C se používá zpětné lomítko na uvedení speciální sekvence znaků. Obsažení znaku uvozovek v datové doméně dosáhneme pomocí sekvence zpětného lomítka a uvozovek. V jazyce Pascal neexistují speciální sekvence a vložení znaku apostrofu se dosáhne jeho zdvojením.

V jednoduchých zdrojových kódech je uvedený princip rozumně použitelný, ale jen do té doby, než vznikne požadavek na vrstvení syntaxí (vícenásobné aplikace ošetření řetězce na speciální znaky). Takový požadavek nastává např. při tvorbě příkazu vykonaném pomocí systémového volání, zápisu SQL dotazů, zápisu regulárních výrazů, zpracování parametrů po přenosu různými protokoly (http, smtp, …), použití značkovacích jazyků (XML, HTML, …) atd. Praxe ukazuje, že v případě, kdy nastane vrstvení, mnoho programátorů selhává a uvolňuje potencionálně nebezpečné aplikace. Jedním z těchto problémů je *SQL-injection*, které souvisí právě se špatným uvozením dat a dává možnost útočníkovi provádět libovolné příkazy, přestože by měly být zpracovány jen jako data.

V sémantickém programování lze problém řešit jednoduše sadou atributů označujících ošetření speciálních znaků. Například při vkládání dat do databáze může překladač rozhodnout o automatickém ošetření těchto dat na základě informace, zda se jedná o řídící část SQL dotazu nebo jeho datovou část.

2.2.4 Univerzálnost

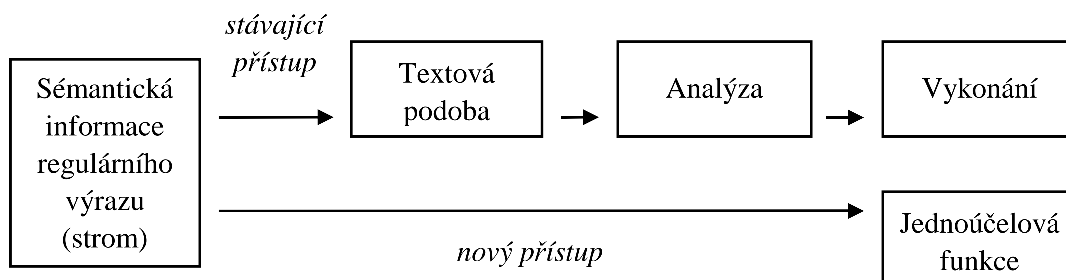
Výše uvedené řešení problému se zápisem konstant v různých číselných soustavách se zavedením pojmu základu číselné soustavy nejen zjednodušil, ale také poskytuje více – univerzálnost. Tato vlastnost vyplynula z možnosti uvést za základ číselné soustavy jakékoliv číslo – nejsme omezeni jenom na 8, 10 a 16 – můžeme používat soustavu o libovolném základu.



Obr. 4: *Zápis číselných konstant*

Sémantické programování jako přístup k řešení úkolu ale není omezeno jenom na zápis algoritmů. Předpokládá se zápis i jiných sémantických informací – např. regulárních výrazů. Regulární výrazy jsou jasně definovány několika pravidly, ale při použití v programátorské praxi se naráží na obtížnost zápisu pravidel a znova se objevuje problém řídící a datové domény. Výsledkem je určitý kompromis – několik znaků je speciálních (tečka, hvězdička, otazník, plus, hranaté závorky, stříška, dolar) a vyjadřují určitá pravidla.

Když ale zanalyzujeme regulární výraz, můžeme ho zapsat také jinak – pomocí stromu vyjadřujícího skladbu regulárního výrazu. S touto stromovou reprezentací pak můžeme pracovat již bez problému s uvozováním speciálních znaků, protože se skládá z uzlů, které přísluší buď datové nebo řídící doméně. Závěrem je, že z regulárních výrazů můžeme jejich podstatu zapsat i jinak, než zažitou formou a tím dosáhnout lepších vlastností. Naskytuje se zde možnost algoritmické tvorby jednoúčelové funkce (viz obr. 5) odpovídající statickému regulárnímu výrazu, která bude mít několikanásobně vyšší výkon (rychlost) v porovnání s dynamicky zpracovávaným regulárním výrazem (pomocí knihovních funkcí), a to jednak z důvodu optimalizace a pak také proto, že odpadne určitá fáze analýzy výrazu zadaného v podobě textu.



Obr. 5: *Zjednodušení statických regulárních výrazů*

2.2.5 Variabilnost

Tato vlastnost souvisí s možností použití alternativních konstrukcí. Programátor si pak nemusí pamatovat tu jedinou správnou, když existují další, zcela ekvivalentní konstrukce a jsou pro konkrétní použití vhodnější.

Příkladem může být zápis polí, kde je programátor omezen jediným způsobem:

```
unsigned char ip[4];                /* C */
```

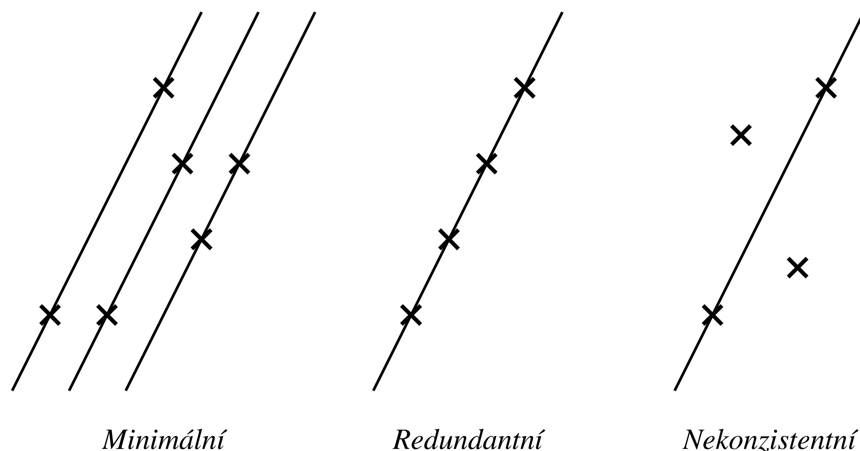
Některé jazyky, jako např. jazyk Pascal, připouští definovat první a poslední index:

```
var ip: array[0..3] of byte;        { Pascal }
```

V sémantickém programování neexistují tyto pevné formy a pole se zapisuje volněji, a to pomocí kombinace alespoň dvou parametrů ze tří, kterými lze určit pole – *první index*, *poslední index* a *velikost pole*. Protože existuje výchozí hodnota parametru *první index* (což je nula), lze také definovat pole použitím pouze jednoho parametru (*poslední index*, *velikost pole*).

Variabilnost je také použita při definici celočíselných typů – někdo potřebuje definovat číselný rozsah hodnot, které může proměnná daného typu mít a jiní raději specifikují počet bitů a znaménko. Vhodnost konkrétního způsobu definice závisí totiž na problému – první způsob je obecně vhodnější pro psaní algoritmů, druhý pro psaní systémových součástí závislých na pevně určené hardwarové architektuře.

Na obr. 6 je ilustrativně pomocí přímky zobrazena analogie k variabilnosti v sémantickém programování. Jak víme, přímku lze jednoznačně definovat dvěma body, když je bodů více, tak musí vykazovat určitou konzistenci. V opačném případě je definice neplatná. Stavy nekonzistentnosti se dají detekovat v sémantickém programování pomocí kontrolních algoritmů, obdobně jako existují vzorce pro uvedený příklad s přímkou.



Obr. 6: *Analogie k variabilnosti*

2.2.6 Abstrakce

V dnešních jazycích se můžeme setkat jen se specifickými typy – např. u čísel jsou to celá čísla (angl. *integers*) a čísla s pohyblivou řádovou čárkou (angl. *floating point numbers*). V jazyce C se mezi nimi rozlišuje za pomocí existence desetinné tečky:

```
2006                    /* C – celé číslo            */
2006.0                  /* C – pohyblivá řádová čárka */
```

V sémantickém programování se na rozdíl od těchto zápisů uvažuje o číslech jinak. Existují zde jak čísla celá, tak s pohyblivou řádovou čárkou. Kromě těchto dvou existuje typ pro racionální čísla (vyjádřitelné zlomkem, jako podíl dvou celých čísel) a komplexní čísla (vektor se dvěmi složkami – s reálnou a imaginární částí). Dále existuje abstraktní typ – obecné číslo.

K číslům může v závislosti na významu příslušet řada atributů. Můžeme mít čísla, která vyjadřují datum a čas ve formátu *Unix timestamp*, nebo čísla vyjadřující měnu, která jsou automaticky formátována při výstupu. Atributem je tedy také výchozí formátování čísla při výstupu (základ číselné soustavy, počet celých a desetinných míst, doplnění nulami, zarovnání).

Abstrakce, tak jak zde byla popsána může vzbuzovat dojem, že implementace je nutně na základě objektového přístupu, kde existuje dědičnost a hierarchie. To však není pravdou. Čísla nemusí být objekty, definice v rámci sémantického programování to ani nevyžaduje.

Dalším typem abstrakce v sémantickém programování je příkaz opakování dle určeného počtu cyklů. Ve stávajících jazycích není obdobný příkaz, ale je potřeba definovat proměnnou (dostatečné velikosti), která bude sloužit jako počítadlo a pak napsat příkaz cyklu s porovnáním počítadla a jeho dekrementací nebo inkrementací. Tento přístup je příliš spjat se syntaxí jazyka a může vnést zbytečné chyby. A to jak v přesném počítání, tak ve zcela špatném případně při neoptimální volbě typu pro počítadlo. V sémantickém programování byl tedy zaveden příkaz pro opakování, který zaručí přenos nativní sémantické informace (tj. vlastního opakování) bez zbytečné degradace implementací a syntaxí.

2.2.7 Přenos nativní sémantické informace

Jak ukazoval i předchozí příklad související s abstrakcí u opakování, v sémantickém programování se klade důraz na informace relevantní k algoritmu a vypouští se implementační detaily, které nejsou důležité pro daný algoritmus nebo můžou dokonce být v jeho neprospěch.

Některé detaily, které byly vypuštěny při zápisu algoritmu se dají při překladu algoritmicky dopočítat a tím zaručit optimální implementaci, ale také existují detaily, které nelze vypustit. Například u volitelných datových typů (proměnná obsahuje hodnotu daného typu, nebo neobsahuje nic), které lze implementovat více způsoby (např. pomocí ukazatele nebo pomocí příznaku) přijde vhod možnost naznačení lepšího způsobu implementace (angl. *hint*).

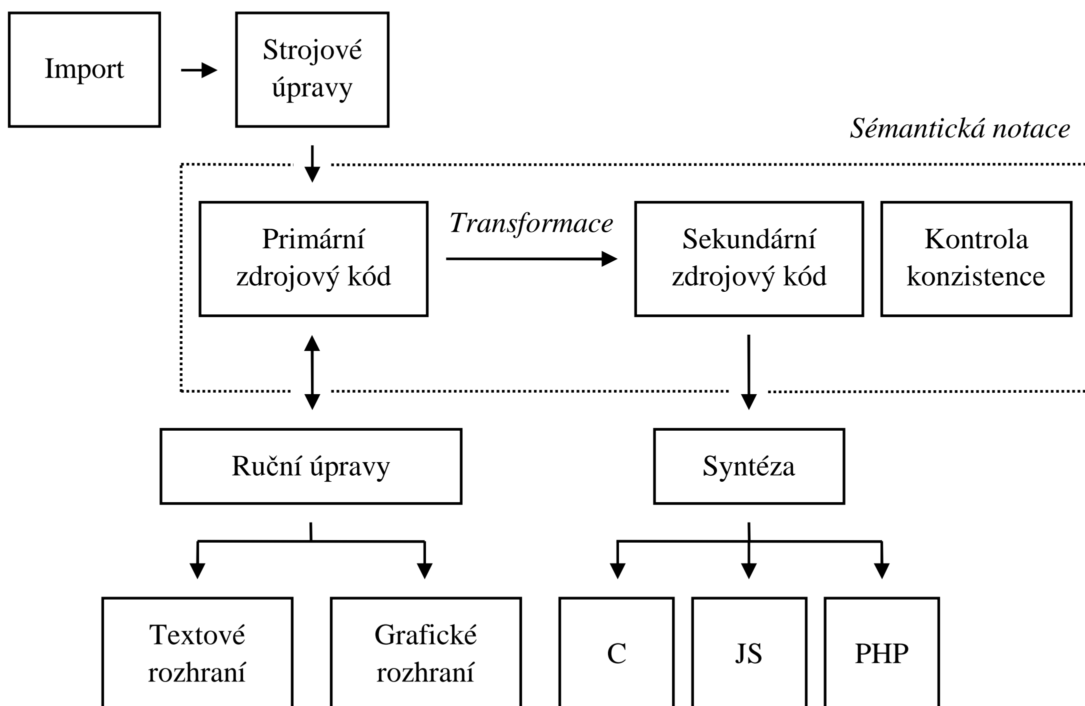
Hlavní rozdíl plynoucí z této vlastnosti je nejvíce viditelný u číselných typů, kde se sémantické programování neuspokojí s několika klíčovými slovy, ale vyžaduje specifikovat číselný typ např. rozsahem hodnot nebo počtem bitů a jestli se povoluje znaménko.

2.3 Aplikace v praxi

Jednu z možných způsobů aplikace naznačuje obr. 7. Základem je sémantická notace, která je uvnitř rozčleněna na primární a sekundární kód. Primární je ten, který se přímo edituje a sekundární je z něj odvozený, např. pomocí transformačních pravidel. Když je notace ve formě XML, můžeme za transformaci považovat provedení XSLT nebo jiného podpůrného skriptu pro zjednodušení psaní programů. Primární zdrojový kód je ale potřeba nejdříve vytvořit nebo získat. K tomu vedou dvě cesty. Buď se tato notace vytvoří importováním stávajícího kódu, nebo se vytvoří ručně za pomoci textového nebo grafického uživatelského rozhraní.

U ruční editace je důležité včasné varování o chybách. O tyto varování se stará modul pro kontrolu konzistentnosti běžící na pozadí, nepřetržitě hlídající zdrojové kódy. Tím, že tato kontrola není prováděna jen v době překladu, jak je tomu u současných jazyků, získáváme několik výhod. Např. hlášení o chybě se objeví relativně v krátkém časovém odstupu od vytvoření chyby, nebo rozprostření zatížení procesoru v čase. S tímto pak souvisí jak větší výkonnost uvedeného systému na stejné konfiguraci (negenerují se výkonové špičky a mrtvé intervaly), tak možnost použití slabší konfigurace při zachování rozumných reakčních dob.

Když jsou zdrojové kódy zkontrolované a validní, nastává fáze překladu a syntézy. Protože sémantická notace není přímo použitelná, je potřeba z ní vytvořit zápis jiný – např. vygenerovat některý z jazyků kompilovaných nebo používaných jako webové skripty. Tato fáze může probíhat na pozadí v součinnosti s kontrolou konzistence. Výsledný efekt je pak obdobný jako při použití skriptovacího jazyka – není potřeba čekat na zdlouhavý překlad projektu.



Obr. 7: *Nasazení sémantického programování v praxi*

3 Návrh sémantické notace

Celkový návrh sémantické notace byl ovlivněn procedurálními jazyky, protože kombinují vysokou míru efektivity s jednoduchostí tvorby programů. Vhodným směrem by bylo také objektově orientované programování, které ale z důvodu komplexnosti není prozatím zcela prozkoumáno ve vztahu se sémantickým programováním.

3.1 Syntaxe

Každá notace, i když sémantická, musí mít nějakou syntaxi, protože ji je potřeba ukládat, přenášet, archivovat atd. Lepší pojem než syntaxe je ale formát souborů (při ukládání) nebo typ serializace dat (při přenosu). Volba v případě sémantické notace padla na XML (viz. [4]), který splňuje podmínky kladené na formát souborů nejlépe, protože:

- je nezávislý na platformě
- je standardizován, celé znění standardu je volně dostupné (viz. [5])
- není zatížen licenčními poplatky
- je univerzální, jednoduše rozšiřitelný
- je vhodný pro strojové i ruční úpravy
- obsahuje metadata, kterými popisuje sám sebe

Dalšími kandidáty byly textové formáty se syntaxí podobnou současným jazykům nebo binární soubory. V prvním případě je problematické rozšiřování, protože je nutnost přepisovat syntaktický analyzátor a gramatiku a také není jednoduché strojově upravovat takové soubory. Binární formáty byly zavrhnuty z důvodu problematické ruční editace a obnovy po poškození souboru.

3.1.1 Globální identifikátor

Syntaxe souvisí hlavně s pravidly o tom, jak se zdrojový kód píše. V sémantické notaci, jak je zde navrhována, existují fragmenty, které lze jedinečně identifikovat pomocí globálně jednoznačného identifikátoru (angl. *GUID – globally unique identifier*). Fragmenty s různým identifikátorem se nacházejí v oddělených souborech, což ve výsledku zjednodušuje jejich správu a následné použití při editaci, kontrolách a překladu.

Identifikátor fragmentu není jen globálně jednoznačný, má také jiné vlastnosti. Jednou z těchto vlastností je, že z identifikátoru lze určit typ fragmentu (jaká data identifikuje – typ, funkci, program). Jinou, neméně důležitou vlastností, je hierarchická organizace identifikátorů. Soubory s notací se totiž nachází na disku v adresářích, které jsou hierarchické a zodpovídají logické struktuře aplikace nebo řešení. Toto uspořádání se tedy používá i pro identifikátory – ve zdrojových souborech se pak můžou vyskytovat odkazy jak lokální (ve stejném adresáři), tak relativní (adresáře výše nebo na stejné úrovni) a dokonce i absolutní (identifikátor uvedený celou cestou).

Hierarchická organizace identifikátorů je použita ve smyslu jmenných prostorů (používaných např. v jazyce C++).

3.1.2 Domény a jmenné prostory

Data, která chceme ukládat, jsou z několika domén. V prvotním návrhu sémantické notace byly identifikovány tyto tři domény:

- typová
- hodnotová
- kódová

V typové doméně se nacházejí elementární datové typy (číslo, znak, řetězec, atd.) a typové operátory sloužící k vytvoření komplexnějších datových typů (pole, struktura). Doména hodnot, nebo také konstant pak definuje zápis výchozích hodnot, nebo konstantních výrazů. Poslední kódová doména slouží pro organizaci a zápis vlastního kódu – definice programů, funkcí, příkazů, výrazů, operátorů.

Každé z těchto domén přísluší jedinečný jmenný prostor v XML, takže jejich použití v případě smíšené notace je zjednodušeno (např. při specifikování vlastních typů a výchozích hodnot proměnných, nebo použití konstantních výrazů v kódu). Dále bude pro zkrácení zápisu v příkladech zdrojových kódů vynechána definice jmenného prostoru a bude použit u značek jen příslušný prefix (*type*, *const*, *code*).

3.1.3 Značky

Názvy značek jsou odvozeny od fragmentů (programových konstrukcí, datových typů a jiných prvků jazyka), které je potřeba zapisovat. Většina fragmentů je zapsatelná pomocí jedné značky, která disponuje sadou parametrů a vlastní hodnotou (např. elementární typy). Jiné fragmenty jsou zapsané pomocí více značek – např. u typu funkce jsou navíc značky, které slouží pro identifikaci sekce parametrů a návratové hodnoty. Těmto značkám lze říkat také organizační a slouží na vytvoření jednoznačné struktury fragmentu. Organizační značky jsou kontextové a jejich význam je určen v kontextu, ve kterém jsou použity. V případě XML to znamená závislost názvu nadřazeného elementu. Tato kontextová závislost je jenom na základní úrovni a jednoznačný význam lze tedy určit za pomocí přímo nadřazeného uzlu.

Pro ulehčení psaní programů v sémantické notaci je povoleno používání alternativních názvů značek (angl. *alias*). Převážně se jedná o jednu nebo více zkrácených forem. Aplikace alternativních názvů (konverze na standardní název) se provádí v první fázi překladu, při načítání zdrojového souboru. Několik příkladů:

Kontext:	Zkrácený tvar:	Standardní název:
	type:func	type:function
type:function	type:ret	type:returns
	code:ret	code:return
	code:mul	code:arithmeticMultiplication
	code:curr	code:iterationValue
	code:current	code:iterationValue
	code:arg	code:argumentValue

3.1.4 Hello world!

U popisu programovacích jazyků je zvykem uvádět příklad aplikace, která vypíše řetězec „Hello *World!*“. Uvedené zadání v sémantické notaci splňuje následující zdrojový kód:

```
<?xml version="1.0" encoding="UTF-8"?>
<program id="helloworld" type="/application/cmdline"
  xmlns="http://rozsnyo.com/mdfs/code"
  xmlns:const="http://rozsnyo.com/mdfs/const">

    <body>

        <write>
            <const:string>Hello World!</const:string>
            <const:newline />
        </write>

    </body>

</program>
```

Tento jednoduchý program je typu */application/cmdline*. Definice tohoto speciálního typu určujícího prostředí, ve kterém program běží je uložena v adresáři *application* v libovolně pojmenovaném souboru jako:

```
<?xml version="1.0" encoding="utf-8"?>
<typedef id="cmdline"
  xmlns="http://rozsnyo.com/mdfs/type">

    <!-- int main(int argc, char *argv[]); -->

    <function>

        <parameters>

            <parameter id="argc">
                <integer />
            </parameter>

            <parameter id="argv">
                <array min="0" hint="index:unlimited">
                    <string />
                </array>
            </parameter>

        </parameters>

        <returns>
            <integer />
        </returns>

    </function>

</typedef>
```

Jak je vidět, sémantická notace ve formě XML je jednoduše čitelný text. Lze říci, že každý programátor je schopen tuto notaci i bez předchozího seznámení pochopit. Následující kapitola se proto věnuje konstrukcím použitelných při tvorbě programů v sémantické notaci.

3.2 Elementární datové typy

3.2.1 Pravdivostní hodnota

Prvním elementárním datovým typem je typ *boolean* vyjadřující pravdivostní hodnotu. Jeho zápis je jednoduchý:

```
<type:boolean />
```

Do proměnné tohoto typu lze uložit dvě různé hodnoty:

```
<const:false />

<const:true />
```

3.2.2 Celé číslo

Dalším elementárním typem jsou celočíselné typy, které lze definovat bez bližší specifikace pomocí značky:

```
<type:integer />
```

Když ale potřebujeme mít jistotu, že typ bude tak veliký, jak potřebujeme, můžeme ho definovat pomocí několika ekvivalentních zápisů. Buď jako rozsah:

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

nebo pomocí počtu bitů a znaménka:

```
<type:integer bits="8" sign="none" />
```

Parametr pro znaménko může nabývat tří hodnot – znaménko neexistuje (uvedený příklad), znaménko je v rámci a znaménko je nad rámec uvedeného počtu bitů.

Hodnoty jsou tvořeny předdefinovanými konstantami

```
<const:zero />

<const:one  />
```

Nebo je lze vytvořit z řetězce:

```
<const:integer>2006</const:integer>
```

Jak již bylo naznačeno, můžeme použít libovolnou číselnou soustavu:

```
<const:integer base="16">FADE</const:integer>
```

3.2.3 Číslo s pohyblivou řádovou čárkou

Dalším číselným datovým typem jsou čísla s pohyblivou řádovou čárkou, jejichž zápis je také jednoduchý a bez bližší specifikace je:

```
<type:float />
```

Specifikovat tento typ lze znova více způsoby. První z nich je definování části mantisy a exponentu zvlášť, pomocí počtu bitů a znaménka. Jiný, jednodušší způsob, je definování počtu bitů celého čísla podle standardu IEEE 754 (viz. [6]):

```
<type:float ieeeBits="32"  /> <!-- single precision          -->
<type:float ieeeBits="48"  /> <!-- single-extended precision -->
<type:float ieeeBits="64"  /> <!-- double precision          -->
<type:float ieeeBits="80"  /> <!-- double-extended precision -->
<type:float ieeeBits="128" /> <!-- quadruple                 -->
```

Konstanty jsou znova jak uživatelské, tak předdefinované:

```
<const:float>3.1415926</const:float>
<const:Pi />
```

3.2.4 Komplexní číslo

Posledním elementárním číselným typem, se kterým počítá sémantická notace jsou komplexní čísla. Implementace operací s komplexními hodnotami je v dnešních jazycích řešena po kódové stránce pomocí knihoven a po datové pomocí struktury nebo vektoru obsahující reálnou a imaginární část. Toto řešení ale není příliš sémantické, takže podpora komplexních čísel byla zapracována přímo do návrhu sémantické notace. Komplexní čísla se definují jako:

```
<type:complex />
```

S daty tohoto typu lze následně pracovat pomocí aritmetických operátorů a relačních operátorů, s výjimkou operátorů větší resp. menší, protože seřazení komplexních čísel není přímo možné (neexistuje relace částečného uspořádání). Pro komplexní čísla dále existují speciální sémantické operátory, které slouží ke konstrukci komplexního čísla, nebo vrací jeho reálnou nebo imaginární část, vzdálenost od počátku souřadné soustavy, resp. vzdálenost dvou komplexních čísel.

Konstanty používají matematický zápis a existuje i speciální konstanta – imaginární jednička, označována matematicky jako i nebo v elektrotechnice jako j:

```
<const:complex>2+3i</const:complex>
<const:imaginaryOne />
```

3.2.5 Znak

Jinou sadou elementárních typů jsou typy pracující s textovou informací. Základní typ této skupiny je takový typ, který je schopný definice jednoho znaku:

```
<type:character />
```

Bližší specifikace je znova možná a souvisí s velikostí znaku:

```
<type:character type="ascii"   />           <!--  7 bitů -->

<type:character type="byte"    />           <!--  8 bitů -->

<type:character type="unicode" />           <!-- 31 bitů -->
```

Jiným druhem specifikace je sdělení kódové stránky (znakové sady), která nám zaručí mapování vnitřního číselného kódu na konkrétní znaky:

```
<type:character type="byte" charset="iso-8859-2" />
```

U některých typů lze definovat přenosové kódování (typ serializace) jako:

```
<type:character type="unicode" encoding="UTF-8" />
```

Konstantní znaky zapíšeme již obvyklým způsobem do značek se stejným jménem jako je typ, jen v jmenném prostoru konstant:

```
<const:character>A</const:character>
```

Samozřejmě existují i speciální konstanty jako mezera a nový řádek (zde je zřejmá sémantická hodnota – prakticky to totiž bude znak CR, LF, CRLF, nebo EOL v závislosti na operačním systému nebo prostředí):

```
<const:space   />            <!-- mezera            -->

<const:newline />            <!-- nový řádek      -->
```

Ze standardu ASCII jsou řídící znaky uvedeny jako speciální znakové konstanty, příkladem jsou uvedeny některé známější:

```
<const:asciiNUL  />          <!-- 00 null            -->
<const:asciiBEL  />          <!-- 07 bell            -->
<const:asciiBS   />          <!-- 08 backspace       -->
<const:asciiTAB  />          <!-- 09 horizontal tab  -->
<const:asciiLF   />          <!-- 0A line feed       -->
<const:asciiVT   />          <!-- 0B vertical tab    -->
<const:asciiCR   />          <!-- 0D carriage return -->
<const:asciiESC  />          <!-- 1B escape          -->
```

3.2.6 Řetězec

Logicky následujícím textovým typem je typ vhodný pro ukládání řetězců. Opět existuje obecná forma:

```
<type:string />
```

Pro specifikaci lze použít parametry uvedené u znakového typu (*type, charset, encoding*). Výhodou specifikace znakové stránky je to, že v případě přiřazení hodnot z jedné proměnné do druhé se provede automatická konverze mezi kódovými stránkami a výsledek tedy bude konzistentní. Tímto přístupem odpadne část problémů, se kterými se potýkají některé dnešní programy.

Navíc u řetězců existuje parametr pro určení délky, např. 8 platných znaků se zapíše jako:

```
<type:string size="8" />
```

Implementační detaily lze jen naznačit – např. vytvoření dynamicky alokovaného řetězce ukončeného nulovým znakem dosáhneme za pomoci:

```
<type:string hint="allocation-type:dynamic;zero-terminated" />
```

Když potřebujeme dynamický řetězec, který bude obsahovat nulové znaky, tak to naznačíme příznakem *blob*, který zabezpečí, že se bude ukládat délka řetězce do skryté proměnné. Toto také zabezpečí přímý přístup k délce řetězce, která se nebude muset algoritmicky počítat pomocí hledání nulového ukončovacího znaku. Zápis tohoto příznaku je:

```
<type:string hint="allocation-type:dynamic;blob" />
```

Jiný způsob je plně statická alokace, bez dynamických změn velikosti:

```
<type:string hint="allocation-type:static" />
```

Statická alokace může být dále upřesněna na *zero-terminated* případně *blob*.

Řetězcové konstanty se tvoří standardním způsobem a způsob reprezentace v programu je určen místem použití tak, aby byla implementace efektivní. Zápis konstanty je tedy pouze:

```
<const:string>Hello world!</const:string>
```

U psaní (nejen) řetězcových konstant je potřeba pamatovat na to, že speciální znaky je potřeba uvodit, ale způsob je jednoduchý a standardizovaný – uvozuje se dle pravidel XML:

```
<const:string>Následující text je &quot;v uvozovkách&quot;</const:string>
<const:string>ampersand, menší, větší = &amp;, &lt;, &gt;</const:string>
```

Podrobnosti o uvozování poskytne standard XML, který je volně dostupný (viz. [5]).

3.2.7 Výčet

Posledním standardním elementárním typem je výčet, který má syntaxi o něco složitější. Skládá se ze dvou značek. Jedna uvozuje samotný výčet a druhá označuje jeho prvky:

```
<type:enumeration>
    <type:element>PONDĚLÍ</type:element>
    <type:element>ÚTERÝ</type:element>
    <type:element>STŘEDA</type:element>
    <type:element>ČTVRTEK</type:element>
    <type:element>PÁTEK</type:element>
    <type:element>SOBOTA</type:element>
    <type:element>NEDĚLE</type:element>
</type:enumeration>
```

V sémantickém programování je u výčtu možno definovat i číselné hodnoty jednotlivých prvků:

```
<type:enumeration>
    <type:element value="0">NE</type:element>
    <type:element value="1">ANO</type:element>
    <type:element value="2">NENÍ ZNÁMO</type:element>
</type:enumeration>
```

Také lze definovat tyto hodnoty pomocí automatické řady, jak aritmetické – specifikované pomocí první hodnoty a inkrementu (různého od 0):

```
<type:enumeration start="1" increment="1">
    <type:element>PONDĚLÍ</type:element>
                 :
                 :
    <type:element>NEDĚLE</type:element>
</type:enumeration>
```

tak i geometrické pomocí první hodnoty a násobku (různého od 1), které jsou vhodné na bitové masky, konkrétní příklad z protokolu TCP/IP:

```
<type:enumeration start="1" multiply="2">
    <type:element>RESERVED</type:element>
    <type:element>DONT_FRAGMENT</type:element>
    <type:element>MORE_FRAGMENTS</type:element>
</type:enumeration>
```

Zde mají jednotlivé prvky tyto výsledné hodnoty:

	Decimálně	***Binárně***
RESERVED	1	0001
DONT_FRAGMENT	2	0010
MORE_FRAGMENTS	4	0100

3.2.8 Speciální typy

V sémantickém programování existují tři speciální typy, které jsou použity převážně jako zástupné typy. První typ – prázdný, slouží pro definování typu, který nenabývá žádné hodnoty. Využití je např. u návratové hodnoty funkce, která nemusí vždy existovat, což je případ procedur (funkce a procedura v pojetí programovacího jazyka Pascal). Značení typu je:

```
<type:void />
```

Dalším typem, téměř opakem prázdného typu, je typ libovolný. Svoje uplatnění nachází v importovaném kódu z beztypových zdrojů, kde každá proměnná může nabývat hodnoty libovolného typu. Také může sloužit jako náhrada typu *Variant* známého z technologie COM/DCOM. Zápis typu je opět minimalistický:

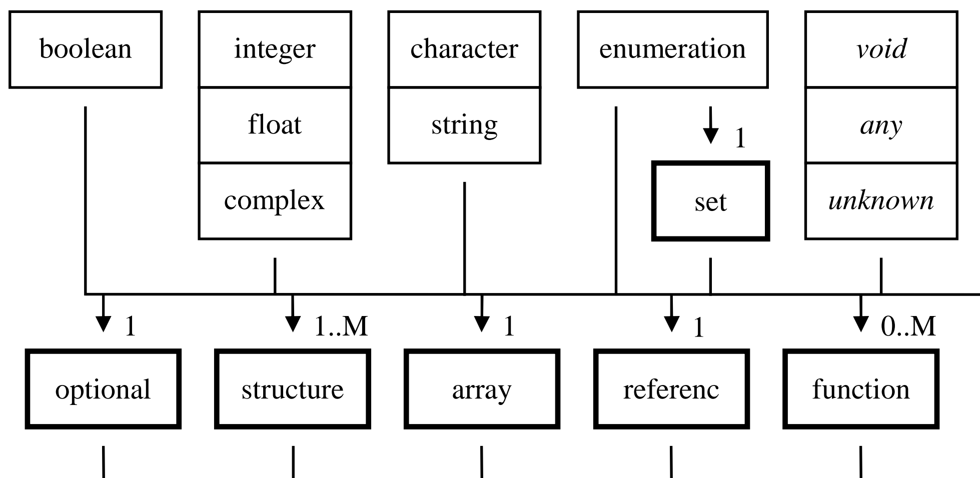
```
<type:any />
```

Posledním ze speciálních typů je neznámý typ. Slouží jako zástupný typ po dobu než programátor doplní do zdrojového kódu prozatím neurčený typ. Interně se používá jako výsledek operace, kterou nelze provést nad požadovanými operandy. Forma zápisu tohoto typu:

```
<type:unknown />
```

3.3 Typové operátory

Pro konstrukci složitějších typů lze aplikovat typové operátory a tím vytvořit libovolně strukturovaný typ. Přehled elementárních datových typů (tenké ohranučení) a typových operátorů (tlusté ohraničení) poskytuje obr. 8. Vyznačen je také počet zdrojových elementů potřebných k aplikaci typového operátoru.



Obr. 8: *Přehled všech datových typů*

3.3.1 Volitelná hodnota

Nejjednodušším typovým operátorem je volitelná hodnota. Jeho operandem je libovolný datový typ a výsledkem je typ, který kromě prostoru hodnot původního typu může obsahovat také neurčenou hodnotu, označenou jako *null*:

```
<const:null />
```

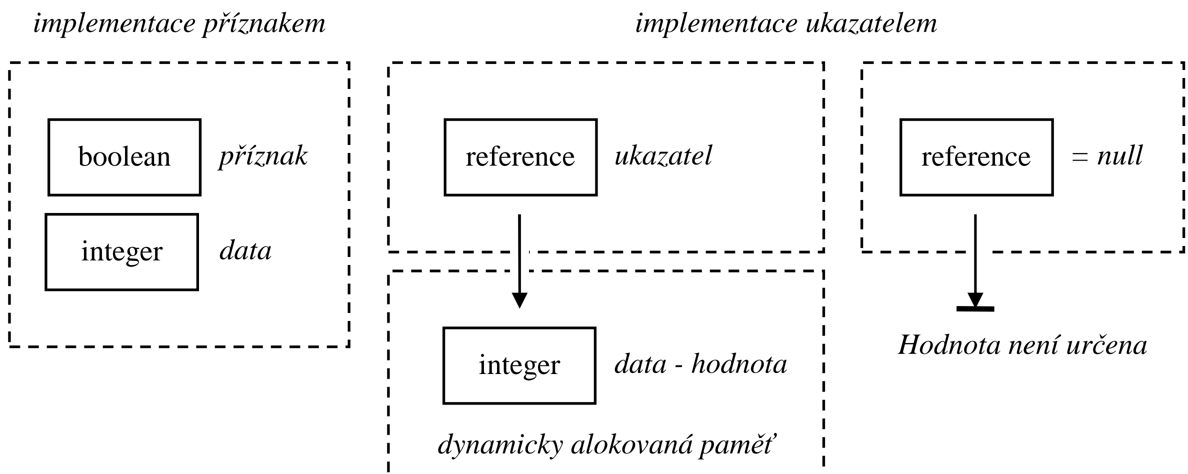
Aplikace operátoru, např. na typ *boolean*, se provede zápisem:

```
<type:optional>
    <type:boolean />
</type:optional>
```

Tato konstrukce může nabývat jak hodnot původního typu, tj. *true* a *false*, tak i speciální hodnoty *null,* které značí nedefinovaný stav.

Implementace operátoru je možná dvojím způsobem (obr. 9). Buď se vytvoří struktura o dvou prvcích, kde první bude vždy příznak neurčené hodnoty a druhý bude typu dle operandu. Tento způsob se nazývá implementace příznakem. Druhý způsob se nazývá implementace pomocí ukazatele a využívá dynamickou alokaci paměti. Zde buďto ukazatel ukazuje na platnou (alokovanou) pozici, kde jsou data dle typu operandu, nebo naopak neukazuje nikam a tento stav odpovídá tomu, že hodnota datového typu není určena. Naznačení implementace se provede pomocí atributu *hint*:

```
<type:optional hint="implement:by-flag">
    <type:integer />
</type:optional>

<type:optional hint="implement:by-pointer">
    <type:integer />
</type:optional>
```



Obr. 9: *Způsoby implementace typového operátoru volitelná hodnota*

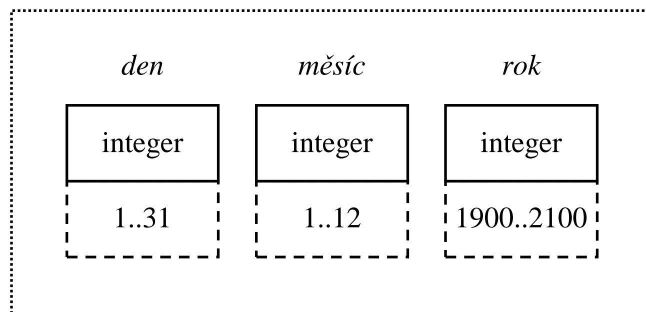
3.3.2 Struktura

Následujícím typovým operátorem, tentokrát akceptujícím více operandů, je operátor pro tvorbu struktury. I když je možnost ho aplikovat na jeden a více operandů, skutečný smysl má pouze u dvou nebo více operandů.

Aplikace vlastního operátoru, skládajícího se ze dvou značek se provede následovně:

```
<type:structure>

 <type:element name="den">
  <type:integer min="1" max="31" />
 </type:element>

 <type:element name="měsíc">
  <type:integer min="1" max="12" />
 </type:element>

 <type:element name="rok">
  <type:integer min="1900" max="2100" />
 </type:element>

</type:structure>
```

Uvedená konstrukce definuje strukturu vhodnou na ukládání data – obsahuje složky pro den, měsíc a rok (znázorněna také na obr. 10; to, že typ není pojmenovaný souvisí s tím, že jeho definice se nenachází v konstrukci *typedef*). Položky struktury mohou být pojmenovány libovolně, ale unikátně, tj. nemohou existovat dvě položky se stejným jménem (parametr *name*).



Obr. 10: *Příklad struktury*

Struktura má využití především při složitějších programech, kde chceme zachovat logické vazby mezi daty (především seskupení). Částečně může také sloužit jako mezikrok k objektově orientovanému návrhu, kde struktura plní úlohu zapouzdření dat. Pokud jsou prvky struktury navíc odkazy na funkce, můžeme mít jednoduchou náhradu polymorfních metod. U všech funkcí v úloze metod se požaduje, aby prvním parametrem takovéto funkce byl odkaz na strukturu samotnou – v opačném případě se nedá zaručit příslušnost metody k objektu.

3.3.3 Pole

Jelikož struktura pole je homogenní (prvky jsou stejného typu), tak operátor pro vytvoření pole potřebuje právě jeden operand, kterým se určí typ prvku. Dále je nutno specifikovat rozměr pole, nebo naznačit, že jde o pole neomezené.

Příklad zápisu pole o 4 prvcích s hodnotou 0 až 255 (např. dříve zmíněna IPv4 adresa):

```
<type:array min="0" max="3">
 <type:integer min="0" max="255" />
</type:array>
```

Vlastnost variability se projevuje v sémantickém programování také u tohoto operátoru, protože stejného efektu u definice rozsahů indexů lze dosáhnout i alternativní hlavičkou – následující zápisy jsou si plně ekvivalentní:

```
<type:array min="0" max="3"         >

<type:array         max="3"         >

<type:array min="0"         size="4">

<type:array                 size="4">

<type:array min="0" max="3" size="4">

<type:array         max="3" size="4">
```

Pokud z uvedených parametrů nelze určit velikost pole, předpokládá se indexace bez omezení hraniční hodnoty. Aby se ale předešlo chybám (a také varovným hlášením překladače), je vhodné tento fakt explicitně vyzdvihnout naznačením pomocí parametru *hint*:

```
<type:array min="0" hint="index:unlimited">
```

3.3.4 Množina

Některé typové operátory lze aplikovat jenom na specifické datové typy. Jedním z těchto operátorů je operátor pro tvorbu množin. V sémantickém programování slouží k tvorbě konečných množin a jeho aplikace je možná pouze na datový typ výčet a to následovně:

```
<type:set>
 <type:enumeration>
  <type:element>RED</type:element>
  <type:element>GREEN</type:element>
  <type:element>BLUE</type:element>
 </type:enumeration>
</type:set>
```

Tato konstrukce umožňuje ukládat kombinace barev – má 8 různých stavů. Obecně počet kombinací pro množinu je 2^N, kde N je počet prvků výčtového typu, ze kterého se množina vytváří.

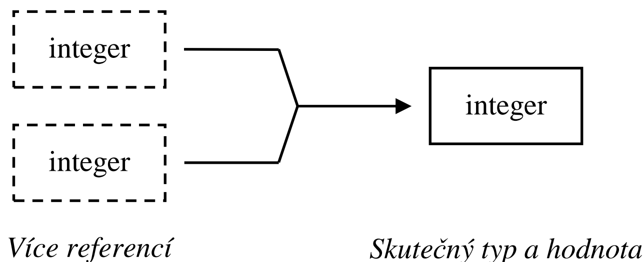
3.3.5 Reference

Sémantické programování nezná pojem ukazatel jako takový, ale používá se pojem reference. Referenci lze z typu vytvořit pomocí operátoru. Protože sémantické programování patří mezi silně typované jazyky, přiřazovat lze pouze hodnoty referencí, které se odkazují na hodnoty stejných typů. Neomezené přetypování známé z jazyku C tedy možné není.

Příklad pro vytvoření reference:

```
<type:reference>
    <type:integer />
</type:reference>
```

Tato reference „ukazuje" na místo, kde je celé číslo typu *integer* (obr. 11). S proměnnou, která obsahuje referenci, lze provádět omezený počet operací, např. přiřadit (referenci) nebo dereferencovat (získat cílovou hodnotu). Hodnotu takové proměnné typu reference pak dostaneme pomocí operace vytvoření reference.



Více referencí *Skutečný typ a hodnota*

Obr. 11: *Ilustrace reference*

3.3.6 Funkce

V sémantickém programování je funkce také typem a tvoří se pomocí typového operátoru. Operátor rozlišuje dvě sekce – parametry a návratovou hodnotu. Parametry jsou volitelné a když existují, jsou pojmenovány jednoznačnými názvy. Jejich počet je pevně určen při definici typu funkce. Druhá sekce definuje typ výsledné hodnoty funkce. V případě, že zde není definován žádný typ, nebo tato sekce chybí, tak funkce žádnou hodnotu nevrací.

Příklad typu funkce, která očekává celočíselný parametr a vrací celé číslo (např. faktoriál):

```
<function>

    <parameters>
        <parameter id="vstup"><integer /></parameter>
    </parameters>

    <returns>
        <integer />
    </returns>

</function>
```

3.4 Příkazy a operátory

3.4.1 Organizační pravidla

Pro udržení rozumné organizace zdrojových kódů, jak z pohledu programátora, tak z pohledu překladače, bylo navrženo několik pravidel a postupů.

Jedním z těchto pravidel je možnost definovat si vlastní datový typ pomocí značky *typedef*. Tato značka se nachází v kořenovém uzlu a má za úlohu registrovat příslušný fragment kódu pod zadaným identifikátorem. Po této registraci je možno používat typ na libovolném místě ve zdrojových kódech pomocí vhodných odkazů (relativní, absolutní cesta k identifikátoru).

Příklad definice typu:

```
<?xml version="1.0" encoding="utf-8"?>
<typedef id="i2i" xmlns="http://rozsnyo.com/mdfs/type">

    <function>

        <parameters>
            <parameter id="input">
                <integer />
            </parameter>
        </parameters>

        <returns>
            <integer />
        </retutrn>

    </function>

</typedef>
```

Tato definice registruje typ funkce pod identifikátorem *i2i*. V případě, že je XML soubor uložen v adresáři, se před identifikátor z parametru *id* doplní cesta (vzhledem k nastavenému kořenovému adresáři), tím můžeme dostat identifikátor např. *priklady/faktorial/i2i*.

Dalším organizačním pravidlem je zápis vlastních programů. Zápis je podobný předchozímu příkladu, protože programy jsou také fragmenty s vlastním identifikátorem. Navíc programy jsou typovány, tudíž v kořenovém uzlu dokumentu je o parametr více – *type*. Podle typu se pozná, pro jaké prostředí je program určen. Standardně podporovaným typem programů je */application/cmdline*, což je program spuštěný z příkazové řádky – je volán se sadou parametrů a může vracet návratový kód. Příkladem na nejkratší program je:

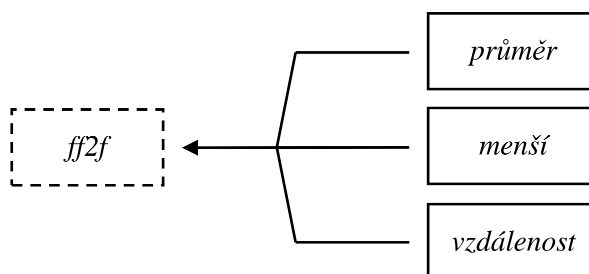
```
<?xml version="1.0" encoding="UTF-8"?>
<program id="empty" type="/application/cmdline"
  xmlns="http://rozsnyo.com/mdfs/code">

    <body />

</program>
```

Ve skutečných programech značka *body* nebude prázdná a před ní se budou nacházet definice uživatelských funkcí.

3.4.2 Definice funkce

Základním stavebním prvkem programů v procedurálních jazycích jsou funkce. V předchozím textu o datových typech bylo uvedeno, že funkce mají vlastní typ a bylo také ukázáno, jak se tento typ definuje. Kromě typu je samozřejmou součástí funkce její tělo. V sémantickém programování je tedy jasně definována hranice mezi rozhraním (typ funkce) a implementací (tělo funkce).

Příklad využití ukazuje obr. 12. Typ funkce, *ff2f*, obsahuje dva operandy a návratovou hodnotu s pohyblivou řádovou čárkou. Tento typ může využívat hned několik různých matematických funkcí:



Společné rozhraní – typ funkce *Více různých implementací*

Obr. 12: *Rozhraní a implementace funkce*

Konkrétní definice jedné z těchto funkcí je následující (několik značek je pro přehlednost uvedeno ve zkrácené formě):

```
<function id="průměr" type="ff2f">

    <return>
        <div>
            <add>
                <arg>a</arg>
                <arg>b</arg>
            </add>
            <const:int>2</const:int>
        </div>
    </return>

</function>
```

Jak je zde vidět, stačí specifikovat typ pomocí odkazu na identifikátor (použití jen *ff2f* značí, že definice je ve stejném jmenném prostoru, resp. definiční soubor typu je ve stejném adresáři). Po této specifikaci pak lze ve funkci využívat parametry funkce – pomocí značky *arg*, (v plném znění jako *argumentValue*). Následně se očekává, že funkce skutečně vrátí návratovou hodnotu pomocí příkazu *ret* (resp. *return*).

3.4.3 Návrat z funkce

Většina funkcí má návratovou hodnotu, jejíž typ je přesněji specifikován v definici typu funkce. Hodnotu, kterou pak funkce skutečně vrací určíme příkazem *return* (zkrácený tvar je *ret*). Tento příkaz zároveň slouží pro okamžitý návrat z funkce.

Konstrukce známá z programovacích jazyků *Pascal* a *Object Pascal* (*Delphi*), kde se návratová hodnota přiřazuje do virtuální proměnné *result,* se v této prvotní definici sémantického programování nenachází.

Příklad použití příkazu *return* k vrácení hodnoty 2006:

```
<return>
    <const:integer>2006</const:integer>
</return>
```

3.4.4 Příkaz větvení

Základním příkazem větvení je konstrukce *if-then* resp. *if-then-else*. Tato konstrukce je přímo podporována v sémantickém programování v následujícím tvaru:

```
<if>
    <condition><!-- podmínka --></condition>
    <then><!-- kód provedený při splněné podmínce   --></then>
    <else><!-- kód provedený při nesplněné podmínce --></else>
</if>
```

Ve značkách *condition* (zkráceně *cond*) se nachází podmínka, která je vlastně výrazem s návratovou hodnotou typu *boolean*. V sekcích *then* a *else* se pak nachází příslušné sekvence příkazů. Tato konstrukce má dále v sémantickém programování i neúplné podoby, např. je možno vynechat jednu z větví – jak část *else* (jako v dnešních jazycích), tak i po novém část s *then* – přičemž konstrukce *if-else* se v takovémto případě interně transformuje na *if-then* s podmínkou v negaci.

Příkladem pro skutečnou podobu příkazu je tělo funkce *menší*:

```
<if>
    <cond>
        <lt>
            <arg>a</arg>
            <arg>b</arg>
        </lt>
    </cond>
    <then>
        <ret>
            <arg>a</arg>
        </ret>
    </then>
    <else>
        <ret>
            <arg>b</arg>
        </ret>
    </else>
</if>
```

3.4.5 Příkazy cyklu

V sémantickém programování není nutno používat jen stávající konstrukce. Např. nejznámější konstrukce cyklu *for* ani nemá přímý ekvivalent, protože ji lze nahradit ekvivalentním cyklem *while-do*, která má v sémantickém programování tvar:

```
<while>
    <condition>
        <!-- podmínka -->
    </condition>
    <do>
        <!-- kód provedený při splněné podmínce -->
    </do>
</while>
<!-- pokračování zde, při nesplněné podmínce -->
```

Přímá náhrada uvedeného cyklu *for* z programovacího jazyka C ve tvaru:

```
for( hlavička; podmínka; inkrement ) {
    tělo;
}
```

se po konverzi do sémantické notace změní na:

```
<!-- hlavička -->
<while>
    <condition>
        <!-- podmínka -->
    </condition>
    <do>
        <!-- tělo -->
        <!-- inkrement -->
    </do>
</while>
```

V sémantickém programování pak existuje konstrukce *do-while*, ve které se tělo provede alespoň jednou a podmínka se kontroluje až po provedení těla. Tato konstrukce má zápis:

```
<do>
    <body>
        <!-- kód těla -->
    </body>
    <while>
        <!-- podmínka -->
    </while>
</do>
```

Dalším příkazem cyklu, tentokrát typickým pro sémantické programování, je prosté opakování – s předem určeným počtem iterací. Tvar ve zdrojovém kódu pro 5 opakování je:

```
<repeat count="5">
    <!-- tělo cyklu -->
</repeat>
```

Poslední typ cyklu je identifikovatelný jako cyklus od-do, označený značkou *loop*. Zde se definují meze intervalu, přes který se iteruje. Pokud je výchozí hodnota větší než konečná, iteruje se záporným krokem. V případě, že krok není definován (protože lze vynechat část *step*), uvažuje se jednotková hodnota kroku.

Úplný zápis konstrukce má tvar:

```
<loop id="identifikace">
    <from>
        <!-- výchozí hodnota -->
    </from>
    <to>
        <!-- konečná hodnota -->
    </to>
    <step>
        <!-- skok při iteraci -->
    </step>
    <body>
        <!-- tělo cyklu -->
    </body>
</loop>
```

Specialitou této konstrukce je hodnota iterace, k níž se lze dostat pomocí znalosti identifikace cyklu uvedené v parametru *id*. Vlastní hodnota se následně získá elementárním výrazem, který je značen pomocí značky *iterationValue* nebo *current* (existují dvě různé, plně ekvivalentní značky):

```
<iterationValue>identifikace</iterationValue>

<current>identifikace</current>
```

3.4.6 Příkaz výstupu

V případě, že program je takového typu, že existuje možnost výstupu informací, tak se tento výstup provede příkazem *write* (zkráceně *out*). Protože sémantická notace je silně typovaná, příkaz výstupu zná typ výrazů, které chce programátor vypsat a proto lze libovolně tyto výrazy kombinovat – výpis bude proveden individuálně pro každý výraz.

Následuje příklad na příkaz výstupu v kombinaci s výše uvedeným cyklem *loop*, přístupem k hodnotě iterace a sémantickou konstantou nového řádku:

```
<loop id="number">
    <from>
        <const:int>1</const:int>
    </from>
    <to>
        <const:int>10</const:int>
    </to>
    <body>
        <write>
            <current>number</current>
            <const:newline />
        </write>
    </body>
</loop>
```

3.4.7 Logické operátory

Logické operátory provádějí operaci konjunkce (logické a, *and*), disjunkce (logické nebo, *or*) a negace (logické ne, *not*). Operandy pro tyto operace i výsledek operace jsou typu pravdivostní hodnota (*boolean*).

Operátor pro logickou negaci je unární, což znamená, že akceptuje jediný operand. Jako příklad lze uvést negaci konstanty *false*:

```
<not>
    <const:false />
</not>
```

Hodnota tohoto výrazu je rovna konstantě *true*:

```
<const:true />
```

Operátory konjunkce a disjunkce nejsou omezeny na binární formu, ale mohou být provedeny na libovolném (nenulovém) počtu operandů. Tyto operandy jsou v sémantickém programování na stejné úrovni. Neaplikuje se žádné závorkování do dvojic ani zleva ani zprava.

Příklad se čtyřmi operandy v sémantickém programování:

```
<and>
    <!-- operand A -->
    <!-- operand B -->
    <!-- operand C -->
    <!-- operand D -->
</and>
```

Tento kód by při povinné binární formě s levým závorkováním musel být zapsán zbytečně nepřehledným zápisem:

```
<and>
    <and>
        <and>
            <!-- operand A -->
            <!-- operand B -->
        </and>
        <!-- operand C -->
    </and>
    <!-- operand D -->
</and>
```

U logických operátorů lze nastavit typ vyhodnocování. Výchozí metodou je zkrácené vyhodnocování, které zpracovává operandy zleva (od prvního) a v případě, že je výsledek nezměnitelný pomocí dalšího operandu, tak se rovnou vrátí výsledná hodnota. Opačný přístup je metoda úplného vyhodnocování všech operandů. Volbu tohoto typu vyhodnocování nastavíme parametrem *evaluate*:

```
<and evaluate="all">
    <!-- operand A -->
    <!-- operand B -->
</and>
```

3.4.8 Relační operátory

Relační operátory slouží pro porovnávání dvou nebo více operandů. Existuje celkově šest operátorů tohoto druhu. Operátory jsou uvedeny včetně názvu značky ve zkrácené formě:

- rovný *eq*
- nerovný *neq*
- menší *lt*
- menší nebo rovný *lte*
- větší *gt*
- větší nebo rovný *gte*

Aplikace těchto operátorů je možná na operandy stejného nebo kompatibilního typu a výsledek výrazu s relačním operátorem je vždy typu *boolean*. Na některé datové typy lze aplikovat jen podmnožinu relačních operátorů, např. na komplexní čísla jde použít jen operátory rovný a nerovný.

V případě, že se operátor aplikuje na více než dva operandy, musí platit relace definovaná operátorem pro všechny dvojice operandů nacházející se vedle sebe. Pomocí této vlastnosti lze konstruovat výrazy např. pro ověření vlastností posloupností o známém počtu prvků:

```
<!-- je posloupnost stoupající ? -->
<lt>
    <!-- operand 1 -->
    <!-- operand 2 -->
    <!-- operand 3 -->
    <!-- operand 4 -->
</lt>

<!-- je posloupnost neklesající ? -->
<lte>
    <!-- operand 1 -->
    <!-- operand 2 -->
    <!-- operand 3 -->
    <!-- operand 4 -->
</lte>
```

Využitím uvedené vlastnosti lze konstruovat efektivnější výrazy pro porovnání, protože každý operand bude vyhodnocen maximálně jednou. V dnešních jazycích by pro tyto vlastnosti porovnání bylo zapotřebí manuálně vytvářet pomocné proměnné, které by uchovávaly hodnoty právě porovnávaných operandů. Taktéž odpadá potřeba použití logických spojek a např. pro porovnání čtyř hodnot stačí aplikovat vhodný operátor v následující formě:

```
<eq>
    <!-- operand 1 -->
    <!-- operand 2 -->
    <!-- operand 3 -->
    <!-- operand 4 -->
</eq>
```

Typ vyhodnocování výrazu s relačním operátorem a třemi nebo více operandy je zkrácený, ale pro situace, kdy je potřeba vyhodnotit všechny operandy, lze opět nastavit režim úplného vyhodnocování pomocí parametru *evaluate* s hodnotou *all*.

3.4.9 Aritmetické operátory

Pro provádění aritmetických operací jsou k dispozici následující aritmetické operátory:

- sčítání a odečítání *add, sub*
- násobení a dělení *mul, div*
- zbytek po dělení *mod*
- negace *neg*
- znaménko *sgn*

Některé z těchto operátorů umožňují aplikaci i na více než dva operandy – např. u sčítání nebo násobení je výsledek této operace jednoznačný. U odečítání se pak od prvního operandu odečítají ostatní a u dělení je postup analogický (první operand se dělí ostatními). Zbytek po dělení je binárním operátorem. Na rozdíl od současných jazyků má navíc definováno chování i pro operandy s pohyblivou řádovou čárkou. Poslední dva operátory jsou již unární – negace vrátí hodnotu operandu s opačným znaménkem a operátor „znaménko“ vrací hodnotu -1, 0 nebo 1 dle znaménka operandu. Příklad: matematický vzorec pro výpočet průměru dvou hodnot je:

$$\frac{a+b}{2} \tag{1}$$

V sémantické notaci bude tento vzorec zapsán jako:

```
<div>
    <add>
        <arg>a</arg>
        <arg>b</arg>
    </add>
    <const:int>2</const:int>
</div>
```

Množina aritmetických operátorů v uvedeném rozsahu není úplná, což znamená, že neobsahuje všechny elementární matematické operace, které jsou v současných jazycích v podobě funkcí součástí knihoven (práce s mocninami, logaritmy, atd.). Použití knihoven funkcí pro tento účel ale nepřipadá do úvahy, protože namísto informace o matematické operaci by se zapisovala implementace – volání funkce. Řešením je tedy definovat pro každou matematickou operaci vlastní značku, která může mít i vlastní jmenný prostor (např. *math*). Pak bude pro vzorec jako:

$$\sin\frac{\pi}{4} \tag{2}$$

existovat ekvivalent v sémantické notaci ve tvaru:

```
<math:sin>
    <div>
        <const:Pi />
        <const:int>4</const:int>
    </div>
</math:sin>
```

3.4.10 Volání funkce

Operace volání funkce se provádí operátorem volání funkce. Tento operátor vyžaduje identifikaci funkce, která se specifikuje ve většině případů lokálním nebo relativním odkazem. Dále se při volání funkce předávají parametry funkce, které jsou tvořeny operandy právě popisovaného operátoru volání funkce. Tyto operandy se vyčíslují v pořadí v jakém jsou uvedeny u operátoru volání funkce a po vyčíslení se pak provede samotná operace volání.

Příklad na volání funkce *faktoriál* s parametrem 10:

```
<call function="factorial">

    <parameter>
        <const:int>10</const:int>
    </parameter>

</call>
```

Rozlišení parametrů funkce se provádí pomocí parametru *id*. Parametry se ale nemusí rozlišovat, pokud je z typu funkce jednoznačné, který parametr je který. V případě, že existuje více parametrů stejného typu a nebudou rozlišeny při volání názvem, tak bude překladač zobrazovat varování o možné nejednoznačnosti.

Příklad s rozlišováním parametrů, např. u šifrování, kde data a klíč jsou stejného typu:

```
<call function="zasifruj">

    <param id="data">
        <const:string>Text který chceme zašifrovat</const:string>
    </param>

    <param id="klic">
        <const:string>TAJNÉ HESLO</const:string>
    </param>

    <param id="metoda">
        <const:enum type="sifrovaci metoda">DES</const:enum>
    </param>

</call>
```

Možnosti operátoru volání funkce jsou v sémantickém programování rozšířené o zápis rekurzivního volání. Tento tvar se ale může nacházet jen ve funkci, která se tímto operátorem volá. Použití přímé rekurze je pro programátora výhodnější, protože nezávisí na identifikátoru funkce, který se může v průběhu vývoje programu měnit.

Příklad fragmentu kódu pro rekurzivní volání, který se nachází ve funkci na výpočet hodnoty faktoriálu:

```
<recursion>
    <param id="input">
        <sub>
            <arg>input</arg>
            <const:one />
        </sub>
    </param>
</recursion>
```

3.4.11 Indexace polí

K indexaci polí slouží operátor, který požaduje alespoň dva operandy: samotné pole a požadovaný index. V případě, že je potřeba indexovat vícerozměrné pole, je v sémantickém programování uvedeno více indexů.

Příklad pro indexaci jednorozměrného pole – získání nultého prvku z pole *argv*:

```
<index>
    <arg>argv</arg>
    <const:zero />
</index>
```

Příklad pro indexaci vícerozměrného pole, např. uvnitř dvojitého cyklu (značky *current*):

```
<index>
    <arg>matice</arg>
    <current>radek</current>
    <current>sloupec</current>
</index>
```

3.4.12 Přístup k prvkům struktury

Hodnota prvku struktury se získá pomocí operátoru pojmenovaného *access*, který vyžaduje jeden operand – vlastní strukturu. Název elementu, který se má získat, je uložen ve formě parametru *element*. Příklad pro získání roku ze struktury pro datum:

```
<access element="rok">
    <arg>datum</arg>
</access>
```

V sémantické notaci lze použít i jednodušší formu, protože parametr *element* lze uvést přímo u některých značek. Výše uvedený příklad je tedy zapsatelný jako:

```
<arg element="rok">datum</arg>
```

K prvkům komplexních struktur lze přistupovat i pomocí jednoduššího zápisu, známého z jazyka Pascal. V sémantické notaci má tato konstrukce následující tvar:

```
<with id="identifikace">
    <structure>
        <!-- výraz typu struktura -->
    </structure>
    <do>
        <!-- příkazy -->
    </do>
</with>
```

Identifikace je nepovinná a slouží jen pro rozlišení vnořených konstrukcí. V sekci *do* se pak nachází kód, ve kterém značky *element* slouží pro přístup k jednotlivým elementům:

```
<element from="identifikace">název elementu</element>
```

3.4.13 Práce s komplexními čísly

Komplexní čísla mají podporu v základní definici sémantického programování pomocí několika operátorů. Tyto operátory slouží pro vytvoření komplexního čísla, získání jeho reálné nebo imaginární složky nebo převedení do polárních souřadnic.

Příklad vytvoření komplexního čísla *1,9+8,2i* za pomoci operátoru *compose*:

```
<compose>
    <real>
        <const:float>1.9</const:float>
    </real>
    <imaginary>
        <const:float>8.2</const:float>
    </imaginary>
</compose>
```

V reálné nebo imaginární sekci mohou být uvedeny i výrazy používající proměnné a matematické operace. Zpětný převod, tedy získání složek komplexního čísla, se provádí operátory *real* a *imaginary*. Příkladem je uvedeno získání těchto částí z parametru funkce:

```
<real>
    <arg>komplexní</arg>
</real>

<imaginary>
    <arg>komplexní</arg>
</imaginary>
```

Z komplexního čísla lze kromě reálné a komplexní složky získat také polární souřadnice – pomocí operátorů *angle* a *radius*. Příklady aplikace těchto operátorů:

```
<angle>
    <arg>komplexní</arg>
</angle>

<radius>
    <arg>komplexní</arg>
</radius>
```

V sémantickém programování má operátor pro úhel ještě jednu zvláštnost, protože dovoluje určit chování v případě nedefinovaného stavu, který nastává při jeho aplikaci na nulovou hodnotu (zde úhel není možné určit). Existují dvě volby pro toto chování a implicitní je určena dle typu, do kterého se úhel přiřazuje. Je tedy možné vracet nulu (pro čistě číselný typ):

```
<angle onZero="zero">
    <arg>komplexní</arg>
</angle>
```

nebo vracet volitelnou hodnotu s tím, že při aplikaci na nulu hodnota nebude určena:

```
<angle onZero="optional">
    <arg>komplexní</arg>
</angle>
```

3.4.14 Definice a používání proměnných

V sémantickém programování lze samozřejmě používat i proměnné, které můžeme definovat pro libovolný blok příkazů – uvnitř těla programu, těla funkce, bloku cyklu a také v blocích *then* a *else*.

Definici lokální proměnné celočíselného typu ve funkci provedeme pomocí značky *local*:

```
<function id="název funkce" type="typ funkce">

    <local id="název proměnné">
        <type:integer />
    </local>

    <!-- tělo funkce -->

</function>
```

V případě, že chceme definovat proměnnou typu, který má vlastní globální identifikátor, použijeme k tomu parametr *type*:

```
<function id="název funkce" type="typ funkce">

    <local id="název proměnné" type="název typu" />

    <!-- tělo funkce -->

</function>
```

Hodnota proměnné se čte pomocí operátoru *variable* (zkráceně *var*), která požaduje identifikátor proměnné. Proměnné mohou mít v hierarchicky vnořených blocích stejná jména, ale v tom případě ztrácíme možnost přistupovat k hodnotám proměnných definovaných výše.

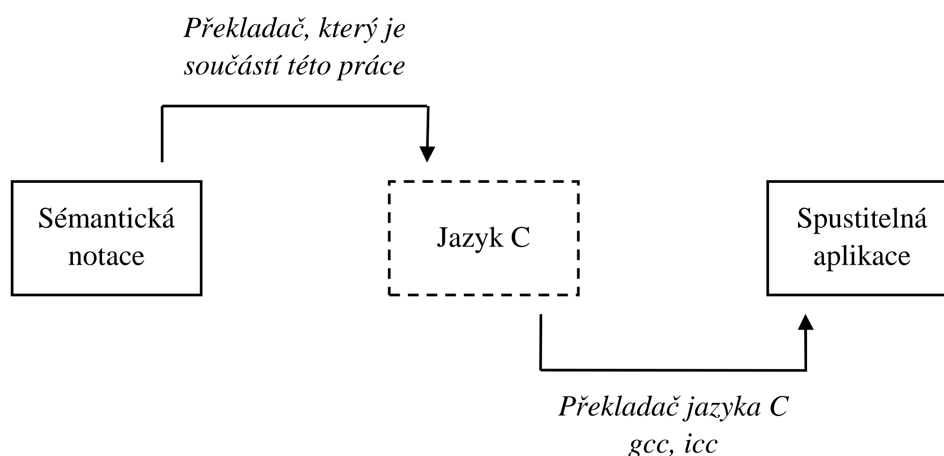
Zápis hodnot do proměnných se naopak provádí pomocí operátoru nastavení (značka *set*), který požaduje alespoň dva operandy – proměnnou a hodnotu. V případě, že se má nastavit více proměnných na stejnou hodnotu, je možno tyto proměnné uvést v rámci jednoho operátoru nastavení.

Příklad pro přičtení prvku pole k celkové sumě ilustrující použití operátorů pro čtení a zápis proměnných:

```
<set>
    <var>suma</var>
    <add>
        <var>suma</suma>
        <index>
            <arg>pole</arg>
            <current>i</current>
        </index>
    </add>
</set>
```

4 Realizace překladače

Pro potřeby ověření vlastností sémantického programování byl vytvořen překladač zdrojových kódů ze sémantické notace do programovacího jazyka C. Tento překladač je vlastně odpovědí na základní problém – jak získat spustitelný program ze sémantické notace. Prezentovaná metoda vytvoření spustitelné aplikace není přímá, protože využívá mezi-překladu do jazyka C (obr. 13).



Obr. 13: *Vytvoření spustitelné aplikace*

Překladač implementuje jen podmnožinu definice sémantické notace, protože v dnešní době ještě neexistují takové nástroje na správu dat, se kterými by byl překladač optimálně implementovatelný. Požadavky, které jsou kladeny na uvedený nástroj jsou:

- ukládání strukturovaných dat a efektivní přístup k nim
- ukládání nestrukturovaných dat
- na systémové úrovni se musí zabezpečit
    - překlad fragmentů
    - kompozice dat

První bod splňují nativní XML databáze, které v dnešní době již existují. Druhý bod lze splnit buď relační databází, nebo hierarchickým souborovým systémem. Třetí lze v současnosti vyřešit pomocí uživatelských skriptů a překladačů. Nástrojem, který by splňoval všechny požadavky optimálnosti je tedy integrace několika programů do jednoho, což ale nebylo možné provést z důvodu nedostatků zdrojů (čas, programátoři).

Z uvedených důvodů byl tedy překladač realizován jako samostatně fungující aplikace s omezenou množinou zpracovatelných typů a operátorů. Tento překladač je schopen generování pouze jednoduchých, ale plně funkčních programů ze sémantické notace.

4.1 Vnější prostředí

Překladač je implementován v programovacím jazyce PHP řady 5 a ke svému běhu proto potřebuje mít nainstalován i interpret tohoto skriptovacího jazyka. Pro čtení souborů ve formátu XML je použito rozšíření jazyka PHP nazvané *SimpleXML*, které je standardní součástí distribuce interpretu. Verze, na které byla otestována funkčnost překladače je *php-5.1.4*.

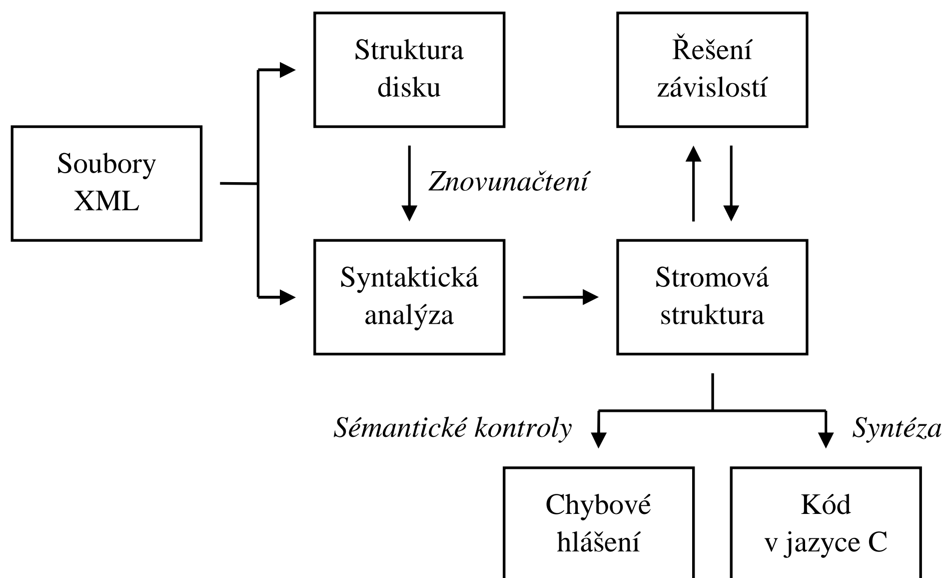
Pro další krok překladu, z jazyka C do spustitelné podoby, je použit překladač *gcc – GNU C compiler* (verze 3.4.6). Experimentálně byl ověřen i překlad pomocí dalšího překladače – *icc* (*Intel C compiler,* ve verzi 9.0), který poslouží stejně dobře, ale jeho licenční politika je komplikovanější (pro komerční použití je potřeba zaplatit licenční poplatek).

Překladač byl vytvořen na operačním systému Linux, ale je možné jej použít rovněž v obdobných systémech (BSD, unix), které disponují interpretem jazyka PHP a překladačem jazyka C. Na operačních systémech Microsoft Windows není a ani nebude možno provozovat překladač plnohodnotně a to z důvodu odlišné implementace víceuživatelské práce se soubory.

4.2 Vnitřní struktura

Vnitřní strukturu překladače znázorňuje obr. 14. Fragmenty zdrojového kódu v podobě souborů jsou uloženy na disku v určeném adresáři, jehož struktura je načtena do paměti. Tato struktura je pak průběžně synchronizována se skutečným obsahem disku a jsou zde detekovány změny ve zdrojových kódech. Když se soubor změní, je znova načten.

Načtení souborů probíhá pomocí syntaktických analyzátorů. Výstupem analýzy je strom, se kterým se pracuje dále na vyšší úrovni – probíhají sémantické kontroly, zpracování závislostí, optimalizace a syntéza kódu.



Obr. 14: *Vnitřní struktura překladače*

4.3 Obecné vlastnosti

4.3.1 Podporované prvky sémantického programování

Jak již bylo zmíněno, překladač neimplementuje vše, co bylo v předchozí kapitole definováno, ale jen podmnožinu této sémantické notace. Následující seznamy specifikují prvky sémantického jazyka, které v současné době dokáže nový překladač skutečně zpracovat.

Datové typy

- pravdivostní hodnota - *boolean*
- číselné - *integer*, *float*
- znakové - *character*, *string*
- definice pole - *array*
- definice typu funkce - *function*
- definice uživatelských typů - *typedef*

Konstanty

- pravdivostní hodnota – *true, false*
- číselné
  - uživatelské - *integer, float, number*
  - sémantické - *one, zero*
- znakové
  - uživatelské - *character, string*
  - sémantické - *newline*

Programové konstrukce

- definice programu - *program*
- definice funkcí v programu - *function*
- větvení - *if*
- cykly - *loop, repeat*
- návrat z funkce – *return*
- příkaz výstupu – *output*
- operátory
  - základní aritmetické (sčítání, odečítání, násobení, dělení, zbytek po dělení)
  - všech 6 relačních
  - logické (negace)

Typy programů

- */application/cmdline* – program pro příkazovou řádku

Optimalizace v procesu překladu

- vyhodnocování konstantních výrazů
- eliminace mrtvého kódu
- rozvinutí smyček
- spojení příkazů výstupu

4.3.2 Simultánní překlad

Simultánní překlad je vlastnost, která nám zabezpečí výsledek překladu v co nejkratším čase, protože překlad bude spuštěn automaticky na pozadí. V dnešních překladačích jsem se s touto vlastností nesetkal a proces překladu byl vždy uskutečněn až na základě požadavku programátora, který vlastním spuštěním ztratí určité množství času a hlavně je jeho práce pozdržena po dobu vlastního překladu. V projektu sémantického programování tedy byl zvolen přístup jiný, založený na automatickém spuštění překladu v případě potřeby, tj. např. když se změní zdrojové kódy.

Implementace simultánního překladu založeného na změnách ve zdrojových kódech přináší řadu výhod. Především jde o ušetření času potřebného k ručnímu spuštění procesu překladu, který se provádí automaticky pro všechny změněné části. S tímto pak souvisí vyšší efektivita, protože se překládají jen části, které je třeba skutečně přeložit znova. Z pohledu programátora se jedná o jeho menší zátěž, protože se nemusí starat o překlad a konečný výsledek překladu (spustitelná aplikace nebo seznam kritických chyb) je mu vždy k dispozici v nejkratším možném čase.

Jako každá věc, i simultánní překlad má nevýhody, mezi které patří zejména jeho značná složitost a náročnost implementace. Tento problém je naštěstí jednorázově vyřešitelný pomocí podrobné analýzy, v důsledku čehož tedy získáme za jednorázovou investici nepřetržité zlepšení v procesu vývoje aplikací.

4.4 Volby překladače

Překladač disponuje řadou volitelných nastavení, která ovlivňují zejména proces optimalizace. Tato nastavení jsou v prezentované prvotní verzi překladače aplikovatelná pouze na celý zdrojový XML soubor. V další verzi bude možno nastavení specifikovat lokálně – pro každý podstrom ve stromové struktuře dokumentu.

Nastavení jsou ukládána ve formě XML elementů se jménem *option* a jmenným prostorem *compiler*. Každé nastavení má své jméno a hodnotu, která je buď pravdivostní hodnotou (zda je volba zapnuta nebo vypnuta, indikováno hodnotami 0 a 1), nebo číselnou hodnotou (hodnota parametru). Příklad rozpoznávaných parametrů:

```
<compiler:option name="Convert loop to repeat">0</compiler:option>
<compiler:option name="Unroll loops">10</compiler:option>
<compiler:option name="Skip dead code">0</compiler:option>
<compiler:option name="Join outputs">0</compiler:option>
```

První parametr, *Convert loop to repeat*, povoluje nebo zakazuje změnu konstrukce z cyklu na opakování v případě, že je počet iterací známý v době překladu. Druhým, *Unroll loops*, lze specifikovat počet iterací, které se rozbalí do kódu bez porovnání a skoků, případně jím lze rozbalování zakázat. Třetí, *Skip dead code,* nastavuje režim generování mrtvého kódu - nikdy nezvolené větve větvení. Poslední volba, *Join outputs*, ovlivňuje proces optimalizace výstupních operací, tj. spojování za sebou jdoucích příkazů výstupu do jednoho.

4.5 Implementované optimalizace

4.5.1 Vyhodnocování konstantních výrazů

Vyhodnocování konstantních výrazů je základní optimalizací, která se nachází ve většině překladačů, protože se na ní dají zakládat další optimalizace, jako eliminace mrtvého kódu nebo rozvinutí smyček.

Implementace samotné optimalizace spočívá v zavedení rozhraní se dvěmi metodami a jeho implementací ve všech uzlech interní stromové struktury, které jsou odvozeny od abstraktních tříd pro operátory a výrazy. Toto rozhraní je definováno jako:

```
interface mdfsConstant {

    function isConstant();  // boolean
    function getCValue();   // the value or null

}
```

První metoda, *isConstant*, je dotazem, zda má uzel konstantní hodnotu a druhá metoda, *getCValue,* slouží k získání této konstantní hodnoty příslušného uzlu. Obě metody jsou v elementárních výrazech (argument funkce, proměnná, hodnota iterace, konstanta, atd.) vyhodnoceny přímo, na rozdíl od operátorů, kde jsou vyhodnoceny za pomocí volání metod na operandy. Tady platí pravidlo, které říká, že když je některý z operandů nekonstantní, pak výsledek operace bude také nekonstantní. Z tohoto samozřejmě existují výjimky, např. násobení nulou, které lze provést i s nekonstantním výrazem, ale výsledek operace bude konstantní 0. Implementace pro abstraktní třídy operátorů a výrazů je viditelná zde:

```
abstract class codeOperator
    extends mdfsSyntaxNode
    implements mdfsConstant

abstract class codeExpression
    extends mdfsSyntaxNode
    implements mdfsConstant
```

Pro třídy reprezentující konstanty je zavedení této optimalizace z poloviny řešena globálně, u abstraktní třídy pro konstanty. Celá definice této třídy je následující:

```
abstract class mdfsConstNode
    extends mdfsSyntaxNode
    implements mdfsConstant
{

    function isConstant() {
        return true;
    }

}
```

Z uvedeného kódu je zřejmé, že konstanty mají skutečně konstantní hodnoty.

4.5.2 Eliminace mrtvého kódu

Na základě předchozí pomocné optimalizace, vyhodnocování konstantních výrazů, lze provádět eliminaci mrtvého kódu ve větvení. Metoda spočívá ve vyhodnocení podmínky, která v případě, že je konstantní, indikuje provedení jen jedné větve. Přesná větev se pak určí pomocí vlastní konstantní hodnoty podmínky.

Implementaci zajišťuje metoda *simplify*, která vrátí buďto vlastní uzel příkazu větvení, nebo jen uzly příslušející sekvenci příkazů v sekci *then* nebo *else* dle konstantní hodnoty podmínky:

```
function simplify() { //class codeStatementIf

    if ( $this->condition->isConstant() ) {

        // constant condition
        $value = $this->condition->getCValue();

        // then or else?
        if ( $value === true  ) return $this->then->statements;
        if ( $value === false ) return $this->else->statements;

    }

    // not constant or error
    return $this;
}
```

Eliminaci mrtvého kódu lze pomocí volby překladače *Skip dead code* uživatelsky zapnout i vypnout. Ve výchozím nastavení překladače je zapnuta. Pro praktickou ukázku mějme následující fragment kódu v sémantické notaci:

```
<if>
    <cond>
        <lt><const:int>1</const:int><const:int>2</const:int></lt>
    </cond>
    <then>
        <write><const:str>1 &lt; 2</const:str></write>
    </then>
    <else>
        <write><const:str>1 &gt;= 2</const:str></write>
    </else>
</if>
```

Výsledný kód bez zapnuté optimalizace byl syntetizován ve tvaru:

```
if ((1<2)) {
    printf("1 < 2");
} else {
    printf("1 >= 2");
}
```

Se zapnutou optimalizací byl celý uzel větvení nahrazen obsahem jedné ze dvou větví, která bude skutečně prováděna:

```
printf("1 < 2");
```

4.5.3 Rozvinutí smyček

Rozvinutí smyček je optimalizací, která slouží primárně pro zvýšení výkonu programu, protože namísto opětovného kontrolování podmínky a skoku na začátek těla smyčky se budou provádět jen samotná těla smyček, v počtu dle počtu iterací.

Optimalizace je implementována v konstrukci *repeat*, která obsahuje vždy konstantní počet opakování. Když je samotný počet opakování menší nebo rovný hodnotě volby *Unroll loops*, smyčka se rozepíše na jednotlivé příkazy. V opačném případě bude syntetizována standardním způsobem, např. pomocí počítadla iterací. Ve zdrojovém kódu je podmínka testována opačně:

```
function simplify() { // class codeStatementRepeat

    // does it fit unrolling?
    if ($this->count > self::unrollLoops()) return $this;

    // unroll
    $out = array();
    for($i=1;$i<=$this->count;$i++) {
        foreach($this->body->statements as $stmt) {
            $out[] = $stmt;
        }
    }

    return $out;
}

static function unrollLoops() {
    return compilerOption::get('Unroll loops',1);
}
```

Ukázkový příklad pro tuto optimalizaci je fragment programu který vypíše pět znaků X. V sémantické notaci má tuto podobu:

```
<repeat count="5">
    <write>
        <const:chr>X</const:chr>
    </write>
</loop>
```

Výsledek v jazyce C syntetizovaný standardní metodou (při vypnuté optimalizaci, nebo počtu rozvinutí menším než pět) by byl:

```
/* repeat 5x */ {
    int repeat1 = 5;
    while( repeat1-- ) {
        printf("X");
    }
}
```

Po zapnutí optimalizace (na počet rozvinutí pět nebo více) je syntetizován následující kód:

```
printf("X");
printf("X");
printf("X");
printf("X");
printf("X");
```

4.5.4 Spojení příkazů výstupu

Poslední z optimalizací, která také může zvýšit výkon aplikace, je sjednocení příkazů výstupu nacházejících se za sebou. Implementace této optimalizace není přímo v příkazu výstupu, ale je v abstraktní třídě reprezentující skupinu příkazů, protože optimalizace zahrnuje více uzlů s příkazy.

Výsledný efekt lze předvést na předchozím příkladu. Mějme stejný zdrojový kód, vypisující pět znaků X pomocí smyčky od 1 do 5 a navíc se ještě vypíše konec řádku. Sémantická notace celého těla tohoto programu bude tedy:

```
<body>
    <loop id="number">
        <from><const:int>1</const:int></from>
        <to><const:int>5</const:int></to>
        <body>
            <write><const:chr>X</const:chr></write>
        </body>
    </loop>
    <write><const:newline /></write>
</body>
```

Bez optimalizací se syntetizuje tento zdrojový kód:

```
/* loop */ {
    int number;
    for(number=(1);number<=(5);number++) {
        printf("X");
    }
}
printf("\n");
```

Interně se převede smyčka *loop* na smyčku *repeat*, protože obsahuje konstantní výrazy:

```
/* repeat 5x */ {
    int repeat1 = 5;
    while( repeat1-- ) {
        printf("X");
    }
}
printf("\n");
```

Po rozbalení smyček (nutno explicitně povolit na úroveň alespoň 5) je syntetizováno:

```
printf("X");
printf("X");
printf("X");
printf("X");
printf("X");
printf("\n");
```

A konečná optimalizace spojení příkazů nám zabezpečí jediné volání funkce pro výstup, což sebou nese snížení zátěže z důvodu režie provedení samotného volání funkce:

```
printf("XXXXX\n");
```

5 Závěr

Na rozdíl od semestrálního projektu, ve kterém zcela scházela zpětná vazba z implementační fáze nebo od uživatelů, po práci na tomto projektu je cesta k realizaci mé vize o ideálním programovacím jazyku a vývojovém systému o mnoho jasnější.

5.1 Výhody

V průběhu práce bylo identifikováno několik výhod. Sémantická notace, jakožto abstraktní a rozšiřitelný jazyk, je vhodným prvkem k modelování programových systémů, což podporuje i otevřená architektura založená na ukládání dat ve formě XML. Pomocí jednoduchých skriptů nebo XSL transformací (XSLT) lze z abstraktních modelů generovat kód v sémantické notaci a ten pak přeložit do spustitelného tvaru pomocí nyní již existujícího i když neúplného překladače.

Jinou výhodou, kterou sémantické programování přináší, je skutečná nezávislost na platformě, protože překlad kódu není omezen jen na spustitelné soubory (binární, s instrukcemi procesoru), ale je také možno generovat různé skripty, určené pro webové servery a webové prohlížeče, kde se z bezpečnostních důvodů vylučuje možnost spouštět uvedené binární tvary. Tuto výhodu lze aplikovat také obráceně. Kód, ze kterého se generuje skript vykonávaný interpretem, lze později syntetizovat do kompilované formy a tím např. zvýšit výkon dané aplikace. Toto zvýšení výkonu lze tady provést algoritmicky, na rozdíl od současného stavu v komerční sféře, kde je potřeba najmout a zaplatit si programátory, kteří manuálně implementují aplikaci v zdánlivě kompilovaném jazyce (Java), který nikdy nebude podávat takové výkony jako nativní aplikace.

Další, a téměř určitě ne poslední, výhodou je škálovatelnost systému. Pomocí sémantického programování je potencionálně možno tvořit programy jak pro jednoduché mikrokontroléry na jednom čipu, tak aplikace unixovského typu, pro příkazový řádek, dále také jednouživatelské aplikace s grafickým rozhraním, nebo velké distribuované systémy (ekonomické aplikace apod.), jejichž tvorba je založena na modelech a algoritmické implementaci těchto modelů.

5.2 Obecné nevýhody

Kromě výhod má sémantické programovaní také své nevýhody. Z hlediska programátora jde o nový způsob zápisu programových konstrukcí – pomocí značkování. Nejblíže bude nová notace programátorům, kteří již jsou seznámeni s formátem XML, HTML nebo XHTML. Podobnost sémantické notace lze nalézt také při porovnávání s formou zápisu programů v jazyce LISP (používá prefixovou notaci), ale tento jazyk není v naší lokalitě příliš rozšířen, takže programátoři jeho závorkovou a vnořenou syntaxi neznají. Tato nevýhoda ve formátu zápisu je odstranitelná pomocí použití jiného uživatelského rozhraní, např. použitím grafického vývojového prostředí (angl. *IDE – Integrated development environment*). Grafické prostředí je pro rozumnou úroveň práce v případě sémantického programování nevyhnutelné.

Další nevýhodou plynoucí z nové metody zápisu je rozdělení programu na několik souborů, protože definice typů funkce jsou oddělené fragmenty nacházející se v odděleném souboru. Tímto narůstá režie při manuální správě zdrojových kódů, ale u uživatelského rozhraní na vyšší úrovni lze ale předpokládat automatické dělení na soubory.

Problémem je také to, že zde definované sémantické programování neposkytuje možnost použití stávajících knihoven, takže všechen kód je potřeba potencionálně vytvořit znovu. Toto ale nelze požadovat ani očekávat od programátorů, a proto je nutno navrhnout rozšíření poskytující možnost importu existujících knihoven. Hlavním prvkem tohoto rozšíření bude automatický import hlavičkových souborů a údržba informací o souvisejících knihovnách.

Vlastnost, kterou sémantické programování samo o sobě neřeší, se týká postupného vývoje, produkujícího různé verze programů. V sémantickém programování je předpokládáno použití repositáře, obsahujícího všechny fragmenty v jediné verzi – aktuální. Pro skutečné programování tento přístup není příliš vhodný, protože je často potřeba sledovat změny ve zdrojových kódech a přistupovat k jejich starším verzím, např. z důvodu řešení chyb.

V současných programovacích jazycích se také nachází podmíněný překlad, kterým lze z jediného zdrojového kódu produkovat různé varianty programů, obsahujících odlišné množství vlastností. Tato podmíněnost je řešena výlučně v syntaktické rovině. V sémantickém programování tato možnost sice chybí, ale lze ji nahradit pomocí jiných technologií, např. XSL transformacemi, nebo pak v sémantické rovině lze podmínit překlad za pomocí větvení s konstantní podmínkou a zapnutím eliminace mrtvého kódu.

5.3 Nedostatky v návrhu

V závěru práce na projektu se našlo několik chyb, které bude potřeba napravit. Jedná se např. o datový typ řetězec, který v současné době nesouvisí s typem znaků. Sémantická hodnota řetězce je ale posloupnost znaků, takže by bylo vhodnější pro tvorbu řetězců vytvořit nový typový operátor – zřetězení znaků.

U datového typu množina, který v překladači ještě nebyl implementován, se objevila možnost, že by mohl být vytvořen i nad jiným typem než výčtem, což vyplývá z podstaty sémantické notace (nezávislost na implementaci). Skutečná implementace v reálním prostředí si u této vlastnosti vyžádá obslužný kód založený na seznamech, který bude řádově pomalejší, než implementace pomocí bitového pole. U množin se dále předpokládá práce pomocí operátorů sjednocení, průniku, rozdílu a operátor vracejícího pravdivostní hodnotu jestli existuje prvek v množině. Tyto operátory ale nejsou ještě definovány.

Příkaz nebo operátor výstupu je definován pouze jako pomocný, protože bylo nutné zobrazit výsledek. Skutečné programy pracují se vstupem a výstupem na vyšší úrovni, protože výstupy existují standardně dva (*stdout, stderr)* a pracuje se s nimi na základě určité abstrakce, tj. jako s toky (angl. *streams*). Když si situaci prozkoumáme blíže, zjistíme že tyto toky jsou definovány v unixových operačních systémů pomocí tabulky *file deskriptorů*. V sémantickém programování by tedy toto pole mělo být dostupné přes určité standardní rozhraní.

Sporné je také rozdělení, jestli má být operace provedena voláním funkce nebo aplikací operátoru. Protože existuje hodně případů, ve kterých je lepší zápis pomocí operátorů (viz matematické funkce), bude zapotřebí definovat způsob provázání těchto dvou přístupů.

Poslední vlastností, která by v sémantickém programování měla smysl, ale nebyla zatím definována je pojmenování dimenzí u vícerozměrných polí, obdobně jako existuje pojmenování parametrů funkce. Tato vlastnost umožňuje u volání funkce uvést parametry mimo pořadí a překladač si je uspořádá správně. Analogicky, u polí, se pojmenování dimenzí projeví až při indexaci, kde bude možno uvést indexy mimo pořadí. Prakticky to znamená, že si programátor u definice matice pojmenuje dimenze např. jako řádek a sloupec. Při indexaci prvku pak určí řádkový a sloupcový index a to, který index je první a který druhý vyřeší již sám překladač.

Jak je vidět, nedostatků je celá řada. Tyto nevýhody lze ale poměrně snadno řešit, např. dodefinováním značení umožňujícího implementovat příslušné vlastnosti (podmíněný překlad, použití stávajících knihoven apod.). Snadnost řešení problémů také souvisí s tím, že je samotný sémantický jazyk v poměrně rané fázi vývoje a je zde veliký prostor pro vylepšování (tj. jsou dovoleny větší změny v definici a není potřeba zachovávat zpětnou kompatibilitu).

5.4 Budoucí vývoj

Pro úspěch projektu v budoucnu je potřeba si stanovit oblast, na kterou bude zaměřena praktická realizace překladačů. Nejvýhodnější bude se zaměřit na krajní případy (extrémy).

Na jedné straně spektra totiž stojí velké aplikace (ekonomické informační systémy) a na druhé je podpora pro vestavěné (angl. *embedded*) aplikace (používající mikrokontroléry). U těchto se prostřednictvím zaměření na sémantické vlastnosti nabízí možnost modelovat a navrhovat zároveň jak technickou, tak i programovou část. Mezi uvedenými extrémy je místo pro vývoj obecných (jednoduchých) programů, ale zde existuje dostatek alternativ a není potřeba nového jazyka.

Podrobnější plán do budoucna obecně zahrnuje vytvoření společného grafického vývojového prostředí pro zadávání a úpravu programů v sémantické notaci. Z tohoto prostředí bude možno provádět nasazení aplikací přímo do cílového prostředí, např. se zabezpečí nahrání skriptů na webový server přes FTP. Také zde bude možnost po opravě chyb v generování kódu znovu automaticky syntetizovat a nasadit všechny instance aplikace, které vývojář nebo dodavatelská společnost spravuje. Tímto se dosáhne lepší podpory, než je tomu v současných službách, kde jsou opravy manuální a tím také relativně drahé.

U vestavěných aplikací se pak naskytuje možnost provádět návrh technické části zařízení – v sémantické notaci zapsat schéma zapojení a z ní navrhnout desku tištěných spojů. Následně bude vytvořena programová část *firmware*, který bude programátor tvořit na stejně vysoké úrovni jako ostatní typy aplikací (např. pomocí modelování) a překlad do instrukcí procesoru zabezpečí překladač.

Neposlední aplikací jsou pak různé simulátory, což jsou ve své podstatě interprety sémantické notace. Když myšlenku interpretování rozšíříme, můžeme vytvořit a zpřístupnit objektové rozhraní na způsob DOM (*Document object model*), který slouží na úpravu obsahu webových stránek za běhu. V sémantickém programování by to bylo aplikováno na program samotný, který se bude moci sám upravit. Prostředí jako stvořené pro umělou inteligenci.

Který extrém bude v budoucnosti skutečně dominovat sémantickému programování je zatím otevřená otázka a závisí také na poptávce a možnostech následného komerčního využití, na které se v reálním světě nesmí zapomenout.

Seznam použité literatury

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

[2] 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/

[3] Gordon, M. J. C.: *Programming language theory and its implementation*.
Prentice-Hall, 1988, ISBN 0-13-730417-X.

[4] Bradley, N.: *XML - kompletní průvodce*.
Grada Publishing, 2000, ISBN 80-7169-949-7.

[5] *Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation.*
World Wide Web consortium, 2004.
Dokument dostupný z URL
http://www.w3.org/TR/2004/REC-xml-20040204/

[6] *IEEE 754-1985: Standard for Binary Floating-Point Arithmetic.*
Institute of Electrical and Electronics Engineers, 1985.
Dokumenty dostupné z URL
http://grouper.ieee.org/groups/754/

Uvedené online zdroje byly dostupné v době psaní této práce (duben 2006).

Jiné zdroje

[7] Microsoft Research: *Intentional programming – promotional video.*
Audiovizuální záznam, v době psaní práce již nebyl dostupný.
Původní URL (prosinec 2005):
http://www.cse.unsw.edu.au/~cs3141/ip.asf