Vývoj doménovo-špecifických jazykov

Jaroslav Porubän

Vývoj doménovo-špecifických jazykov

Jaroslav Porubän

Recenzenti:
Ing. Peter Václavík, PhD.
Ing. Milan Nosáľ, PhD.

Vydanie tejto vysokoškolskej učebnice bolo podporené projektom KEGA č. 053TUKE-4/2019 - *Výučba softvérového inžinierstva prostredníctvom sústavných výziev a súťaží*.

Technická univerzita v Košiciach, 2020
1. vydanie

ISBN 978-80-553-3505-6

Obsah

1 Úvod

Koľko jazykov ovládaš, toľko krát si človekom.
Ľudová múdrosť

Jazyky zohrávajú kľúčovú úlohu v ľudskej spoločnosti. Sú základným prostriedkom komunikácie. Umožňujú zdieľať informácie medzi ľuďmi ako aj uložiť informácie pre neskoršie použitie. Sú kľúčom k pochopeniu minulosti, ktorá umožňuje pochopiť súčasnosť a vytárať rozhodnutia do budúcnosti. Jazyk je súčasťou národnej kultúry a je zároveň jeho pamäťou.

Prirodzené jazyky umožňujúce komunikáciu medzi ľuďmi prešli dlhú cestu zmien kým dospeli do súčasného stavu. Jazyky však nedospeli do finálnej podoby a nie sú statickými entitami. Vyvíjajú sa spolu s ľudskou spoločnosťou. Niektoré slová nám už dnes nič nepovedia, niektoré slová zmenili význam a vzniklo veľké množstvo nových slov. Jazyky sú zrkadlom progresu v ľudskej spoločnosti. Opakujúce sa vzory v ľudskej spoločnosti a prostredí, v ktorom spoločnosť existuje, dostávajú svoje pomenovanie. Vzniklo mnoho nových slov. Sú to napríklad slová robot, počítač, softvér. V jazyku je zvyčajne možné identifikovať taktiež vplyv iných koexistujúcich jazykov.

Jazyky majú rôznu konkrétnu formu – zvukovú, textovú, grafickú, posunkovú, mimickú. To isté slovo z pohľadu významu môže mať viacero rôznych reprezentácií. Veď radosť je možné vyjadriť slovom, písmom, mimikou pier a očí ale aj zvukovým prejavom.

V minulom storočí vznikla úplne nová kategória jazykov určená na komunikáciu ľudí s technickým fenoménom doby – počítačom. Tieto umelé jazyky vytvorené ľuďmi je možné súhrnne označiť ako počítačové jazyky. Počítačové jazyky oproti prirodzeným jazykom majú svoje špecifiká práve kvôli úlohe počítačov v komunikácii. Hlavne v začiatkoch éry počítačov boli tieto jazyky stavané pre počítače a človek sa musel prispôsobiť nízkej úrovni abstrakcie v týchto jazykoch. Postupne s vývojom hardvéru sa menil aj charakter jazykov posúvajúc sa bližšie k mysleniu ľudí. Dnes je zrejmé, že počítačový jazyk nemôže obmedzovať človeka v myslení ale musí umožniť jednoducho zapísať myšlienky a ciele ľudí pri realizácii systémov a musí umožňovať efektívne spolupracovať veľkým tímom pri vývoji systémov zjednodušením komunikácie. Dobrý počítačový jazyk je „intuitívny“ – jednoduchý na použitie a pochopenie, ktorý používateľa neobmedzuje ale zároveň sa pri jeho používaní nestratí v množstve detailov.

Táto vysokoškolská učebnica ponúka širší pohľad na tému vývoja doménovo-špecifických jazykov a ich aplikácie v softvérovom inžinierstve, či už ako súčastí softvérového riešenia alebo ako prístupu k budovaniu softvéru v kontexte modelom riadeného softvérového vývoja. Učebnica zároveň opisuje naše originálne výsledky v oblasti moderného vývoja počítačových jazykov a nástroj YAJCo na inkrementálnu tvorbu počítačových jazykov. Prostredníctvom viacerých príkladov v učebnici vysvetľujeme ako tvoriť doménovo-špecifické jazyky použitím rôznych prístupov a nástrojov.

Na tomto mieste by som chcel poďakovať recenzentom Ing. Petrovi Václavíkovi, PhD. a Ing. Milanovi Nosáľovi, PhD. za ich spoluprácu, ktorá rozhodne viedla k zvýšeniu kvality tejto vysokoškolskej učebnice. Ich názor si veľmi cením aj vzhľadom na ich rozsiahle skúsenosti z praxe v oblasti softvérového inžinierstva. Taktiež by som chcel poďakovať kolegom z nášho výskumného tímu SEUG, ktorí boli pre mňa inšpiráciou a tiež ma motivovali k ďalšiemu výskum v tejto oblasti.

2 Jazyky

Jazyk zohráva kľúčovú úlohu v ľudskej spoločnosti. Je to základný prostriedok ľudskej komunikácie. Výkladový slovník Merriam-Webster [76] definuje pojem jazyk ako skupinu slov a metód kombinácie týchto slov, ktorý používa a zároveň mu rozumie komunita ľudí. Samotný jazyk určuje efektivitu s akou bude komunikácia prebiehať.

Jazyk je úzko spätý s myslením. V jazykovede a psychológii sa s pojmom jazyka spájajú dve zložky: konotát a denotát. Denotát označuje „objektívny" význam pojmu, ktorý zdieľa komunita používajúca jazyk. Denotát je neformálne definovaný výkladovými slovníkmi jazyka. Konotát označuje subjektívnu skúsenosť osoby s predmetom, ktorý pojem opisuje. Denotát je daný dohodou zúčastnených v komunite, mnohokrát na základe princípu autority. V prirodzených jazykoch je teda nie vždy jednoznačne určený význam slova, pretože komunikácia je ovplyvnená našou osobnou skúsenosťou.

Prirodzené jazyky sa vyvíjali mnoho rokov a sú určené na komunikáciu medzi ľuďmi. Na druhej strane existujú umelé jazyky vytvorené ľuďmi, spomedzi ktorých sú najvýznamnejšie jazyky počítačové. Počítačové jazyky sú formálne jazyky určené na komunikáciu človek a počítačový systém, respektíve na interakciu medzi počítačovými systémami. Vety vo formálnych jazykoch majú formálne definovanú štruktúru a význam.

2.1. Formálne jazyky

Formálne jazyky majú kľúčový význam pre jednoznačnú komunikáciu zúčastnených strán. Definíciu formálnych jazykov je možné nájsť v [43].

Abeceda (slovník) je ľubovoľná konečná množina symbolov. *Slovo* (reťazec, veta) nad abecedou je ľubovoľná postupnosť konečnej dĺžky, ktorá je zložená zo symbolov tejto abecedy. Prázdne slovo je slovo, ktoré neobsahuje nijaký symbol a označuje sa ε. *Jazyk* je ľubovoľná množina slov nad nejakou abecedou. Ak Σ je abeceda, potom Σ^* označuje množinu všetkých slov, ktoré sa skladajú zo symbolov abecedy Σ, vrátane prázdneho slova ε. Formálny jazyk je definovaný v [43] nasledovne:

Formálny jazyk L nad abecedou Σ je ľubovoľná podmnožina Σ^*.

Formálny jazyk teda môže byť definovaný vymenovaním jeho viet, alebo pomocou definície vlastností, ktoré musí spĺňať každá veta jazyka. Jedným zo spôsobom definície jazyka je použitie generatívnych systémov – gramatík. Tento spôsob opisu formálnych jazykov je významný v oblasti jazykových procesorov hlavne kvôli existujúcim implementáciám generátorov jazykových procesorov založených na teórii bezkontextových gramatík.

Gramatika G je štvorica $G = (N, T, P, S)$, kde N je konečná množina neterminálnych symbolov, T je konečná množina terminálnych symbolov, pričom $N \cap T = \emptyset$, P je množina produkčných (prepisovacích) pravidiel, ktorá je konečnou podmnožinou množiny $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ a S je začiatočný symbol $S \in N$. Ak je jazyk L špecifikovaný gramatikou G používa sa zápis $L(G)$.

Z tejto definície formálnych jazykov podľa [43] je zrejmá orientácia na textovú formu reprezentácie jazykov vyjadriteľnú ako postupnosť symbolov. Formálne jazyky podľa tejto definície predstavujú syntaktické zápisy bez priamej väzby na sémantiku. V tejto práci sú formálne jazyky chápané ako jazyky, ktoré sú určené pre automatizované spracovanie.

2.2. Počítačové jazyky

Pojem počítačový jazyk zahŕňa všetky druhy umelých jazykov používaných v počítačových systémoch a v softvérovom inžinierstve, vrátane programovacích jazykov, doménovo-špecifických jazykov [73], údajových modelov, ontológií [67][93] a modelovacích jazykov [53]. Počítačové jazyky sú formálnymi jazykmi. Počítačový jazyk nie je vždy explicitne opísaný gramatikou alebo metamodelom s oddelenou definíciou konkrétnej a abstraktnej syntaxe a sémantiky. Mnohokrát sú tieto jazyky roztrúsené v zdrojových kódoch, návrhových vzoroch a používateľských rozhraniach. Niekedy je používaný pojem počítačový jazyk aj na označenie množiny návrhových vzorov a vytvorených aplikačných rozhraní [47].

Inžinierstvo počítačových jazykov predstavuje aplikáciu systematického, riadeného a merateľného postupu pre vývoj, používanie a údržbu počítačových jazykov [47]. Inžinierstvo počítačových jazykov postihuje všetky fázy životného cyklu počítačových jazykov vrátane návrhu, implementácie, dokumentovania, testovania, nasadenia, evolúcie a údržby. V centre výskumu sú nástroje, technológie, postupy a formálne metódy podporujúce návrh a implementáciu počítačových jazykov. Inžinierstvo počítačových jazykov vníma počítačové jazyky ako softvérové artefakty (podobne ako programy) s dôrazom na formu opisu jazyka s cieľom uplatňovania princípov a metód akými sú modularizácia, kompozícia, refaktorizácia, separácia, evolúcia a koevolúcia. Medzi hlavné predmety záujmu patria:

- formálne aparáty používané pri návrhu a špecifikácií počítačových jazykov: gramatiky [54], metamodely, ontológie,
- metódy a implementácie jazykových procesorov: zdokonalenia tradičných generátorov jazykových procesorov, systémy atribuovaných gramatík [55][56], systémy prepisovania výrokov [9],
- nástroje pre transformáciu programov: refaktorizačné nástroje [80][81][P25], nástroje pre generovanie softvérových systémov [17], nástroje pre extrakciu modelov,
- nástroje pre kompozíciu, integráciu jazykov a transformáciu medzi rôznymi formami konkrétnej reprezentácie a pre transformáciu modelov [34][77],
- nástroje a metódy pre podporu evolúcie systémov a jazykov [35].

2.3. Jazyky v softvérovom inžinierstve

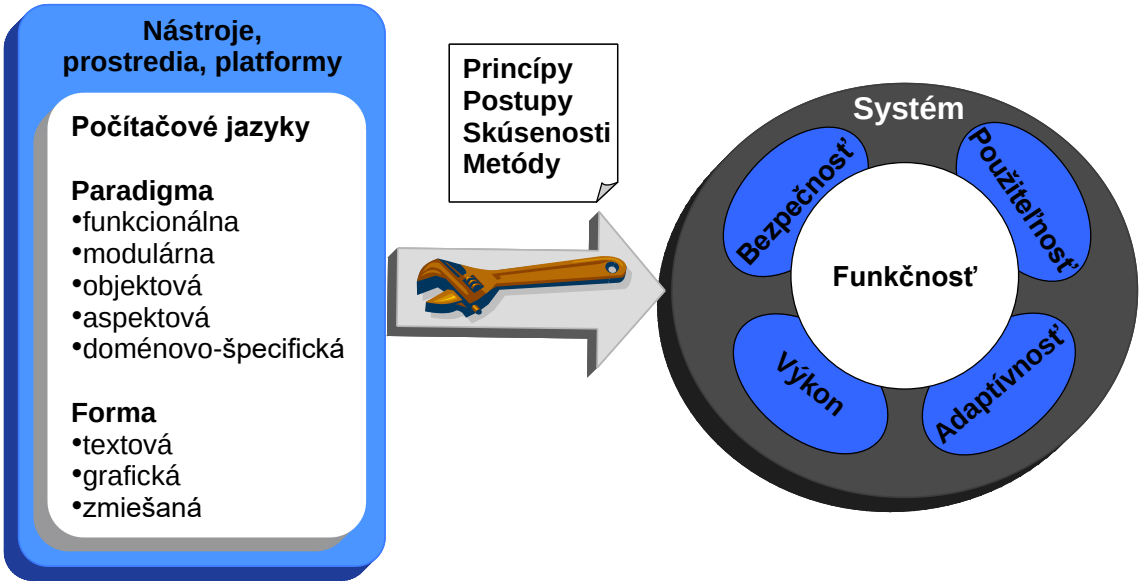
Počítačové jazyky zohrávajú kľúčovú úlohu v softvérovom inžinierstve. Počítačový jazyk umožňuje vytvoriť opis softvérového systému a slúži na vyjadrenie cieľov a zámerov autorov softvérových systémov za použitia princípov, metód, skúseností a overených postupov. Postavenie počítačových jazykov v softvérovom inžinierstve zobrazuje Obr. 1. Jazyk je základným nástrojom na vývoj softvéru.

Pri vývoji a implementácii počítačových jazykov sa informatika a softvérové inžinierstvo sústreďujú na dva základné problémy:

- Prechod od neformálnej špecifikácie vyjadrujúcej požiadavky na systém k formálnej špecifikácii požiadaviek.
- Prechod od formálnej špecifikácie systému k formálnej implementácii systému.

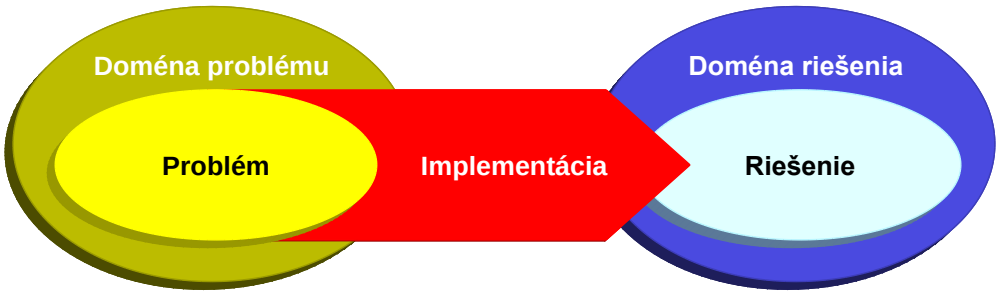
Cieľom vývoja softvérového systému je získať implementovaný systém, ktorý zodpovedá špecifikovaným požiadavkám. Pri prechode od formálnej špecifikácie ku formálnej

implementácii sú matematika a formálne metódy vhodným aparátom. Pri prechode od neformálnych požiadaviek k formálnym požiadavkám sa stretávajú dva svety. Svet expertov v doméne, pre ktorú je implementovaný systém a svet expertov, ktorí systém implementujú. Kľúčová je schopnosť doménového experta formalizovať požiadavky na systém v spolupráci s autormi systému. Formalizácia požiadaviek je prerekvizitou použitia formálnych metód pre vývoj softvéru.



Obr. 1: Postavenie počítačových jazykov v softvérovom inžinierstve

Pri implementácii softvérových systémov je teda možné uvažovať o dvoch doménach (Obr. 2). Doména, v ktorej sa riešený problém vyskytuje (napr. doména patologickej medicíny) a doména, v ktorej je problém riešený (napr. počítačový systém). V doméne problému je používaný jazyk na opis problému (napr. terminológia patologickej medicíny), zatiaľ čo v doméne riešenia je použitý jazyk riešenia problému (napr. objektový jazyk). Táto transformácia (implementácia) musí zachovávať požadované vlastnosti definované v špecifikácii požiadaviek, ktoré sú vyjadrené jazykom domény problému.



Obr. 2: Implementácia softvérových systémov

2.4. Paradigmy programovacích jazykov

Paradigma programovacieho jazyka predstavuje základný spôsob myslenia v programovaní sprostredkovaný počítačovým jazykom. Paradigmy sa navzájom odlišujú

pojmami a abstrakciami, ktoré sú použité na reprezentáciu viet vytvorených prostredníctvom programovacieho jazyka – programov.

Paradigmy programovacích jazykov sa spájajú s jazykmi všeobecného použitia (GPL), v ktorých vymedzujú charakter jazyka. Jazyky všeobecného použitia umožňujú zapísať ľubovoľný algoritmus a ich vyjadrovacia sila je ekvivalentná Turingovmu stroju, označujú sa ako Turing-úplné [12].

Pri implementácii softvérového systému prostredníctvom počítačového jazyka musí programátor vhodne transformovať myšlienky z domény problému do domény riešenia aplikujúc pojmy a abstrakcie zvolenej paradigmy. Programátor teda musí byť stotožnený s myslením v danej paradigme. Paradigma jazyka sa odráža v konštrukciách konkrétnej aj abstraktnej syntaxe a sémantike. Rôzne počítačové jazyky tej istej paradigmy majú zvyčajne rôznu konkrétnu syntax. Abstraktná syntax, definujúca pojmy jazyka a ich štruktúru, je však často pre jazyky tej istej paradigmy viac podobná ako konkrétna syntax a s abstraktnou syntaxou je spätá aj sémantika konštrukcií jazyka. Preto zvyčajne nie je zložité prejsť z jedného programovacieho jazyka do iného v tej istej paradigme ale problémom je zmena paradigmy. Zmena paradigmy, ktorá je reprezentovaná programovacím jazykom, totiž znamená zmenu myslenia. Ak je upnutie programátora na danú paradigmu silné, dochádza k paradigmatickej paralýze. V takomto prípade sa snaží programátor vnímať problém prostredníctvom jednej zvolenej (obľúbenej) paradigmy napriek tomu, že to nemusí byť paradigma optimálna na postihnutie daného problému vzhľadom na jeho charakter.

Medzi najvýznamnejšie paradigmy v súčasnosti je možné považovať:

- objektovo-orientovaná paradigma [84],
- komponentovo-orientovaná paradigma [110],
- funkcionálna paradigma [19],
- paradigma logického programovania [19],
- aspektovo-orientovaná paradigma [24].

Nasledujúce časti opisujú dve paradigmy programovacích jazykov: funkcionálnu a objektovo-orientovanú.

2.4.1. Funkcionálne jazyky

Jednou z významných paradigiem programovania vo vedeckej komunite je funkcionálne programovanie [10][19][28][42][50]. Je to spôsobené hlavne podobnosťou paradigmy s matematickým zápisom ako aj deklaratívnym charakterom funkcionálnych programovacích jazykov. Tvorba programov funkcionálnym spôsobom je založená na uplatnení matematických metód [57]. Napriek tomu, že neexistuje všeobecne akceptovaná definícia funkcionálneho jazyka, za základný pojem je možné považovať funkciu, ktorá je vo funkcionálnych jazykoch podporovaná ako vstavaný údajový typ (angl. first-class value). Funkcionálne jazyky umožňujú používať funkcie vyššieho rádu – funkcia môže byť parametrom alebo návratovou hodnotou inej funkcie. Funkcionálne jazyky sa sústreďujú na hodnoty údajov opísané výrazmi, obsahujúce matematické definície a použitia funkcií s automatickým výpočtom výrazov. Funkcionálne jazyky sú založené na lambda kalkule [46] a podľa [44] je možné funkcionálne programovacie jazyky vnímať ako rozšírený lambda kalkulus.

Funkcionálne jazyky je možné rozdeliť do dvoch kategórií:

- čisté funkcionálne jazyky,

- špeciálne funkcionálne jazyky.

V *čistých funkcionálnych jazykoch* sa nevyskytujú priradenia, pojem premennej ako pamäťovej bunky, ani postupnosť krokov vykonávania [120]. Čisté funkcionálne jazyky umožňujú výpočet na základe požiadavky [121]. Medzi čisté funkcionálne jazyky patria napríklad jazyky Haskell [51][112], Clean [96], Miranda [117].

Medzi *špeciálne funkcionálne jazyky* sa zaraďujú funkcionálne jazyky so špeciálnymi konštrukciami - priradenia, výnimky, referenčný typ. Tieto jazyky sa nazývajú aj funkcionálne jazyky s vedľajšími účinkami [122]. Detailná definícia rozdielu čistých a špeciálnych funkcionálnych jazykov je uvedená v [101]. Medzi špeciálne funkcionálne jazyky sa zaradzujú napríklad jazyky Erlang [3], Scheme [18], Standard ML [85] ako aj funkcionálny jazyk pre platformu .Net s názvom F# [41], [107].

2.4.2. Objektovo-orientované jazyky

V súčasnosti je objektové programovanie najpopulárnejšou paradigmou používanou v softvérovom priemysle [86][98]. Na rozdiel od funkcionálneho programovania, ktoré je založené na matematických princípoch, objektové programovanie vzniklo s cieľom priblíženia programovania vnímaniu človeka, v ktorom vystupujú objekty ako identifikovateľné entity so správaním a stavom. Základným pojmom je objekt, ktorý je odrazom objektu z reálneho sveta na vhodnej úrovni abstrakcie [84].
Ani pre objektovo-orientovanú paradigmu neexistuje úplné a presné vymedzenie. Podstatu objektovo-orientovanej paradigmy je možné charakterizovať nasledovne [105]:

objektovo-orientovaný = abstraktné údajové typy + dedičnosť

Výhodou objektovo-orientovanej paradigmy je existencia metodologickej a nástrojovej podpory pre viaceré fázy životného systému softvérového systému vrátane analýzy a návrhu. Vzhľadom na vysokú popularitu objektovo-orientovanej paradigmy sa navrhnuté riešenie definície počítačových jazykov čiastočne opiera o túto paradigmu (časť 5).

2.5. Multiparadigmatické jazyky

Súčasné jazyky všeobecného použitia sú zvyčajne multiparadigmatické jazyky. Multiparadigmatické jazyky podporujú viac ako jednu paradigmu pre implementáciu softvérového systému. Takto umožňujú voľbu najvhodnejšej paradigmy pre implementáciu systému a to nielen pre implementáciu celého systému ale pre jeho jednotlivé časti. Časť systému je vyjadriteľná prostredníctvom paradigmy podporovanej multiparadigmatickým jazykom, ktorá pre danú časť umožňuje najjednoduchšie vyjadrenie podproblému.

Skutočnosť, že mnoho konštrukcií z jazykov môže byť vyjadrených prostredníctvom konštrukcií v iných jazykoch neznižuje význam vytvárania multiparadigmatických jazykov. Takéto jazyky poskytujú stručnejšie vyjadrenie riešení problémov použitím danej paradigmy [103].

Nasledujúci príklad je ukážkou implementácie pojmu funkcie známeho z funkcionálnych jazykov v objektovo-orientovanom jazyku, ktorý priamo nedefinuje pojem funkcie. Tento príklad prezentuje možnosť vyjadrenia funkcií v objektových jazykoch. Pojem funkcie je implementovaný prostredníctvom objektu anonymnej triedy. Riešeným problémom je výber reťazcov zo zoznamu, ktorých dĺžka (počet znakov) je menšia ako 5. Prvý zápis je ukážkou realizácie výberu prostredníctvom jazyka Haskell [51], ktorý definuje funkciu `filter`.

```
filter (\x -> length x < 5) ["Java", "C#", "Haskell", "Python"]
```

Druhý zápis predstavuje realizáciu riešenia problému prostredníctvom objektovo-orientovaného jazyka Java.

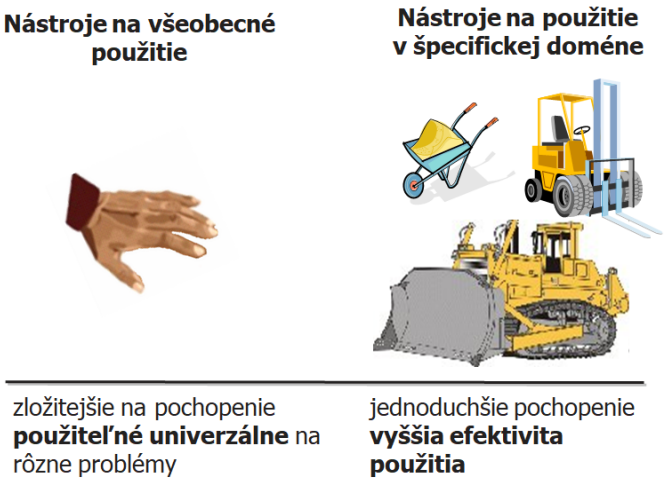
```
public interface Filter<E> {
    boolean test(E elem);
}

...

List<String> list = new ArrayList<String>();
list.add("Java");
list.add("C#");
list.add("Haskell");
list.add("Ruby");
list.add("Python");

list = filter(list, new Filter<String>() {
    public boolean test(String elem) {
        return elem.length() < 5;
    }
});
```

Z príkladu je zrejmé, že zápis funkcie v objektovo-orientovanom jazyku Java má viac riadkov ako zápis v paradigme, ktorá má definovaný pojem funkcie. Nevýhodou takéhoto prepisu nie je len stručnosť vyjadrovania. Takto reprezentovaný pojem môže byť problematické identifikovať vo vete (programátorom resp. automatizovane), lebo jeden pojem môže mať viacero vyjadrení a vyjadrenie môže byť rovnaké pre rôzne pojmi. Pri prepise pojmu v inej paradigme sa teda môže stratiť informácia o existencii výskytu pojmu vo vete ak nie je táto transformácia bijektívna.

2.6. Doménovo-špecifické jazyky

Doménovo-špecifický jazyk je navrhnutý s cieľom špecifikácie riešení v konkrétnej problémovej doméne na rozdiel od jazykov všeobecného použitia. Hlavným cieľom návrhu doménovo-špecifických jazykov je zvýšiť úroveň abstrakcie špecifikácie riešenia problému priblížením sa k doméne problému. Myšlienka vytvárania špecifických nástrojov je používaná aj v iných inžinierskych disciplínach (Obr. 3). Zvýšenie abstrakcie umožňuje programátorovi špecifikovať riešenie použitím pojmov a ich zápisov v štýle problémovej domény. Pri použití jazykov všeobecného použitia používa programátor z pohľadu domény problému zvyčajne relatívne nízku úroveň abstrakcie. V ideálnom prípade by doménovo-špecifický jazyk mohol byť jazyk domény problému a zároveň aj jazyk riešenia. Okrem použitia vyššej úrovne abstrakcie doménovo-špecifické jazyky prostredníctvom jazykových procesorov prinášajú zvyčajne možnosť efektívnejšej statickej analýzy programov vzhľadom na explicitné používanie pojmov z domény. Doménovo-špecifické jazyky sú v literatúre označované aj ako aplikačne-orientované jazyky [102], jazyky špeciálneho použitia [124], špecializované [8], úlohovo-orientované [87], aplikačné [75] a malé jazyky [7].

Doménovo-špecifický jazyk môže mať textovú ale aj grafickú (vizuálnu) konkrétnu reprezentáciu. Vizuálna reprezentácia sa hlavne kvôli postupnému vytváraniu nástrojov stáva stále populárnejšou. Textová reprezentácia prináša zvyčajne vyššiu produktivitu tvorby viet ako vizuálna. Jeden jazyk z pohľadu abstraktnej syntaxe môže mať viacero konkrétnych reprezentácii, z ktorých si zvolí používateľ najvhodnejšiu formu podľa charakteru riešeného problému a svojich schopností.

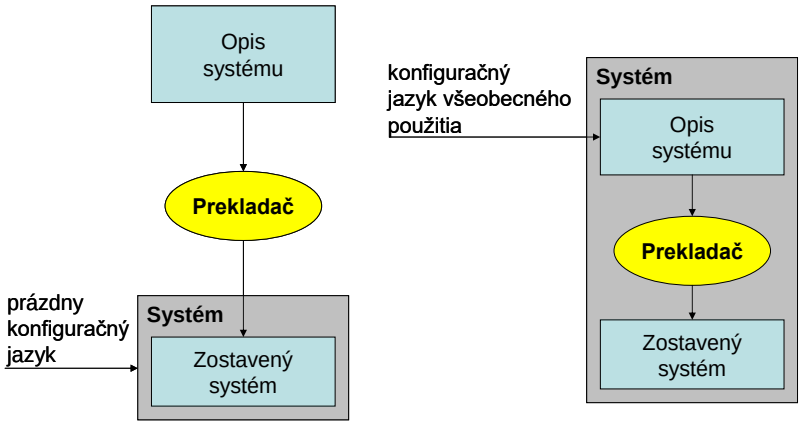


Obr. 3: Všeobecné a špecializované nástroje

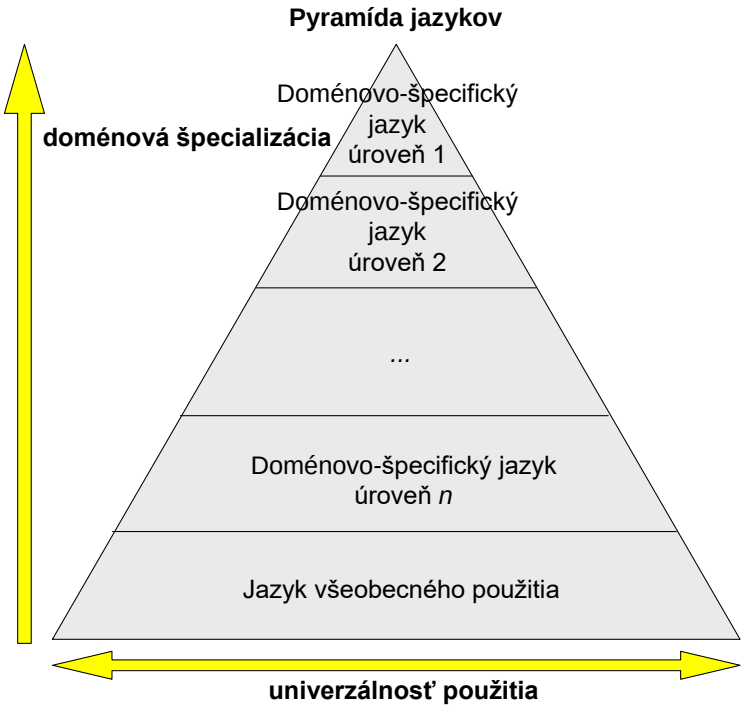
Podľa [16] si implementácia textových doménovo-špecifikých jazykov vyžaduje expertné skúsenosti v oblasti jazykových procesorov, gramatík a generátorov jazykových procesorov a je to často náročná úloha. Aj autori v článku [82] uvádzajú, že vytváranie doménovo-špecifikých jazykov je zložité, pretože kladie vysoké nároky na autorov DSL. Autor by mal mať expertné znalosti z domény ale aj skúsenosti s vývojom počítačových jazykov.

Doménovo-špecifické jazyky zohrávajú kľúčovú úlohu pri modifikácii softvérových systémov. Na Obr. 4 sú zobrazené dva pohľady na konfiguráciu softvérových systémov, ktoré vysvetľujú dôležitosť úrovne abstrakcie počítačových jazykov pre modifikovanie systému. V podstate ide o dva rôzne pohľady na ten istý systém. V prvom prípade je systém zostavený do výslednej podoby, ktorá neumožňuje modifikáciu (konfiguračný jazyk je prázdny jazyk). V druhom prípade je systém meniteľný na základe použitia jazyka všeobecného použitia umožňujúc významné zmeny v systéme. Modifikáciu je nutné realizovať v jazyku domény riešenia (jazyk všeobecného použitia) a nie v jazyku domény problému, čo vyžaduje patričného experta z oboch domén.

Z tohto dôvodu je možné dospieť k myšlienke viacúrovňovej abstrakcie s plynulým prechodom na nižšiu detailnejšiu úroveň. Systém je vyjadrený jazykmi na rôznej úrovni abstrakcie (Obr. 5), pričom každá ďalšia úroveň obsahuje viac implementačných detailov. Jednotlivé úrovne zodpovedajú explicitne špecifikovaným počítačovým jazykom.



Obr. 4: Dva pohľady na modifikáciu systému



Obr. 5: Pyramída jazykov

2.7. Programy ako zdroje znalostí

Zaoberať sa jazykmi má kľúčový význam aj v kontexte znalostí spoločnosti. V súčasnosti existuje množstvo prevádzkovaných softvérových systémov, ktoré sú zapísané vo forme artefaktov v počítačových jazykov. Na rozdiel od jazykov prirodzených sú počítačové jazyky formálne a teda artefakty, prostredníctvom ktorých sú softvérové systémy realizované, tvoria pravdepodobne najväčšiu formálnu, umelo vybudovanú bázu znalostí ľudstva. Vytvorené softvérové systémy teda nie sú len zdrojom údajov ale obsahujú aj postupy ako informácie spracovať a použiť. Napríklad bankový systém neobsahuje len údaje o klientoch, ich účtoch a stavoch na účtoch ale zároveň definuje podmienky za akých je možné vykonať prevod medzi účtami.

Jedným zo základných problémov použitia týchto znalostí je zložitosť automatizovanej extrakcie z dôvodu často vysokej miery implementačných a technologických detailov. Kľúčová je pri tom úloha identikátorov, ktoré predstavujú mená označujúce pojmy. Hlavnou úlohou identifikátorov z pohľadu jazykových procesorov je odkazovanie na konkrétny výskyt pojmu, zatiaľ čo z pohľadu programátorov je kľúčové aj meno, ktoré by malo označovať význam výskytu daného pojmu. Identifikátory sú však volené programátorom všeobecne zabezpečujúc vysokú mieru abstrakcie, čo môže znamenať rovnakú zložitosť pri automatizovanom spracovaní ako spracovanie textov v prirodzených jazykoch. Mená identifikátorov sú pre vykonávanie zvyčajne bezpredmetné napriek ich výraznému významu pri pochopení fungovania systému

Nasledujúci príklad je ukážkou dvoch fragmetov zdrojových kódov, ktorých syntaktická štruktúra je rovnaká, pozmenené sú len identifikátory. Ako prvý je uvedený fragment zdrojového kódu so zvolenými identifikátormi, ktorý neumožňuje v podstate ani určiť doménu použitia. Na druhej strane však môže byť tento kód použitý aj vo viacerých doménach, čo môže byť nepochopené čitateľom tohto kódu.

```
public void spr(BObject b1, BObject b2, double k) {
     if (k > 0 && b1.esm() > k) {
         b1.hbe(k);
         b2.ppo(k);
     } else {
         //...
     }
}
```

Druhý fragment zdrojového kódu používa identifikátory z domény, ktoré napovedajú, že sa jedná o doménu bankovníctva.

```
public void transfer(Account from, Account to, double ammount) {
    if (ammount > 0 && from.getDisposableBalance() > ammount) {
        from.withdraw(ammount);
        to.deposit(ammount);
    } else {
        //...
    }
}
```

Z týchto dvoch fragmetov kódu je možné odhaliť kľúčovú úlohu pomenovaní v programoch. V prípade použitia doménovo-špecifických jazykov sú pojmy domény definované konštrukciami jazyka a v týchto jazykoch identifikátory nezohrávajú takú kľúčovú úlohu.

2.8. Kompozícia a evolúcia jazykov

V súčasnosti je zrejmý rastúci dopyt po softvérových systémoch, ktoré sú jednoducho prispôsobiteľné konkrétnym požiadavkám a prostrediu, v ktorom budú používané, respektíve dokonca automaticky adaptované na požadované zmeny počas používania. Rýchle prispôsobenie sa permanentným zmenám požiadaviek na softvérový systém v krátkej dobe sa stáva konkurenčnou výhodou a prináša možnosti rýchleho nasadenia na trhu [126]. Zmeny v softvérovom systéme tak zastávajú významné miesto v životnom cykle softvéru. Niektoré štúdie uvádzajú [23][25][95][126], že náklady spojené s údržbou a zmenami v softvérovom systéme sú v rozsahu 50 až 90 percent celkových nákladov na softvér.

Evolúcia softvérového systému zahŕňa všetky aktivity spojené s vytvorením novej verzie systému z už existujúceho funkčného systému. Evolúcia systémov je dnes taká bežná, že vývoj systémov od úplného začiatku je skôr výnimkou [126].

Efektívna možnosť realizácie funkčných zmien je jednou z požiadaviek na softvérový systém. Softvérový systém by nemal byť brzdou evolúcie prostredia, v ktorom je používaný. Naopak, mal by plynulo umožňovať realizáciu zmien spôsobených novými požiadavkami, ktoré prichádzajú od používateľov systému [48]. Nie je dnes neobvyklé, že požiadavky na systém (požiadavky na funkčnosť ale aj požiadavky nepriamo súvisiace s funkčnosťou), sú špecifikované postupne a samotný systém je vytváraný postupným rozširovaním [74], kopírujúc evolúciu prostredia, v ktorom je softvérový systém používaný. Softvérový systém nereagujúci na evolúciu prostredia, v ktorom je používaný, postupne zastaráva [92].

Výskum evolúcie softvérových systémov je zameraný hlavne na nájdenie vhodných abstrakcií, formálnych modelov a metód, nástrojov a postupov pre podporu realizácie evolúcie softvérových systémov s cieľom zvýšenia produktivity, spoľahlivosti, zvýšenia kvality a zníženia ceny za zmeny vynútené evolúciou [69]. Publikácie [78][79]

charakterizujú hlavné výzvy a otázky vo výskume evolúcie softvérových systémov. Kľúčová je orientácia na oblasť návrhu formálneho modelu evolúcie [68] softvérových jazykov, vytvorenia mechanizmov na podporu súčasnej evolúcie artefaktov (koevolúcia) a podporu evolúcie artefaktov definovaných v rôznych softvérových jazykoch [P4][P5][P9][P10].

Používanie doménovo-špecifických jazykov pri modelmi riadenom vývoji softvéru [39][104] síce zjednodušuje realizáciu evolúcie v softvérovom systéme, na druhej strane však prináša požiadavku na definovanie kompozície doménovo-špecifických jazykov a mechanizmov ich evolúcie.

Pre zabezpečenie efektívnej evolúcie softvérových systémov je preto podstatná podpora evolúcie v konkrétnom softvérovom jazyku ako aj evolúcia softvérových jazykov [P16][P17][P29] použitých na vyjadrenie artefaktov systému, prostredníctvom ktorých je systém vytváraný. Jeden z dôvodov často pomalej evolúcie softvérových jazykov je vysoká cena za transformáciu artefaktov vyjadrených v jazyku do novej verzie jazyka. Táto situácia je ešte zložitejšia ak je systém vytvorený kompozíciou artefaktov definovaných v rôznych jazykoch, pričom tieto jazyky nepodliehajú evolučným zmenám rovnakou rýchlosťou [P25]. Práve z tohto dôvodu je jedným z hlavných cieľov výskumu hľadanie modelov a mechanizmov pre definovanie evolúcie jazykov v kompozícii ako aj následné vytvorenie mechanizmov pre automatizovanú transformáciu artefaktov na základe zmien vyvolaných evolúciou jazyka.

Aj keď sa môže zdať, že počítačové jazyky patria k statickým entitám, ktoré podliehajú zmenám minimálne realita je mnohokrát iná. Dynamika zmien jazyka je zvyčajne daná doménou, pre ktorú je jazyk vytvorený. Zmeny v doméne ale nie sú jediným dôvodom pre zmeny jazyka opisujúceho danú domému. Do výraznej miery je jazyk závislý aj od skupiny, pre ktorú je určený ako aj od skupiny autorov jazyka. Dôvodmi evolúcie jazyka teda sú:

- zmeny v doméne – spôsobené prirodzenou evolúciou domény,
- zmeny spôsobené autormi jazyka – spôsobené autormi jazyka napríklad z dôvodu optimalizácie jazykových konštrukcií alebo zmenou interpretácie doménových pojmov,
- zmeny iniciované používateľmi jazyka – spôsobené napríklad zmenou špecifikácie domény.

Príkladom rôznej dynamiky počítačových jazykov môže byť aj kontrast jazykov všeobecného použitia oproti doménovo-špecifickým jazykom. Doménovo-špecifické jazyky podliehajú zmenám zvyčajne častejšie pretože priamo súvisia s konkrétnou doménou. Príkladom formálneho opisu evolučných zmien jazyka je napríklad mapovanie významových množín v slovníku WordNet z verzie 2.1 na verziu 3.0 [27].

3 Anatómia jazyka

Táto kapitola sa venuje opisu anatómie formálnych jazykov. Pochopenie anatómie jazyka je kľúčové pre výskum v oblasti návrhu počítačových jazykov ako aj v oblasti implementácie jazykových procesorov. Pri skúmaní formálnych jazykov sa zvyčajne jazyk opisuje troma zložkami [39]:

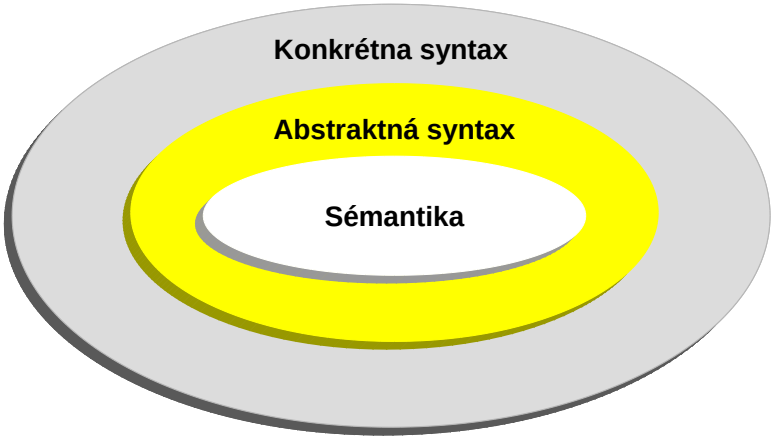
- abstraktná syntax,
- konkrétna syntax,
- sémantika.

Abstraktná syntax jazyka charakterizuje v abstraktnej forme elementy (pojmy), ktoré vytvárajú jazyk a definuje pravidlá pre kompozíciu týchto elementov.

Konkrétna syntax jazyka určuje ako sú jednotlivé jazykové elementy reprezentované v konkrétnej podobe, ktorú môže interpretovať človek aj počítač. Existuje viacero foriem pre konkrétnu reprezentáciu jazyka. Medzi tieto formy patrí napríklad textová forma, vizuálna forma, respektíve kombinácia týchto foriem. Niekedy sa pre konkrétnu syntax používa aj označenie *zápis*.

Sémantika jazyka definuje *význam* viet z daného jazyka zvyčajne prostredníctvom sémantiky elementov jazyka a pravidiel kompozície sémantiky pre tieto elementy.

Anatómia počítačových jazykov z pohľadu jednotlivých zložiek je zobrazená na Obr. 6. Sémantika počítačového jazyka je definovaná vo väzbe na abstraktnú syntax jazyka. Konkrétna syntax opisuje zápis viet jazyka a je spätá s abstraktnou syntaxou.



Obr. 6: Anatómia počítačového jazyka [39]

3.1. Abstraktná syntax jazyka

Abstraktná syntax jazyka opisuje vnútornú štruktúru jazyka a používa sa pre definovanie sémantiky jazyka a programov [91]. Definícia abstraktnej syntaxe pozostáva:

- z množiny syntaktických oblastí,
- z množiny pravidiel pre kompozíciu syntaktických oblastí.

Syntaktická oblasť je množina syntaktických položiek, ktoré majú spoločnú syntaktickú štruktúru. Syntaktická oblasť definuje jazykový pojem. Pravidlá pre kompozíciu syntaktických oblastí definujú spôsoby vytvárania zložených prvkov syntaktickej oblasti.

Najbežnejším spôsobom vyjadrenia abstraktnej syntaxe (ale aj konkrétnej syntaxe) sú *bezkontextové gramatiky* zapísané Backus-Naurovou formou (BNF). Ďalším spôsobom

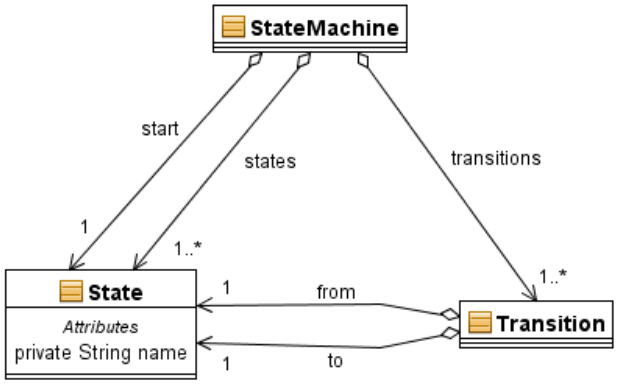
opisu abstraktnej syntaxe jazyka je použitie *metamodelov*, ktoré majú významné postavenie hlavne v modelom riadenom vývoji softvéru [39][104]. Metamodelovanie získalo popularitu pre definíciu abstraktnej syntaxe jazyka vďaka použitiu tohto prístupu na definíciu abstraktnej syntaxe jazyka UML [99]. UML je vyjadrené v podmnožine UML. Metamodel charakterizuje jazykové elementy ako triedy a relácie medzi triedami použitím asociácií a atribútov.

Ako príklad pre definíciu abstraktnej syntaxe jazyka bude uvedený jazyk stavových automatov. V tomto jazyku sú základnými pojmami stavový automat (`StateMachine`), stav (`State`) a prechod (`Transition`). Stavový automat v definovanom jazyku má práve jeden počiatočný stav, neprázdnu množinu stavov a neprázdnu množinu prechodov. Jednotlivé stavy sú označené menom (`Identifier`). Prechod má definovaný počiatočný a konečný stav.

Abstraktnú syntax jazyka stavových automatov je možné definovať prostredníctvom bezkontextových gramatík nasledovne.

```
StateMachine ::= StartState State+ Transition+
State ::= Identifier
StartState ::= Identifier
Transition ::= Identifier Identifier
```

Abstraktnú syntax jazyka je možné definovať aj prostredníctvom diagramu tried, ktorý reprezentuje štruktúru jazyka. Diagramy tried je teda možné použiť na vyjadrenie abstraktnej štruktúry jazykov a teda diagram tried je možné považovať za metamodel. Abstraktná syntax jazyka stavových automatov je zobrazená na Obr. 7.

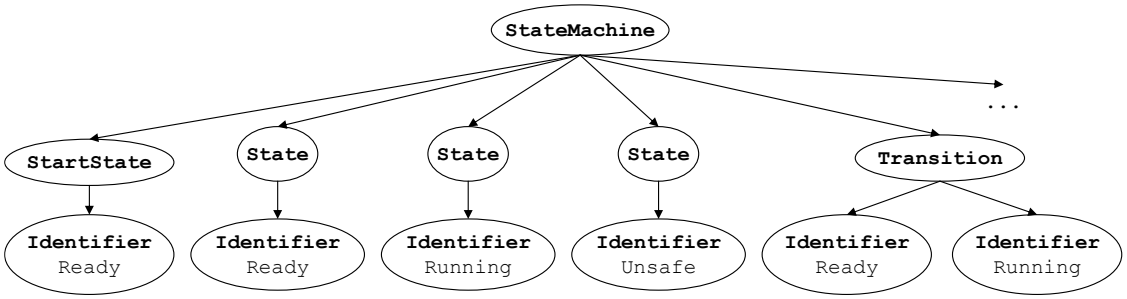


Obr. 7: Abstraktná syntax jazyka stavových automatov

Metamodel definuje množinu abstraktných syntaktických grafov, zatiaľ čo bezkontextové gramatiky definujú množinu stromov, prostredníctvom ktorých je možné reprezentovať vety z jazyka. Metamodely teda poskytujú prirodzenejšiu formu reprezentácie referencií. V bezkontextových gramatikách neexistuje na vyjadrenie referencie špeciálna konštrukcia. Na vytváranie odkazov sú používané textové identifikátory. Táto rozdielnosť je daná aj charakterom formy konkrétnej reprezentácie. Metamodely sú zvyčajne používané na definovanie jazykov s vizuálnou konkrétnou syntaxou, zatiaľ čo bezkontextové gramatiky sú používané na definovanie konkrétnej syntaxe textových jazykov. Táto práca je zároveň poukázaním na možnosť použitia metamodelov na reprezentáciu textových jazykov (časť 5).

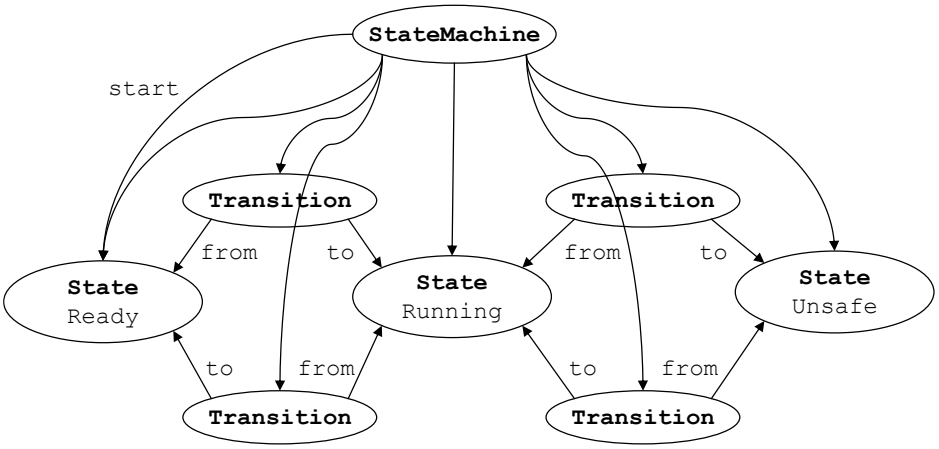
3.1.1. Abstraktný syntaktický strom

Abstraktný syntaktický strom predstavuje stromovú reprezentáciu abstraktnej štruktúry vety z počítačového jazyka. Každý uzol abstraktného syntaktického stromu určuje výskyt jazykového pojmu vo vete. Príklad abstraktného syntaktického stromu pre vetu z jazyka stavových automatov je uvedený na Obr. 8.



Obr. 8: Abstraktný syntaktický strom

Abstraktný syntaktický graf vytvorený na základe definovaného modelu pre jazyk stavových automatov je zobrazený na Obr. 9. Na rozdiel od abstraktného syntaktického stromu, uvedený graf obsahuje aj hrany reprezentujúce referencie, ktoré boli v strome reprezentované textovými identifikátormi. Týmito hranami sú hrany označujúce počiatočný stav a hrany pre prechody zo stavu do stavu.

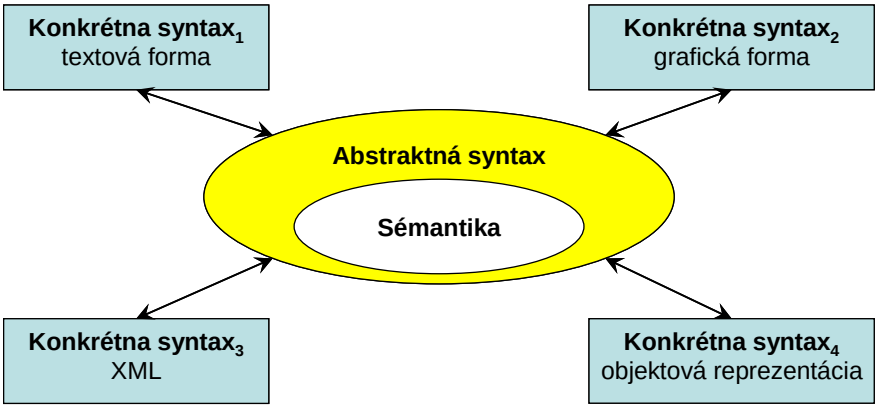


Obr. 9: Abstraktný syntaktický graf

3.2. Konkrétna syntax jazyka

Konkrétna syntax definuje jednoznačný spôsob reprezentácie viet počítačového jazyka. Vety môžu byť reprezentované ako postupnosť symbolov (textová forma) alebo aj vizuálne prostredníctvom geometrických útvarov (grafická forma). Jeden počítačový jazyk s definovanou abstraktnou syntaxou a sémantikou môže mať viacero konkrétnych foriem reprezentácie. Používateľ počítačového jazyka si môže v takomto prípade zvoliť obľúbenú formu reprezentácie viet jazyka, respektíve formy podľa potreby kombinovať (Obr. 10).

Konkrétna syntax textových jazykov sa zvyčajne špecifikuje prostredníctvom bezkontextových gramatík. Pre špecifikáciu grafickej formy konkrétnej syntaxe jazykov sa používajú prostredia pre tvorbu jazykov.



Obr. 10: Konkrétna syntax - rôzne reprezentácie viet jazyka

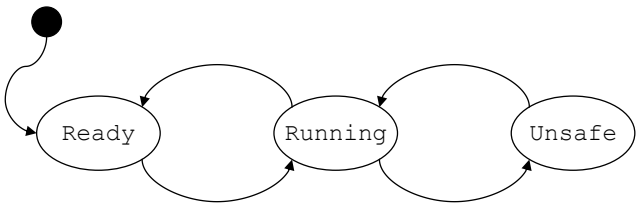
Pre uvedený jazyk stavových automatov je možné definovať konkrétnu syntax napríklad nasledovne.

```
StateMachine ::= StartState State+ Transition+
StartState ::= 'start' Identifier
State ::= 'state' Identifier
Transition ::= 'transition' 'from' Identifier 'to' Identifier
Identifier::= ['a'-'z''A'-'Z']+
```

Nasledujúca veta je príkladom vety z jazyka stavových automatov podľa uvedenej konkrétnej syntaxe v textovej forme.

```
start Ready
state Ready
state Running
state Unsafe
transition from Ready to Running
transition from Running to Ready
transition from Running to Unsafe
transition from Unsafe to Running
```

Uvedenú vetu je možné reprezentovať aj v grafickej forme, napríklad spôsobom uvedeným na Obr. 11.



Obr. 11: Grafické vyjadrenie vety jazyka

3.2.1. Syntax pre serializáciu

V [39] je definovaný jeden špeciálny typ konkrétnej syntaxe označený ako *syntax pre serializáciu*. Táto konkrétna syntax je určená pre výmenu viet z počítačového jazyka medzi nástrojmi pre vizuálne jazyky. Serializácia slúži na transformáciu abstraktných syntaktických grafov do podoby postupnosti znakov, resp. bajtov. Najbežnejšie sa používa na takúto serializáciu XML formát uloženia údajov [26]. Pre jazyky s textovou formou konkrétnej reprezentácie nie je takáto syntax podstatná, pretože každú vetu je možné zapísať vo forme postupnosti znakov.

Nasledujúci príklad je ukážkou zápisu vety jazyka stavových automatov vo forme XML dokumentu. Tento zápis vyjadruje v podstate abstraktný syntaktický strom vety z jazyka.

```
<machine start="Ready">
      <state name="Ready" />
      <state name="Running" />
      <state name="Unsafe" />

      <transition from="Ready" to="Running" />
      <transition from="Running" to="Ready" />
      <transition from="Running" to="Unsafe" />
      <transition from="Unsafe" to="Running" />
</machine>
```

Pre potreby uloženia konkrétnej syntaxe grafických jazykov je možné vytvoriť dokument definujúci rozmiestnenie a použitie geometrických útvarov reprezentujúcich jednotlivé jazykové elementy a ich väzby. Takéto oddelenie je vhodnou separáciou pre uloženie abstraktnej a konkrétnej syntaxe umožňujúce jednoduchšie spracovanie viet jazyka a určenie sémantiky vety nezávisle od konkrétnej formy zápisu.

Okrem XML formy sa dnes často používa na zápis viet formát JSON, ktoré popularita je spojená s webovými aplikáciami a jazykom Javascript. Vo formáte JSON by veta z jazyka stavovaných automatov mohla vyzerať nasledovne.

```
{
      "start": "Ready",
      "states": ["Ready", "Running", "Unsafe"],
      "transitions": [
            {"from": "Ready", "to": "Running"},
            {"from": "Running", "to": "Ready"},
            {"from": "Running", "to": "Unsafe"},
            {"from": "Unsafe", "to": "Running"}
      ]
}
```

Ďalšou alternatívou k formátom XML a JSON je formát YAML, ktorý sa snaží priniesť ešte úspornejšiu formu (kratší text) vynechaním zátvoriek. Veta v tomto formáte by mohla mohla vyzerať nasledovne.

```
start: "Ready"
states:
      - "Ready"
      - "Running"
      - "Unsafe"
transitions:
      -     from: "Ready"
            to: "Running"
      -     from: "Running"
            to: "Ready"
      -     from: "Running"
            to: "Unsafe"
      -     from: "Unsafe"
            to: "Running"
```

3.3. Sémantika jazyka

Formálna sémantika definuje jednoznačný význam programov vo výrazoch jazyka matematiky. Definícia formálnej sémantiky jazyka má svoj význam pri [91]:

- *návrhu počítačového jazyka* – presná a jednoznačná dokumentácia v matematických pojmoch,
- *implementácii počítačového jazyka* – umožňuje generovať jazykový prekladač zo sémantickej definície,
- *používaní počítačového jazyka* – pochopenie jazykových konštrukcií.

Sémantika definuje význam viet (programov) počítačových jazykov. Medzi najvýznamnejšie prístupy definovania formálnej sémantiky počítačových jazykov patria [88]:

- operačná sémantika,
- denotačná sémantika a
- axiomatická sémantika.

Operačná sémantika definuje význam programu pomocou činností, ktoré prebehnú pri vykonaní v počítači. Zaoberá sa tým ako vzniká výsledok programu. Opis operačnej sémantiky je možné nájsť v [88][97].

Denotačná sémantika opisuje význam programov pomocou prvkov množín, ktoré sú definované formulami jazyka teórie množín. Tieto množiny sa nazývajú množinami významu, alebo tiež sémantickými oblasťami. Denotačná sémantika je charakterizovaná v [2][88][106].

Axiomatická sémantika slúži na dokazovanie čiastočnej korektnosti programov. Axiomatická sémantika je založená na matematickej logike. Axiomatická sémantika je opísaná v [88].

4 Návrh a implementácia jazykov

Pri implementácii doménovo-špecifických jazykov je možné identifikovať tri základné prístupy:

- externé doménovo-špecifické jazyky,
- interné doménovo-špecifické jazyky,
- použitie jazykových prostredí.

4.1. Externé jazyky

Externé doménovo-špecifické jazyky nie sú závislé od iných jazykov. Pre externé doménovo-špecifické jazyky vytvára autor jazyka jazykový procesor, ktorý je určený na spracovanie viet tohto jazyka so znalosťou konkrétnej, abstraktnej syntaxe a sémantiky vytváraného doménovo-špecifického jazyka. Výhodou je úplná možnosť určenia syntaxe bez vplyvu ostatných jazykov a spracovanie procesorom, ktorý zvyčajne umožňuje nielen syntaktickú ale aj sémantickú kontrolu. Na druhej strane ale autor takéhoto jazyka a procesora musí poznať technológiu spracovania textových dokumentov. Najčastejšie sa pre implementáciu takýchto jazykových procesorov používajú generátory jazykových procesorov [94]. Príkladom externého jazyka je jazyk stavových automatov uvedený v časti 3.2.

4.1.1. Generátory jazykových procesorov

Generátor jazykových procesorov (angl. parser generator) je nástroj, ktorý generuje jazykový procesor na základe formálnej špecifikácie tohto procesora. Vstupom generátora je zvyčajne bezkontextová gramatika zapísaná v požadovanom formáte a výstupom je zdrojový kód jazykového procesora. Pred použitím konkrétneho generátora jazykových procesorov je nutné naštudovať konkrétny jazyk generátora pre špecifikáciu jazykového procesora. Okrem určenia konkrétnej syntaxe umožňuje generátor jazykových procesorov určovať aj sémantické akcie. Niektoré generátory jazykových procesorov umožňujú automatické generovanie abstraktných syntaktických stromov na základe špecifikácie konkrétnej syntaxe jazyka. Väčšina generátorov je však orientovaná na konkrétnu syntax.

Medzi najznámejšie a najpoužívanejšie generátory jazykových procesorov patrí Lex/Yacc, Flex/Bison, JavaCC, ANTLR, SableCC, Gold. Porovnanie generátorov jazykových procesorov je možné nájsť v [111]. Jednotlivé generátory sa odlišujú technológiou rozpoznania viet (LR, LL, LALR, GLR) ako aj cieľovým jazykom, pre ktorý je generovaný jazykový procesor. Napriek tomu, že v ideálnom prípade by mal generátor akceptovať ľubovoľnú gramatiku, existujú dva dôvody prečo takéto generátory nie sú zvyčajne implementované [94]. Prvým je nižšia efektivita univerzálnych metód rozpoznania viet. Druhým je, že nie pre každú bezkontextovú gramatiku je možné skonštruovať deterministický zásobníkový automat. Aj keď výskum v oblasti generátorov jazykových procesorov siaha desaťročia do minulosti, je viac orientovaný na rozpoznanie viet jazyka s textovou konkrétnou syntaxou ako na komplexný návrh počítačových jazykov postihujúc abstraktnú syntax a sémantiku.

4.2. Interné jazyky

Ďalšou možnosťou implementácie doménovo-špecifických jazykov je vytváranie interných (vložených) jazykov. Na rozdiel od externých jazykov nie je nutné pre takéto

jazyky vytvárať špeciálny jazykový procesor. Pre implementáciu interného doménovo-špecifického jazyka sa používa zvolený jazyk všeobecného použitia, ktorý sa nazýva hostiteľský jazyk. Autor takéhoto jazyka nemusí mať skúsenosti s bezkontextovými gramatikami, ani generátormi jazykových procesorov, čo môže znamenať zjednodušenia implementácie. Na druhej strane je však syntax interného jazyka ovplyvnená hostiteľským jazykom, v ktorom je implementovaný doménovo-špecifický jazyk. Implementácia interných doménovo-špecifických jazykov je charakterizovaná v [45][82]. [33] definuje vzory implementácie interných jazykov a na príkladoch aj prezentuje jednotlivé identifikované vzory.

4.2.1. Implementácia v objektových jazykoch

Prvý príklad je ukážkou implementácie interného doménovo-špecifického jazyka v objektovo-orientovanom jazyku Java podľa vzoru „Object Scoping“ [33]. Pri použití tohto vzoru sú pojmy doménovo-špecifického jazyka definované prostredníctvom metód v rodičovskej triede. Veta jazyka je zapísaná v triede, ktorá od tejto rodičovskej triedy dedí. Uvedený príklad je implementáciou jazyka stavových automatov. Najprv je uvedená rodičovská trieda definujúca pojmy jazyka (detaily implementácie sú vynechané).

```
public abstract class AbstractStateMachine {
    public AbstractStateMachine() {
        define();
        validate();
    }

    protected void start(String name) { ... }

    protected void state(String name) { ... }

    protected void transition(String from, String to) { ... }

    protected abstract void define();

    private void validate() { ... }
}
```

Nasledujúci fragment predstavuje vetu z doménovo-špecifického jazyka, ktorá definuje stavový automat.

```
public class TestMachine extends AbstractStateMachine {
    protected void define() {
        state("Ready");
        state("Running");
        state("Unsafe");

        transition("Ready ", "Running");
        transition("Running", "Ready ");
        transition("Running", "Unsafe");
        transition("Unsafe", "Running");

        start("Ready");
    }
}
```

Implementácia doménovo-špecifického jazyka prostredníctvom hostiteľského jazyka všeobecného použitia zároveň neumožňuje úplne obmedziť množinu konštrukcií v doménovo-špecifických jazykoch, čo môže byť nežiadúce pri tvorbe viet.

4.2.2. Implementácia vo funkcionálnych jazykoch

V tejto časti je uvedený príklad implementácie interného doménovo-špecifického jazyka vo funkcionálnom programovacom jazyku Haskell prostredníctvom algebraických typov a modulov. Pojmy jazyka stavových automatov je možné definovať nasledovne.

```
module StateMachine (
  define, MachineType (Machine), StateType (State), TransitionType
(Transition)
) where

data MachineType = Machine String [StateType] [TransitionType]
data StateType = State String
data TransitionType = Transition String String

define :: MachineType -> MachineType
define m | validate m = m

validate :: MachineType -> Bool
validate (Machine start states transitions) =
      elem (State start) states  && (all (\(Transition s1 s2) -> (elem
(State s1) states) && (elem (State s2) states)) transitions)
```

Konkrétny stavový automat je možné zapísať použitím definovaného interného jazyka nasledovne.

```
import StateMachine

testMachine = define (Machine "Ready"
      [State "Ready", State "Running", State "Unsafe"]
      [Transition "Ready" "Running", Transition "Running" "Ready",
      Transition "Running" "Unsafe", Transition "Unsafe" "Running"])
```

4.2.3. XML, YAML a JSON jazyky

XML (Extensible Markup Language) umožňuje definovanie textovo-orientovaných jazykov pomocou označovania údajov značkami. XML umožňuje definovať doménovo-špecifické jazyky. Na rozdiel od dokumentov zapísaných v binárnych jazykoch je takýto formát jednoduchšie čitateľný človekom ale na druhej strane si zachováva možnosť automatického spracovania výpočtovou technikou. XML teda nie je závislé od počítačového ani softvérového prostredia. XML je definované konzorciom W3C ako odporúčanie [26]. XML nie je jazyk ale je to odporúčanie na tvorbu jazykov, ktoré spĺňajú XML kritériá. XML teda opisuje syntax ako vytvárať vlastné jazyky. XML nemá aplikačne špecifické použitie ale je ho možné použiť podľa potreby v rôznych aplikačných doménach. Výhodou XML technológie je existencia jazykových procesorov rôzneho charakteru a pre rôzne platformy, ktoré umožňujú pomerne jednoduché spracovávanie XML dokumentov. Nevýhodou je prílišná nadbytočnosť značiek, čo spôsobuje nižšiu efektivitu nielen pri priamom vytváraní XML dokumentov človekom ale aj pri ich čítaní. Príkladom je nasledujúci zápis aritmetického výrazu, ktorý je vyjadrený aj v štruktúrovanom XML dokumente.

`1 + 2 * 7`	`<add>`
	`  <number>1</number>`
	`  <mul>`
	`  <number>2</number>`

```
    <number>7</number>
  </mul>
</add>
```

Napriek tomu zohrávajú XML jazyky významnú úlohu v jazykovom inžinierstve a sú vhodné najmä pre komunikáciu medzi heterogénnymi počítačovými systémami.

Ďalšími jazykmi, ktoré sú podobne použiteľné ako jazyk XML na definovanie vlastných počítačových jazykov sú jazyky YAML a JSON. Samozrejme, aj v týchto prípadoch ale musíme prispôsobiť konkrétnu syntax pravidlám a obmedzeniam týchto jazykov.

4.2.4. Atribútovo-orientované programovanie

Atribútovo-orientované programovanie [13][89][119] je technika umožňujúca označovať jazykové elementy v programe prostredníctvom štruktúrovaných poznámok (anotácií). Tieto anotácie nesú informácie, ktoré je možné interpretovať v kontexte anotovaných jazykových elementov. Atribútovo-orientované programovanie je podporované napríklad v programovacom jazyku Java aj v jazyku C#. Atribútovo-orientované programovanie je vhodné aj na implementáciu interných doménovo-špecifických jazykov.

Navrhnutá metóda špecifikácie jazykových procesorov uvedená v časti 5 spája výhody prístupov implementácie interných aj externých jazykov spolu s princípmi atribútovo-orientovaného programovania.

4.3. Jazykové prostredia

V poslednej dobe nadobúdajú prostredia pre tvorbu doménovo-špecifických jazykov a doménové modelovanie (označované angl. Language Workbench [32]) stále väčší význam. Je to spôsobené hlavne rastúcou kvalitou týchto prostredí a zjednodušovaním ich použitia. Pomocou prostredí je možné vytvárať grafické dizajnéry pre vizuálne jazyky, písať transformácie z grafických modelov do výstupného jazyka a generovať výstupné vety v cieľovom jazyku. Nasledujúce časti stručne opisujú niektoré prostredia pre tvorbu doménovo-špecifických jazykov a doménové modelovanie.

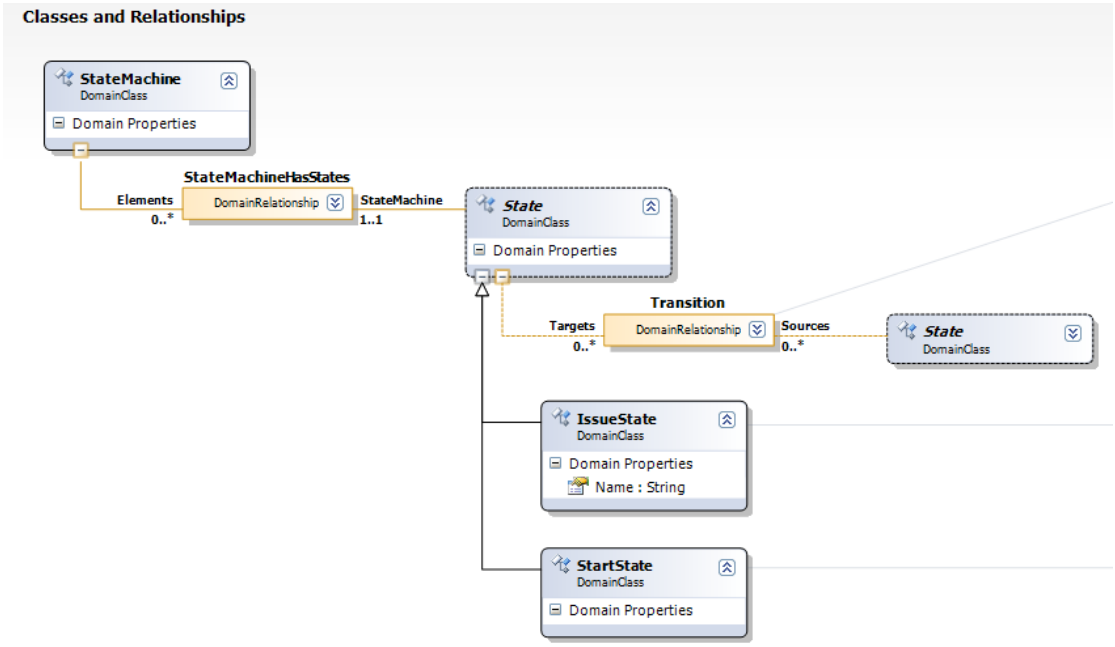
4.3.1. Microsoft Visual Studio DSL

Microsoft Visual Studio DSL [16][70] je nástrojom, ktorý umožňuje navrhovať a implementovať grafické doménovo-špecifické jazyky. Tento nástroj je súčasťou vývojového prostredia Microsoft Visual Studio. Visual Studio DSL nástroj bol vytvorený podľa softvérovo inžinierskeho prístupu, ktorý sa nazýva *softvérové továrne*. Softvérové továrne majú svoj pôvod v CASE nástrojoch a modelom riadenom vývoji softvéru. Obrazovka prezentujúca návrh jazyka v nástroji Visual Studio DSL je zobrazená na Obr. 12. Na základe špecifikácie jazyka prostredníctvom metamodelu je vygenerovaný vizuálny editor viet (Obr. 13).

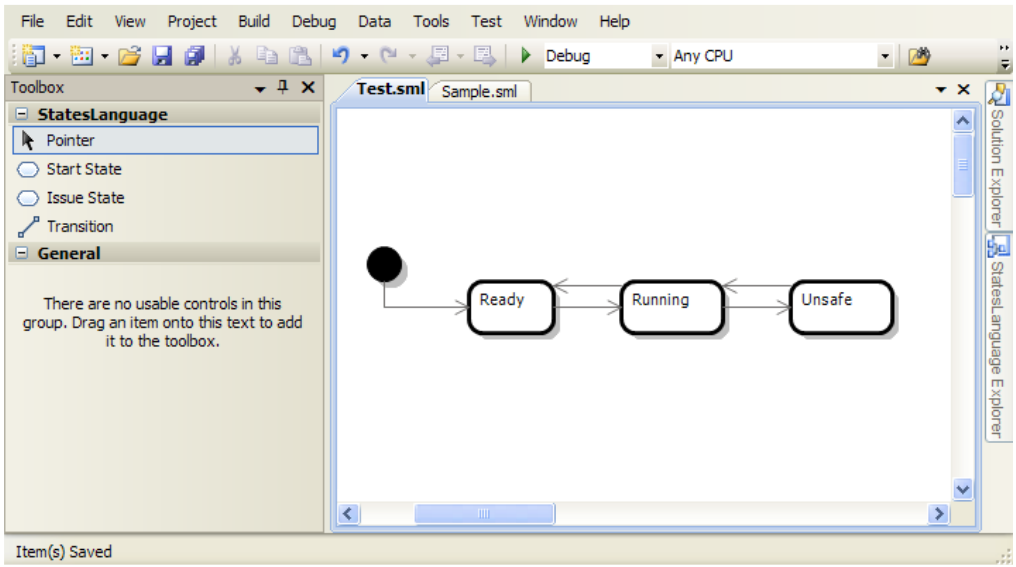
4.3.2. GME: Generic Modeling Environment

GME [65][66] je konfigurovateľné prostredie určené pre vytváranie doménovo-špecifických modelov a syntézu programov. Prostredie je možné konfigurovať prostredníctvom metamodelov umožňujúcich špecifikovať syntax a sémantiku pojmov domény a spôsoby kompozície pojmov. Metamodelovací jazyk je odvodený od diagramu

tried jazyka UML a je rozšírený o jazyk OCL. Vizualizáciu modelu je možné prispôsobiť požiadavkám používateľov jazyka prostredníctvom rozhraní. Na Obr. 14 je pre názornosť uvedená obrazovka z prostredia GME.



Obr. 12: Obrazovka z nástroja Microsoft Visual Studio DSL – návrh jazyka

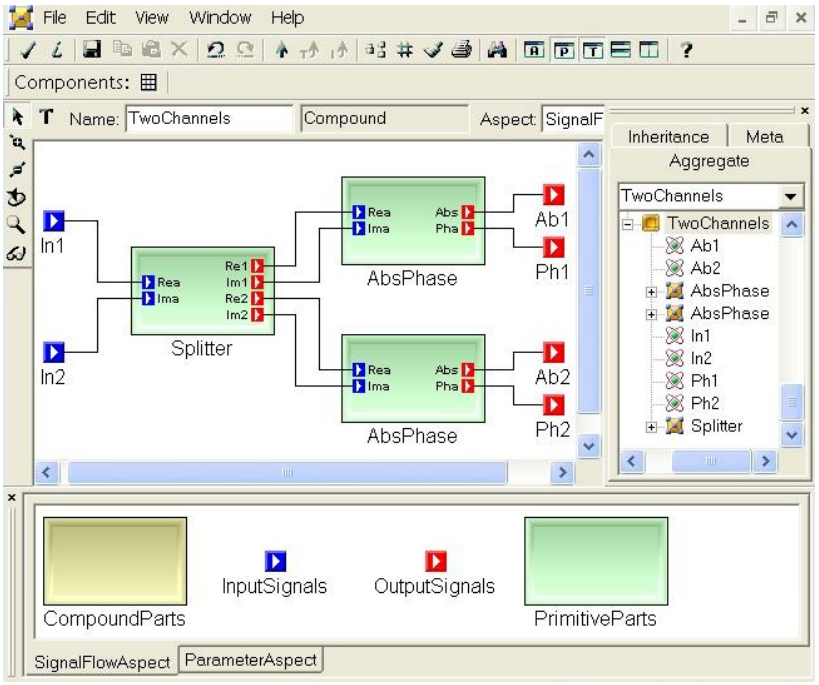


Obr. 13: Vygenerovaný vizuálny editor viet jazyka

4.3.3. Jetbrains Meta Programming System

Jetbrains Meta Programming System (MPS) je vývojové prostredie spoločnosti JetBrains [21] podporujúce vytváranie počítačových jazykov. Toto prostredie je inšpirované myšlienkou jazykovo-orientovaného programovania [123]. Cieľom vytvorenia tohto prostredia bolo uľahčenie vytvárania doménovo-špecifických jazykoch. Na základe

špecifikácie jazyka sú zároveň generované nástroje podporujúce tvorbu viet v tomto jazyku: editor s dopĺňaním kódu a refaktorizačné nástroje.



Obr. 14: Obrazovka z nástroja GME

4.3.4. MetaEdit+

MetaEdit+ [52][114] je prostredie pre vytváranie doménovo-špecifických jazykoch a ich používanie. Prostredie MetaEdit+ je výstupom výskumu na univerzite v Jyväskylä. MetaEdit+ umožňuje vytvárať nástroje pre doménovo-špecifické jazyky konfiguráciou generických nástrojov prostredníctvom metamodelov. MetaEdit+ používa vlastný metamodelovací jazyk GOPPRR [52] (Graph-Object-Property-Port-Role-Relationship). Všetky vytvorené metamodely a modely sú uložené v objektovej forme.

5 Generovanie jazykových procesorov

Táto kapitola prezentuje pôvodnú inovatívnu metódu návrhu a implementácie jazykových procesorov, ktorá je výsledkom výskumu v oblasti spracovania formálnych jazykov s textovou reprezentáciu. V tejto metóde nie je konkrétna syntax jazyka špecifikovaná priamo bezkontextovou gramatikou ale na základe doménového modelu jazyka obsahujúceho doménové triedy a ich vzájomné vzťahy. Doménový model jazyka vyjadrujúci abstraktnú syntax je rozšírený o dodatočné informácie špecifikujúce transformáciu abstraktnej syntaxe jazyka na konkrétnu syntax. Navrhnutá metóda sa sústreďuje na abstraktnú syntax jazyka, ktorá vyjadruje pojmy jazyka a je úzko spätá so sémantikou. Autor a implementátor jazyka prechádza počas vývoja jazyka a procesora od abstraktnej syntaxe ku konkrétnej syntaxi podľa požiadaviek na textové vyjadrenie jazyka.

Cieľom navrhnutej metódy nie je vytvorenie novej technológie rozpoznania viet počítačového jazyka na základe špecifikácie konkrétnej syntaxe jazyka bezkontextovou gramatikou. Hlavným prínosom je integrácia existujúcich technológií spracovania viet z jazyka do aparátu na vyššej úrovni abstrakcie zabezpečujúceho technologickú nezávislosť s orientáciou na abstraktnú syntax pri návrhu jazyka. Za základné charakteristiky navrhnutej metódy je možné považovať:

- Orientácia na abstraktnú syntax a sémantiku počítačového jazyka.
- Integrácia bezkontextových gramatík a doménových modelov (vyjadrených diagramom tried).
- Prechod od abstraktnej syntaxe ku konkrétnej syntaxi pri návrhu procesora.
- Nezávislý zápis konkrétnej syntaxe jazyka od technológie rozpoznávania vety počítačového jazyka.
- Automatická konštrukcia abstraktného syntaktického stromu zo vstupnej vety.
- Automatické vytváranie referencií v abstraktnom syntaktickom strome (transformácia na všeobecný graf).
- Charakteristika chýb pri rozpoznávaní vety v pojmoch abstraktnej syntaxe.
- Zjednodušenie implementácie jazykových procesorov použitím vyššieho stupňa abstrakcie.
- Separácia implementácií jazykových konštrukcií v procesore na základe pojmov jazyka.
- Zjednodušenie kompozície jazykov.

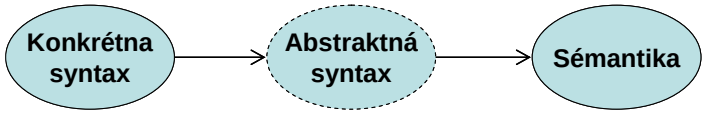
Klasické metódy implementácie jazykových procesorov používajúce generatívny prístup špecifikujú počítačový jazyk z pohľadu konkrétnej syntaxe. Tieto metódy sú silne viazané na konkrétny spôsob rozpoznania vety (zhora nadol, zdola nahor) a používajú komplexný doménovo-špecifický jazyk (napr. YACC, JavaCC) pre špecifikáciu procesora. Tento doménovo-špecifický jazyk neumožňuje zmenu spôsobu rozpoznania vety vzhľadom na jeho väzbu na konkrétny generátor jazykových procesorov. Bezkontextová gramatika opisujúca konkrétnu syntax jazyka musí spĺňať požiadavky generátora jazykových procesorov (napr. LL(1), LL(k), LR(1), LR(k)), ktorý je s typom gramatiky zviazaný.

Výhodou klasických metód špecifikácie jazyka a rozpoznania viet prostredníctvom generátorov procesorov je úplná kontrola nad zdrojmi počas spracovania vety. Úplné riadenie používania zdrojov umožňuje minimalizovať používanie zdrojov, čo bolo v čase

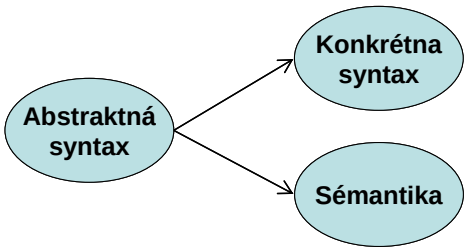
vzniku týchto nástrojov požadované (70-te roky). Touto minimalizáciou používania zdrojov sú postihnuté aj počítačové jazyky na úkor komfortu používania. Na druhej strane ale takéto riadenie používania zdrojov nie je vždy žiadúce, najmä keď je požiadavkou vytvorenie abstraktného syntaktického stromu a zvyčajne neprináša komfort pri implementácii jazykového procesora.

Klasické metódy sú orientované na konkrétnu syntax jazyka, ktorá je kľúčová pri týchto metódach návrhu jazyka. Abstraktná syntax v týchto metódach zvyčajne nie je explicitne spomínaná a je na autorovi jazykového procesora ako bude abstraktná syntax reprezentovaná.

Navrhovaná metóda je orientovaná na abstraktnú syntax jazyka, pripúšťajúc existenciu viacerých konkrétnych reprezentácií jazyka (rôzne textové a grafické formy) ako aj viaceré spôsoby definície sémantiky jazykových konštrukcií. Pri návrhu jazyka a implementácii jazykového procesora v tejto metóde je kľúčová abstraktná syntax, ktorej definíciou sa návrh jazyka začína. Konkrétna syntax a sémantika je definovaná na základe abstraktnej syntaxe jazyka. Porovnanie klasických metód a navrhnutej metódy ku konštrukcii počítačových jazykov je zobrazené na Obr. 15.



Klasický postup pri implementácii jazykového procesora

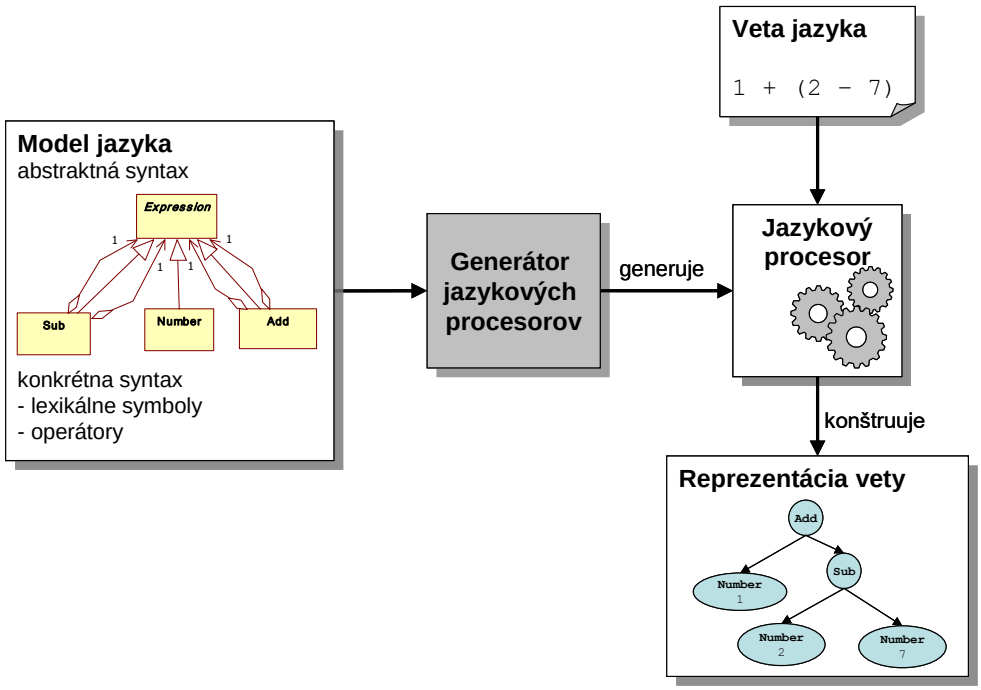


Navrhnutý postup pri implementácii jazykového procesora

Obr. 15: Porovnanie klasické a navrhnutého prístupu k tvorbe jazykových procesorov

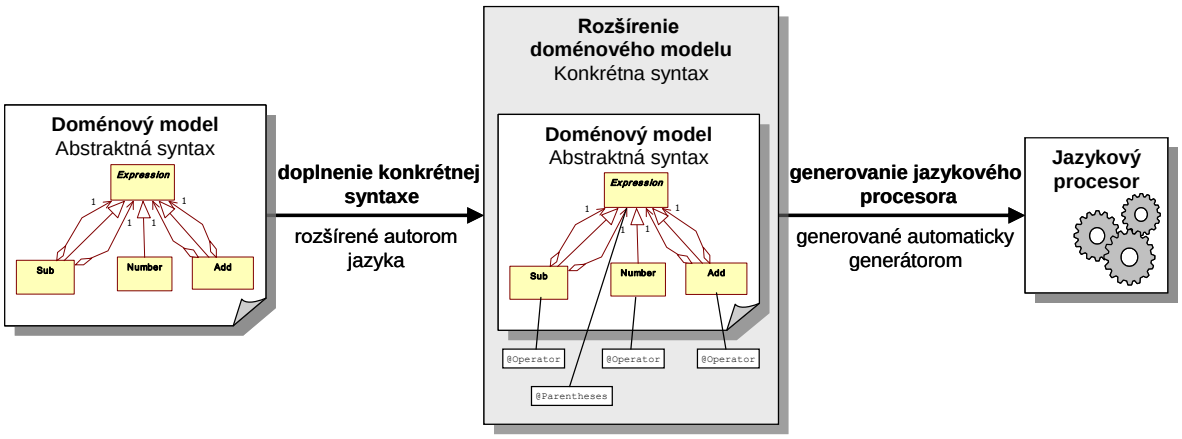
Navrhnutá metóda nie je viazaná na konkrétnu technológiu rozpoznania vety jazyka ani na konkrétny implementačný programovací jazyk pre jazykový procesor. Pre overenie použiteľnosti metódy bol implementovaný experimentálny generátor jazykových procesorov *YAJCo* (Yet Another Java Compiler Compiler) a následne bol generátor použitý pre implementáciu jazykových procesorov niekoľkých počítačových jazykov rôzneho charakteru.

Vstupom vytvoreného generátora jazykových procesorov je doménový model, ktorý je definovaný vo forme doménových tried a ich väzieb. Tento model definujúci abstraktnú syntax je rozšírený o špecifikáciu konkrétnej syntaxe počítačového jazyka prostredníctvom metaúdajov. Na základe vstupu vytvára generátor jazykových procesorov špecifikovaný jazykový procesor. Vygenerovaný jazykový procesor očakáva na vstupe vetu z jazyka a na výstupe produkuje abstraktný syntaktický strom, resp. abstraktný syntaktický graf. Uzlami abstraktného syntaktického stromu sú objekty, ktoré sú inštanciami tried doménového modelu. Konceptuálna schéma použitia generátora jazykových procesorov je zobrazená na Obr. 16.



Obr. 16: Generátor jazykových procesorov

Pri návrhu jazyka a implementácii jazykového procesora v navrhnutej metóde je abstraktná syntax jazyka definovaná doménovým modelom. Doménový model vzniká v úvode tvorby jazyka. Po vytvorení modelu je možné určiť konkrétnu syntax vo forme rozšírenia doménového modelu o anotácie. Na základe anotovaného doménového modelu je vygenerovaný jazykový procesor. Postup definovania počítačového jazyka a implementácie jazykového procesora je zobrazený na Obr. 17.



Obr. 17: Postup implementácie jazykového procesora

Generátor jazykových procesorov *YAJCo* (Yet Another Java Compiler Compiler) je implementovaný ako anotačný procesor v jazyku Java [108]. Pre potreby definovania konkrétnej syntaxe sú použité anotácie, ktoré opisujú lexikálne jednotky a vzťah medzi konkrétnou a abstraktnou syntaxou.

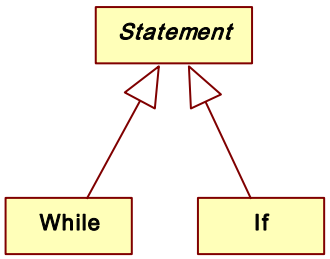
V nasledujúcich častiach sú opísané východiská navrhovanej metódy, doménový model v kontexte definície abstraktnej syntaxe, návrh generátora jazykových procesorov a charakteristika implementovaného nástroja *YAJCo*.

5.1. Doménový model

Doménový model predstavuje abstrakciu domény, pre ktorú je vytvorený. Doménový model definuje pojmy v doméne, ich vzťahy a význam. Doménový model môže byť vyjadrený rôznymi spôsobmi. Konkrétnu doménu je možné v počítačových systémoch reprezentovať jazykom, ktorý definuje abstraktnú štruktúru domény, sémantiku jednotlivých pojmov ako aj konkrétnu reprezentáciu pojmov. Táto práca sa sústreďuje na definíciu abstraktnej štruktúry počítačového jazyka, ktorá môže byť definovaná prostredníctvom reprezentácie doménového modelu, v ktorom sú definované jednotlivé doménové pojmy a ich vzájomné väzby. Na základe abstraktnej štruktúry je definovaná sémantika jazyka ako aj konkrétna reprezentácia viet jazyka. Zároveň je možné týmto poukázať na podobnosť metamodelov a bezkontextových gramatík pre vyjadrenie štruktúry počítačových jazykov.

V súčasnosti je jedným z najpoužívanejších jazykov pre vytváranie systémových modelov jazyk UML [99]. Doménové modely navrhovaných počítačových jazykov budú zobrazované pre názornosť v diagrame tried jazyka UML, ktorý umožňuje reprezentovať štruktúru domény. Pre potreby navrhnutého generátora jazykových procesorov *YAJCo* bude doménový model reprezentovaný prostredníctvom tried jazyka Java.

Pri vyjadrení štruktúry v doménovom modeli je možné odlíšiť dva základné druhy vzťahov: vzťah „je“ (vzťah špecializácie, angl. is–a relationship) [71][72] a vzťah „má“ (vzťah vlastníctva, angl. has-a relationship). Príklad vzťahu „je“ je uvedený na Obr. 18. Na obrázku je možné vidieť, že pojem `While` je zároveň pojmom typu `Statement` (cyklus while je príkazom) a pojem `If` je pojmom typu `Statement` (podmienený príkaz if je príkazom). To zároveň znamená, že na mieste kde je požadovaný príkaz `Statement` je možné použiť `While` alebo `If` [72].



Obr. 18: Príklad vzťahu „je“

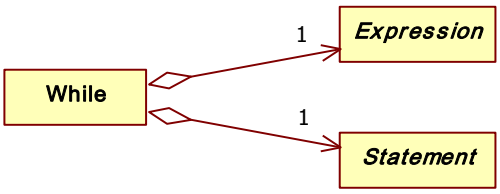
Vzťah „je“ je možné reprezentovať v jazyku Java prostredníctvom konštrukcie `extends` alebo prostredníctvom konštrukcie `implements`. Predošlý príklad s príkazmi imperatívneho jazyka je možné reprezentovať v jazyku Java nasledovne.

```
class If extends Statement {}
class While extends Statement {}
```

Tento vzťah bude pre vytvorený generátor jazykových procesorov zodpovedať nasledujúcemu zápisu v BNF (`Statement`, `If` a `While` sú neterminálne symboly).

```
Statement ::= If | While
```

Príklad vzťahu „má“ je zobrazený diagramom tried na Obr. 19. Pojem `While` je zložený pojem obsahujúci dva pojmy: `Statement` a `Expression` (príkaz cyklu while obsahuje výraz reprezentujúci podmienku a príkaz, ktorý sa bude opakovať pokiaľ je podmienka splnená).



Obr. 19: Príklad vzťahu „má“

Pri vzťahu „má“ sa uvádza aj počet výskytov pojmov vo vzťahu. V uvedenom príklade je to jeden: cyklus while má jednu podmienku a obsahuje jeden príkaz. V jazyku Java je možné zapísať tento vzťah prostredníctvom premenných objektu definovaných v triede.

```
class While {
    Expression expression;
    Statement statement;
}
```

Vzťah „má“ je možné reprezentovať v BNF prostredníctvom zápisu (`While`, `Expression` a `Statement` sú neterminálne symboly).

```
While ::= Expression Statement
```

Prostredníctvom pojmov a ich vzťahov je možné definovať abstraktnú štruktúru domény, ktorá bude slúžiť pre generátor jazykových procesorov pre definíciu abstraktnej štruktúry jazyka. Podobnosti bezkontextových gramatík a metamodelov v kontexte definície jazykov sa venuje [39], ktorá však používa metamodelovanie hlavne na definíciu jazykov s grafickou konkrétnou syntaxou. Štruktúru jazyka je teda možné definovať nielen prostredníctvom bezkontextovým gramatík ale aj metamodelov.

5.1.1. Doménový model aritmetických výrazov

V tejto časti je uvedený príklad špecifikácie doménového modelu pre jednoduchý jazyk aritmetických výrazov. V úvode je formálne definovaná abstraktná syntax a sémantika jazyka aritmetických výrazov. Neskôr je v tejto časti opísaná definícia abstraktnej syntaxe jazyka doménovým modelom použitím diagramov tried a tried jazyka Java. Doménový model reprezentuje abstraktnú syntax jazyka aritmetických operátorov vo forme doménových tried a ich vzťahov. Tento model bude v nasledujúcej časti rozšírený o špecifikáciu konkrétnej syntaxe vo forme anotácií, z ktorého je generovaný jazykový procesor vytvoreným generátorom jazykových procesorov *YAJCo*.

Jednoduchý jazyk aritmetických výrazov opisuje aritmetické výrazy so základnými celočíselnými binárnymi operáciami súčtu, rozdielu, súčinu, podielu a unárnou operáciou určenia opačného čísla. Vo výrazoch sa vyskytujú celé čísla. Syntax jazyka aritmetických výrazov je možné vyjadriť formálne prostredníctvom produkčných pravidiel BNF. Nech e, e_1, e_2, n sú metapremenné syntaktických oblastí **Expression** a **Number**.

$$e, e_1, e_2 \in \textbf{Expression}$$
$$n \in \textbf{Number}$$

Potom produkčné pravidlá pre definíciu abstraktnej syntaxe jazyka majú tvar:

$$e ::= \text{Number } n \mid \text{UnaryMinus } e \mid \text{Add } e_1\ e_2 \mid \text{Sub } e_1\ e_2 \mid \text{Mul } e_1\ e_2 \mid \text{Div } e_1\ e_2$$

Úmyselne pri definícii abstraktnej syntaxe nie je použitý infixný zápis, ktorý by mohol byť podobný zápisu pre definíciu konkrétnej syntaxe jazyka aritmetických výrazov. Prefixné mená slúžia na jednoznačnú identifikáciu druhu aritmetického výrazu v produkčných pravidlách a sémantických rovniciach. Vychádzajúc z uvedených syntaktických oblastí je zadefinovaná sémantika konštrukcií jazyka aritmetických výrazov. Význam aritmetických výrazov je definovaný sémantickou funkciou *Eval* zo syntaktickej oblasti **Expression** do sémantickej oblasti celých čísel **Z**.

$$Eval : \textbf{Expression} \rightarrow \textbf{Z}$$

Sémantická funkcia *Value* transformuje hodnotu zo syntaktickej oblasti zápisu celých čísel **Number** na sémantickú oblasť celých čísel **Z** (hodnota čísla).

$$Value : \textbf{Number} \rightarrow \textbf{Z}$$

Sémantické rovnice definujúce funkciu *Eval* sú nasledovné:

$$\begin{aligned}
&Eval\ [\![\text{Number } n]\!] &&= Value\ [\![n]\!] \\
&Eval\ [\![\text{UnaryMinus } e]\!] &&= -\ Eval\ [\![e]\!] \\
&Eval\ [\![\text{Add } e_1\ e_2]\!] &&= Eval\ [\![e_1]\!] + Eval\ [\![e_2]\!] \\
&Eval\ [\![\text{Sub } e_1\ e_2]\!] &&= Eval\ [\![e_1]\!] - Eval\ [\![e_2]\!] \\
&Eval\ [\![\text{Mul } e_1\ e_2]\!] &&= Eval\ [\![e_1]\!] * Eval\ [\![e_2]\!] \\
&Eval\ [\![\text{Div } e_1\ e_2]\!] &&= Eval\ [\![e_1]\!] / Eval\ [\![e_2]\!]
\end{aligned}$$

Sémantiku jazyka aritmetických výrazov je možné definovať aj iným spôsobom ako transformáciou do sémantickej oblasti celých čísel. V nasledujúcej časti bude sémantika jazyka aritmetických výrazov definovaná prostredníctvom transformácie do postupnosti inštrukcií abstraktného stroja.

$$\textbf{Instruction} = \{\ \text{PUSH } n, \text{NEG, ADD, SUB, MUL, DIV}\ \}$$

Sémantická funkcia *Code* definuje transformáciu zo syntaktickej oblasti **Expression** do sémantickej oblasti postupností inštrukcií **Instruction***.

$$Code : \textbf{Expression} \rightarrow \textbf{Instruction}^*$$

Vymenované inštrukcie sú inštrukciami abstraktného stroja so zásobníkom. Predpokladom je existencia definície sémantiky týchto inštrukcií a abstraktného stroja podľa [88]. Sémantické rovnice definujúce funkciu *Code* sú nasledovné:

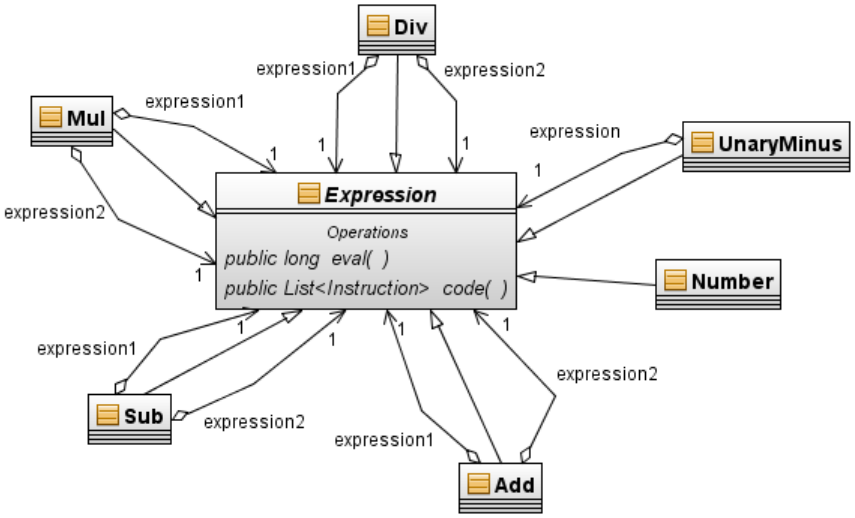
$$\begin{aligned}
&Code\ [\![\text{Number } n]\!] &&= [\text{PUSH } n] \\
&Code\ [\![\text{UnaryMinus } e]\!] &&= Code\ [\![e]\!] ++ [\text{NEG}] \\
&Code\ [\![\text{Add } e_1\ e_2]\!] &&= Code\ [\![e_1]\!] ++ Code\ [\![e_2]\!] ++ [\text{ADD}] \\
&Code\ [\![\text{Sub } e_1\ e_2]\!] &&= Code\ [\![e_1]\!] ++ Code\ [\![e_2]\!] ++ [\text{SUB}] \\
&Code\ [\![\text{Mul } e_1\ e_2]\!] &&= Code\ [\![e_1]\!] ++ Code\ [\![e_2]\!] ++ [\text{MUL}] \\
&Code\ [\![\text{Div } e_1\ e_2]\!] &&= Code\ [\![e_1]\!] ++ Code\ [\![e_2]\!] ++ [\text{DIV}]
\end{aligned}$$

Zápis $[x]$ označuje postupnosť s jediným prvkom x. Zápis $[x_1, \ldots, x_n]$ označuje n-prvkovú postupnosť (pre $n \geq 0$). Operátor ++ slúži na spájanie postupností inštrukcií. Operátor ++ je doľava asociujúci operátor a je definovaný nasledovne.

$$(++) : \textbf{Instruction}^* \rightarrow \textbf{Instruction}^* \rightarrow \textbf{Instruction}^*$$
$$[x_1, \ldots, x_m] ++ [y_1, \ldots, y_n] = [x_1, \ldots, x_m, y_1, \ldots, y_n],\ m \geq 0,\ n \geq 0$$

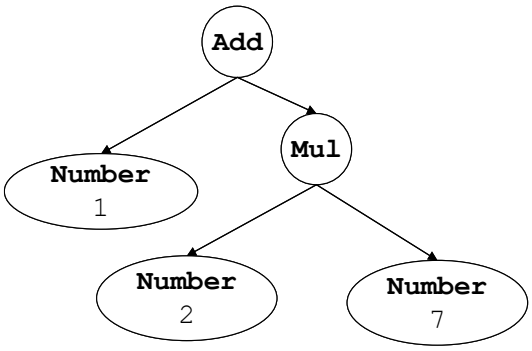
Teraz bude definovaná syntax a sémantika jazyka aritmetických výrazov prostredníctvom doménového modelu podľa formálne definovanej syntaxe a sémantiky. Základnou triedou

je abstraktná trieda `Expression` definujúca aritmetický výraz. Táto trieda nebude mať konkrétnu reprezentáciu v textovej forme a predstavuje len abstraktný pojem. Aritmetickým výrazom neobsahujúcim podvýrazy je celé číslo reprezentované triedou `Number`. Vzťah „číslo je výraz“ je reprezentovaný v doménovom modeli prostredníctvom relácie „je“, reprezentovanej v diagrame tried dedením. Aritmetickým výrazom môže byť taktiež výraz s unárnou operáciou určenia opačného čísla (`UnaryMinus`) ako aj výrazy pre binárne operácie súčtu (`Add`), rozdielu (`Sub`), súčinu (`Mul`) a podielu (`Div`). Tieto výrazy sú aritmetickými výrazmi, čo je v doménovom modeli vyjadrené prostredníctvom relácie „je“. Výrazy binárnych operácií obsahujú vždy dva podvýrazy a výrazy unárnej operácie obsahuje práve jeden podvýraz. Doménový model vyjadrený prostredníctvom diagramu tried je zobrazený na Obr. 20.



Obr. 20: Diagram tried jednoduchého jazyka aritmetických výrazov

Abstraktný syntaktický strom vety z jazyka pozostáva z uzlov konkrétnych typov a hrán. Každý uzol abstraktného syntaktického stromu má definovaný typ doménovou triedou. Príklad abstraktného syntaktického stromu pre vetu z jazyka jednoduchých aritmetických výrazov je na Obr. 21.



Obr. 21: Abstraktný syntaktický strom vety z jazyka aritmetických výrazov

Typ uzla je definovaný v obrázku pre každý uzol uvedením mena príslušnej doménovej triedy. V abstraktnom syntaktickom strome sa nachádzajú uzly konkrétnych doménových tried. Abstraktný syntaktický strom je vytvorený jazykovým procesorom zo vstupnej vety.

Z pohľadu definovania sémantiky je možné každý výraz vyhodnotiť na celé číslo resp. na nedefinovanú hodnotu (v prípade výrazu delenia číslom 0). Toto je v modeli

vyjadrené operáciou `eval()` triedy `Expression`. Táto operácia zodpovedá sémantickej funkcii *Eval*. Ďalšou možnosťou vyjadrenia sémantiky je transformácia do jazyka inštrukcií abstraktného stroja, čo je v modeli vyjadrené operáciou `code()` triedy `Expression`, ktorá zodpovedá sémantickej funkcii *Code*. Operácie `eval()` a `code()` je možné považovať za sémantické funkcie. Tieto operácie budú implementované v konkrétnych doménových triedach pre jednotlivé konštrukcie abstraktnej syntaxe (`Number`, `UnaryMinus`, `Add`, `Sub`, `Mul`, `Div`).

Pre potreby generátora jazykových procesorov *YAJCo* je nutné vytvoriť triedy v jazyku Java, ktoré budú reprezentovať doménové triedy z jazyka aritmetických výrazov v korešpondencii s modelom z Obr. 20. Trieda `Expression` pre abstrakciu aritmetického výrazu bude mať nasledovné vyjadrenie.

```
public abstract class Expression {
    //Sémantické funkcie
    public abstract long eval();
    public abstract List<Instruction> code();
}
```

Trieda `Number` reprezentujúca celé číslo má nasledovné vyjadrenie v jazyku Java. V komentároch sú uvedené príslušné sémantické funkcie.

```
public final class Number extends Expression {
    private final long value;

    public Number(long value) {
        this.value = value;
    }

    // Eval ⟦Number n⟧ = Value ⟦n⟧
    public long eval() {
        return value;
    }

    // Code ⟦Number n⟧ = [PUSH n]
    public List<Instruction> code() {
        return list(new PUSH(value));
    }
}
```

Unárna operácia určenia opačného čísla `UnaryMinus` je implementovaná nasledovne.

```
public final class UnaryMinus extends Expression {
    private final Expression expression;

    public UnaryMinus(Expression expression) {
        this.expression = expression;
    }

    // Eval ⟦UnaryMinus e⟧ = – Eval ⟦e⟧
    public long eval() {
        return -expression.eval();
    }

    // Code ⟦UnaryMinus e⟧ = Code ⟦e⟧++ [NEG]
    public List<Instruction> code() {
        return concat(expression.code(), list(new NEG()));
    }
```

```
}
```

Použité metódy `list()` a `concat()` sú určené na vytvorenie zoznamu inštrukcií obsahujúceho práve jednu inštrukciu (zodpovedá zápisu [*x*]) a spájanie zoznamov inštrukcií (zodpovedá operácii ++ z definície formálnej sémantiky). Všetky binárne operácie sú implementované podobným spôsobom, preto je uvedená len implementácia výrazu operácie súčtu, ktorý je realizovaná prostredníctvom triedy `Add`. Triedy `Sub`, `Mul`, `Div` sú implementované analogicky. Abstraktná syntax nedefinuje konkrétnu formu zápisu iba syntaktické kategórie a ich štruktúru. Označenie podvýrazov `expression1` a `expression2` indexom je použité len na odlíšenie jednotlivých podvýrazov. Abstraktná syntax nedefinuje formu zápisu operátora (infixný, prefixný, postfixný) ani použitý symbol (napríklad symbol +, resp. `sucet`). Operácia súčtu definovaná v doménovom modeli vyjadrenom v jazyku Java má nasledujúci tvar.

```
public final class Add extends Expression {
    private final Expression expression1;
    private final Expression expression2;

    public Add(Expression expression1, Expression expression2) {
        this.expression1 = expression1;
        this.expression2 = expression2;
    }

    // Eval ⟦Add e1 e2⟧ = Eval ⟦e1⟧ + Eval ⟦e2⟧
    public long eval() {
        return expression1.eval() + expression2.eval();
    }

    // Code ⟦Add e1 e2⟧ = Code ⟦e1⟧ ++ Code ⟦e2⟧++ [ADD]
    public List<Instruction> code() {
        return concat(expression1.code(), expression2.code(),
            list(new ADD()));
    }
}
```

Vo všetkých konkrétnych triedach sú implementované sémantické funkcie `eval()` a `code()`. Samotné triedy teda definujú abstraktné konštrukcie a zároveň typy uzlov v abstraktnom syntaktickom strome. Sémantické funkcie teda definujú transformácie do sémantických oblastí. V prípade aritmetických výrazov je to oblasť celých čísel pre funkciu `eval()` a postupnosť inštrukcií pre funkciu `code()`.

Implementovaný jazyk aritmetických výrazov je zároveň ukážkou možnosti separácie implementácií jazykových konštrukcií v procesore na základe pojmov jazyka. Jednotlivé konštrukcie abstraktnej syntaxe sú implementované v doménovom modeli prostredníctvom samostatných tried jazyka Java. Vyšší stupeň separácie je možné dosiahnuť oddelením implementácie jednotlivých sémantických funkcií, ktoré je možné realizovať použitím princípov aspektovo-orientovaného programovania [24][29]. Primárnou záležitosťou pri takomto riešení bude abstraktná syntax a pretínajúcimi záležitosťami budú sémantické funkcie.

5.2. Generovanie jazykového procesora z doménového modelu

Pre textové vyjadrenie jazyka prostredníctvom konkrétnej syntaxe je nutné rozšíriť model o ďalšie informácie – lexikálne jednotky, prioritu operátorov a zátvorkované výrazy.

O tieto informácie je nutné rozšíriť doménový model, aby bolo možné vygenerovať jazykový procesor. Textová forma jazyka je špecifikovaná doménovým modelom rozšíreným o informácie potrebné pre definovanie konkrétnej syntaxe nezávisle od technológie rozpoznania vety. Na základe analýzy viacerých počítačových jazykov boli identifikované vzory pre prechod od abstraktnej syntaxe ku konkrétnej syntaxi. Pre jednotlivé vzory bol navrhnutý spôsob ich označenia v doménovom modeli. Identifikované boli vzory pre kompozíciu lexikálnych jednotiek jazyka a doménových tried (pojmy jazyka).

Výhodou takéhoto riešenia je možnosť automatizovaného a asistovaného prechodu od abstraktnej syntaxe ku konkrétnej. Abstraktná syntax je dôležitá pre definovanie sémantiky. Konkrétna syntax je odrazom abstraktnej syntaxe, ktorá umožňuje spracovanie dostupnými technológiami s cieľom zachovania jednoznačnosti pri rozpoznávaní vety. Pri výbere požadovanej technológie spracovania je možné automatizovane identifikovať možnosti prechodu od abstraktnej syntaxe ku konkrétnej syntaxi, upozorňovať na nejednoznačnosti definovanej konkrétnej syntaxe, respektíve odporúčať zmeny technológie spracovania viet z jazyka.

Generátor jazykových procesorov *YAJCo* je implementovaný v jazyku Java. Vďaka anotáciám v jazyku Java je možné pridať do štruktúry tried dodatočné informácie, ktoré nemajú priamy vplyv na vykonávanie ale môžu byť interpretované anotačnými procesormi. Pre potreby generátora jazykových procesorov tieto anotácie slúžia na doplnenie štruktúry tried o informácie z diagramu tried a určenie vzťahu medzi konkrétnou a abstraktnou syntaxou. Generátor jazykových procesorov *YAJCo* je implementovaný ako anotačný procesor. Anotácie vytvoreného generátora jazykových procesorov je možné rozdeliť do štyroch kategórií:

1. Anotácie pre dodefinovanie štruktúry: `@Optional`, `@Range`.
2. Anotácie pre špecifikáciu konkrétnej syntaxe: `@Before`, `@After`, `@Separator`, `@Token`.
3. Anotácie pre definíciu operátorov: `@Operator`, `@Parentheses`.
4. Anotácie pre definíciu procesora: `@Parser`, `@TokenDef`, `@Skip`, `@Option`, `@Exclude`.

Tieto anotácie sú v podstate doménovo-špecifické anotácie pre doménu generátorov jazykových procesorov. Ich význam je definovaný v kompozícii s konštrukciami jazyka Java. V nasledujúcej časti sú jednotlivé anotačné typy stručné opísané. Vzory použitia anotačných typov a ich korešpondencia s produkčnými pravidlami bezkontextových gramatík je uvedená v časti 5.2.5.

5.2.1. Anotácie pre dodefinovanie štruktúry

Objektové jazyky akými sú napríklad Java a C# nemajú definované všetky konštrukcie, ktoré by zodpovedali zápisom štruktúry v diagramoch tried. Jednou z chýbajúcich konštrukcií je určenie násobnosti a povinnosti väzby objektov tried. Pre vyplnenie tejto medzery v objektovom jazyku Java pre použitie v navrhovanej metóde konštrukcie jazykových procesorov boli vytvorené dva anotačné typy. Tieto anotačné typy umožňujú definovať povinnosť väzby ako aj násobnosť.

- Anotačný typ `Optional` definuje voliteľnosť väzby.
- Anotačný typ `Range` sa používa pre definovanie násobnosti s možnosťou určenia minimálneho a maximálneho počtu objektov vo vzťahu.

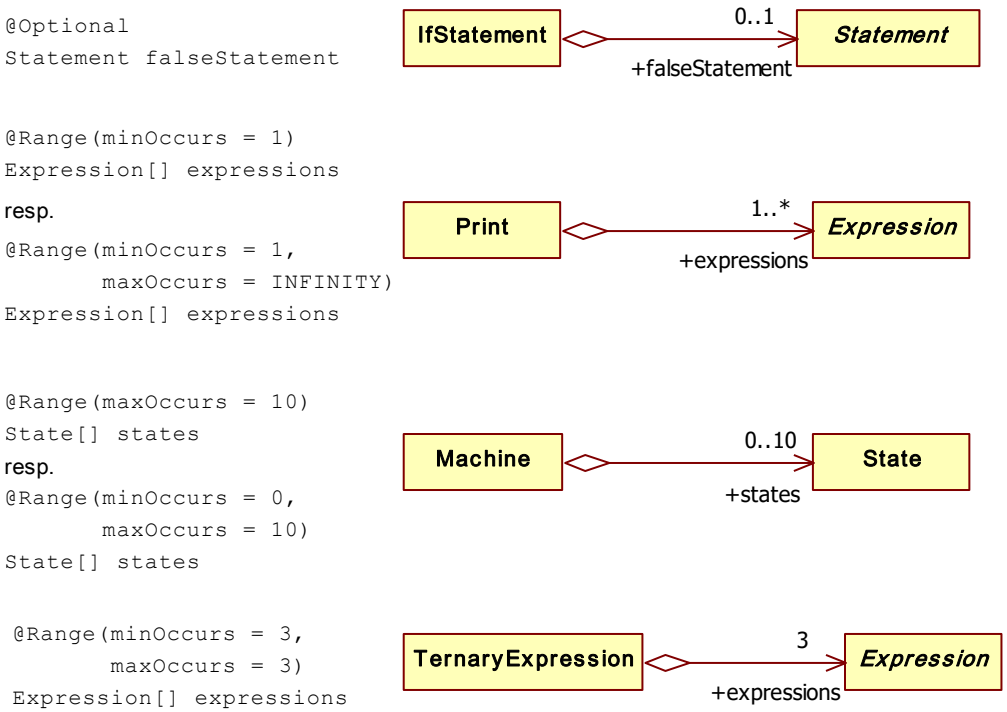
Anotačný typ `Optional` je definovaný v jazyku Java ako bezparametrická označovacia anotácia (analógia návrhového vzoru označovacieho rozhrania, angl. marker interface [38]).

```
public @interface Optional {
}
```

Anotácia `@Range` je použiteľná len pre premenné typu pole. V prípade, že anotácia `@Range` nie je uvedená pri premennej typu pole predpokladá sa rozsah bez obmedzenia (0..*). Anotačný typ `Range` definuje dva parametre, ktorých hodnota určuje minimálny respektíve maximálny počet výskytov označeného elementu vo vzťahu.

```
public @interface Range {
    public static final int INFINITY = -1;

    int minOccurs() default 0;

    int maxOccurs() default INFINITY;
}
```

Anotácie `@Optional` a `@Range` je možné použiť na označenie parametrov konštruktora. Obr. 22 je ukážkou použitia týchto anotačných typov s ekvivalentom uvedeným v diagrame tried. Tieto anotačné typy nie sú priamo späté s konkrétnou syntaxou. Anotácie `@Optional` a `@Range` majú význam pri definícii abstraktnej syntaxe. Z tohto dôvodu sú navrhnuté anotačné typy použiteľné v širšom kontexte ako len pre navrhnutý generátor jazykových procesorov.



Obr. 22: Anotačné typy `@Optional`, `@Range`

5.2.2. Anotácie pre špecifikáciu konkrétnej syntaxe

Pri prechode od abstraktnej syntaxe ku konkrétnej syntaxi je nutné rozšíriť jazyk o definíciu lexikálnych jednotiek, ktoré špecifikujú zápis jednotlivých pojmov v textovej forme konkrétnej syntaxe. Zatiaľ čo abstraktná syntax sa definuje pre potreby určenia významu jazyka na základe určenia významu jednotlivých konštrukcií, konkrétna syntax definuje zápis abstraktných konštrukcií prostredníctvom konkrétnych symbolov.

- Anotácie `@Before`, `@After`, `@Separator` slúžia na označenie a určenie oddeľovačov konštrukcií v konkrétnej syntaxi a ich zápis nemá vplyv na sémantiku ale len na jednoznačné určenie štruktúry (napr. symbol `function`, `begin`). Anotácia `@Before` určuje lexikálne symboly, ktoré budú predchádzať označenej konštrukcii (napr. blok príkazov bude začínať symbolom `begin`). Anotácia `@After` vymenováva lexikálne symboly, ktoré budú nasledovať za označenou konštrukciou (napr. blok príkazov bude končiť symbolom `end`). Anotácia `@Separator` sa používa na určenie oddelenia postupnosti konštrukcií toho istého typu (napr. čiarka bude oddeľovať jednotlivé parametre funkcie).
- Anotácia `@Token` sa používa na špecifikáciu lexikálneho symbolu, ktorý je podstatný pre abstraktnú syntax a sémantiku z pohľadu jeho výskytu a zápisu (napr. číslo, identifikátor). Táto anotácia predstavuje väzbu na parameter, do ktorého bude uložený zápis (resp. hodnota ak dôjde k transformácií) spracovaného lexikálneho symbolu.

Anotácie `@Before`, `@After` a `@Separator` majú jediný parameter, ktorý predstavuje postupnosť mien lexikálnych symbolov, ktoré predchádzajú, nasledujú, resp. oddeľujú konštrukcie pri ktorých sú uvedené. Jednotlivé lexikálne symboly sú pomenované. V anotáciách sú uvedené mená lexikálnych symbolov (časť 5.2.4). Tieto anotačné typy sú definované v jazyku Java nasledovne.

```
public @interface Before {
    String[] value();
}

public @interface After {
    String[] value();
}

public @interface Separator {
    String[] value();
}
```

Anotácie `@Before` a `@After` je možné použiť na označenie parametrov konštruktora ako aj pre samotný konštruktor. Anotáciu `@Separator` je možné použiť s parametrom konštruktora štruktúrovaného typu pole. Nasledujúci príklad definície zápisu príkazu `print` prostredníctvom anotácií `@Before`, `@After` a `@Separator` je vybraný zo štruktúrovaného imperatívneho jazyka.

```
public class Print extends Statement {
    private final Expression[] expressions;

    @Before("PRINT")
    @After("SEMICOLON")
    public Print(
```

```
            @Separator("COMMA")
            @Range(minOccurs = 1) Expression[] expressions) {
        this.expressions = expressions;
    }
}
```

Takto definovaný príkaz print má nasledujúcu konkrétnu syntax (pre $n \geq 1$).

```
print výraz1, ... , výrazn;
```

Anotačný typ `Token` má jediný parameter, ktorým je meno lexikálnej jednotky. Anotáciu `@Token` je možné použiť s parametrom konštruktora. Anotácia `@Token` je definovaná v jazyku Java nasledovne.

```
public @interface Token {
    String value() default "";
}
```

V prípade, že v anotácii `@Token` nie je uvedené meno lexikálnej jednotky, bude meno lexikálneho symbolu odvodené z mena parametra, ktorý je označený anotáciou `@Token`. Na základe tohto pravidla sú dva nasledujúce riadky ekvivalentné pri spracovaní generátorom *YAJCo*.

```
@Token("VALUE") long value
@Token long value
```

5.2.3. Anotácie pre definíciu operátorov

Pri spracovaní textových jazykov sú operátory jedným z často sa opakujúcich vzorov. Skoro každá kniha zaoberajúca sa jazykovými procesormi a spracovaním programovacích jazykov sa venuje operátorom [94]. Pri definícii operátorov sa uvádza ich priorita ako aj asociatívnosť. Niektoré generátory jazykových procesorov priamo obsahujú konštrukcie pre opis operátorov. Generátory jazykových procesorov YACC [49] a Bison [1] obsahujú konštrukcie `%left`, `%right`, a `%nonassoc` na určenie priority a asociatívnosti operátorov. Pre potreby označenia operátorov je definovaný v navrhnutom generátore anotačný typ `Operator` a enumeračný typ `Associativity` pre určenie typu asociatívnosti operátora.

```
public enum Associativity {
    LEFT, RIGHT, NONE, AUTO
}

public @interface Operator {
    int priority() default 1;

    Associativity associativity() default Associativity.AUTO;
}
```

Priorita operátora je určená anotáciou číselne. Priorita sa určuje absolútne a nie relatívne vzhľadom k priorite iného operátora. Menšie číslo znamená nižšiu prioritu operátora, väčšie číslo znamená vyššiu prioritu operátora. V prípade, že dva rôzne operátory majú uvedené to isté číslo vyjadrujúce prioritu prostredníctvom anotácie `@Operator`, tieto operátory majú rovnakú prioritu.

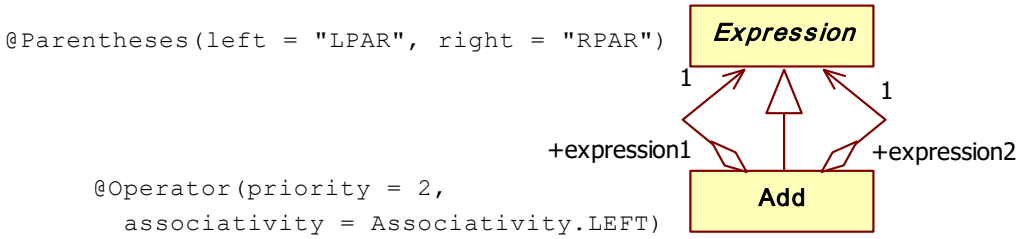
Asociatívnosť operátora sa určuje zvolením typu asociatívnosti – doľava, doprava, neasociatívny. Všetko operátory tej istej priority musia asociovať rovnako. Predvolený typ asociatívnosti (hodnota `AUTO`) zabezpečí voľbu asociatívnosti automaticky podľa

charakteru operátora. Predvolená asociatívnosť pre binárne operátory je doľava. Ak je na danej úrovni priority aspoň jeden operátor označený ako asociatívny doprava, všetky operátory tejto priority označené hodnotou asociatívnosti `AUTO` budú asociatívne doprava.

Pre vytváranie zátvorkovaných výrazov je definovaný anotačný typ `Parentheses`. Táto anotácia špecifikuje lexikálny symbol pre pravú a ľavú zátvorku, ktorý bude použitý v zátvorkovaných výrazoch v zápise vety.

```
public @interface Parentheses {
    String left();

    String right();
}
```

Anotáciu `@Operator` je možné použiť len na označenie konštruktora. Anotácia `@Parentheses` sa používa pre označenie abstraktnej triedy alebo rozhrania. Na Obr. 23 je uvedený príklad použitia anotácií `@Operator` a `@Parentheses` pri definovaní konkrétnej syntaxe zátvorkovaných aritmetických výrazov s binárnym operátorom súčtu.



Obr. 23: Použitie anotácií `@Operator` a `@Parentheses`

5.2.4. Anotácie pre definíciu procesora

Anotácie pre definíciu procesora majú špeciálne postavenie. Neslúžia na anotovanie jazykových elementov ale na špecifikáciu vytváraného jazykového procesora. Základnou anotáciou je anotácia `@Parser`, ktorá definuje generovaný procesor. Tieto anotácie by mohli byť nahradeným konfiguračným jazykom, pretože sa neviažu na jazykové elementy doménového modelu. V tomto prípade sú anotácie vloženým doménovo-špecifickým jazykom (časť 4.2). Výhodou konfigurácie prostredníctvom anotácií je podobnosť s anotovaním doménového modelu (rovnaká syntax), ľahšia integrovateľnosť s doménovým modelom, jednoduchšia použiteľnosť v štandardných jazykových prostrediach ako aj možnosť spracovania anotačným procesorom. Anotačný typ `@Parser` má nasledujúcu definíciu.

```
public @interface Parser {
    String className();

    String mainNode();

    TokenDef[] tokens() default {};

    Skip[] skips() default {};

    Option[] options() default {};
}
```

Anotácia `@Parser` sa používa pre označenie balíka v jazyku Java s domenovým modelom. Generátor jazykových procesorov *YAJCo* začína svoju činnosť práve

spracovaním anotácie `@Parser`. Na základe tejto anotácie získava generátor hlavnú (koreňovú) triedu v doménovom modeli a následne spracováva celý doménový model počnúc touto triedou. Táto anotácia taktiež určuje meno hlavnej triedy vygenerovaného jazykového procesora a definuje lexikálne symboly, biele znaky (oddeľovače) a nastavenia pre generátor.

Anotácie `@TokenDef`, `@Skip`, `@Option` nie sú určené pre označovanie jazykových elementov v jazyku Java. Ich použitie je výhradne možné spolu s anotáciou `@Parser`. Anotácia `@TokenDef` pomenováva lexikálne jednotky definované vo forme regulárnych výrazov. Anotácia `@Skip` definuje biele znaky pre vytváraný jazyk. Anotácia `@Option` je určená pre konfiguráciu generátora jazykových procesorov. Tieto anotačné typy sú definované nasledovne.

```
public @interface TokenDef {
    String name();

    String regexp();
}

public @interface Skip {
    String value();
}

public @interface Option {
    String name();
    String value();
}
```

Poslednou definovanou anotáciou je anotácia `@Exclude`. Anotačný typ `@Exclude` definuje bezperametrické anotácie slúžiace na označenie elementov, ktoré nemajú byť spracovávané generátorom jazykových procesorov.

```
public @interface Exclude {
}
```

Anotačný typ `Exclude` je možné použiť na anotovanie typov v jazyku Java (triedy, enumeračné typy a rozhrania).

5.2.5. Anotovaný model jazyka aritmetických výrazov

V časti 5.1.1 bol definovaný jednoduchý jazyk aritmetických výrazov. V tejto časti bude ukázaný na príklade tohto jazyka prechod od abstraktnej syntaxe ku konkrétnej prostredníctvom označenia doménových tried anotáciami. Pre definovanie konkrétnej syntaxe je nutné určiť spôsob zápisu čísla a formu zápisu operácií. Čísla budú reprezentované arabskými číslicami v desiatkovej sústave. Operácie súčtu, rozdielu, súčinu a podielu budú reprezentované vo forme výrazov s infixnými operátormi. Ako symboly pre operátory budú použité zaužívané znaky +, -, * a /. Operácia určenia záporného čísla bude reprezentovaná prefixným unárnym operátorom so symbolom -. Priorita a asociatívnosť operátorov je uvedená v Tab. 1.

Operátor	Priorita	Asociatívnosť
+, - (binárne)	1 (najnižšia)	doľava
*, /	2	doľava
- (unárne)	3 (najvyššia)	doprava

Tab. 1: Priorita a asociatívnosť operátorov v jazyku aritmetických výrazov

V nasledujúcom texte budú rozšírené doménové triedy o anotácie vyjadrujúce konkrétnu syntax. Abstraktný syntaktický strom je vytváraný z objektov doménových tried. Pre vytváranie objektov sú používané konštruktory, ktoré sú základnými elementmi, ktoré budú anotované definovanými anotáciami. Práve vďaka anotovaniu konštruktorov a ich parametrov je možné definovať konkrétnu syntax s väčšou mierou voľnosti ako pri anotovaní premenných objektu. Konštruktory majú definovanú postupnosť parametrov umožňujúcu definovať poradie symbolov v konkrétnej syntaxi. Definícia konkrétnej syntaxe nemá vplyv na sémantické funkcie, ktoré sú závislé len od abstraktnej syntaxe. Z tohto dôvodu v príkladoch nebudú uvedené sémantické funkcie. Trieda `Number` môže byť rozšírená o anotácie pre konkrétnu syntax nasledovne.

```
public final class Number extends Expression {
    private final long value;

    public Number(@Token("VALUE") long value) {
        this.value = value;
    }

    //Nezmenené sémantické funkcie ...
}
```

Pri implementácií generátora jazykových procesorov bol jeden z cieľov minimalizovať počet definovaných anotačných typov ako aj nevyhnutnosť ich použitia na anotovanie. Vďaka tejto vlastnosti nie je nevyhnutné anotovať všetky doménové triedy. To platí aj pre triedu `Number` a preto je možné anotáciu `@Token` vynechať. Trieda `Number` teda ostane neovplyvnená anotáciami pre konkrétnu syntax aj napriek tomu, že bude definovať pravidlo pre konkrétnu syntax.

```
public final class Number extends Expression {
    private final long value;

    public Number(long value) {
        this.value = value;
    }

    //Nezmenené sémantické funkcie ...
}
```

Teraz budú rozšírené triedy binárnych operácií o anotácie označujúce konštruktory ako operátory definujúc prioritu a asociatívnosť. Asociatívnosť nie je nevyhnutné určiť vzhľadom na to, že pre binárne operátory je definovaná predvolená asociatívnosť doprava. Zároveň je nutné uviesť symbol operátora prostredníctvom anotácie `@Before` tak, aby bol definovaný ako infixný.

```
public final class Add extends Expression {
    private final Expression expression1;

    private final Expression expression2;
```

```
        @Operator(priority = 2)
        public Add(Expression expression1,
                @Before("PLUS") Expression expression2) {
            this.expression1 = expression1;
            this.expression2 = expression2;
        }

        //Nezmenené sémantické funkcie ...
    }
```

Rovnakú konkrétnu syntax je možné definovať aj použitím anotácie `@After`. Tento zápis je však ťažšie čitateľný, pretože anotácie sa zapisujú pred elementom, ktorý má byť označený anotáciou.

```
        @Operator(priority = 2)
        public Add(@After("PLUS") Expression expression1,
                Expression expression2) {
            this.expression1 = expression1;
            this.expression2 = expression2;
        }
```

Rovnako budú pridané anotácie aj do doménových tried `Sub`, `Mul` a `Div`. Trieda `UnaryMinus` bude rozšírená o anotáciu `@Operator` a `@Before`.

```
    public final class UnaryMinus extends Expression {
        private final Expression expression;

        @Operator(priority = 3)
        public UnaryMinus(@Before("MINUS") Expression expression) {
            this.expression = expression;
        }

        //Nezmenené sémantické funkcie ...
    }
```

V jazyku aritmetických výrazov by mali byť podporované aj zátvorkované výrazy, čo bude vyznačené anotáciou `@Parentheses` v abstraktnej doménovej triede `Expression`.

```
    @Parentheses(left = "LPAR", right = "RPAR")
    public abstract class Expression {
        //Nezmenené sémantické funkcie ...
    }
```

Pre úplnosť je ešte nutné definovať prostredníctvom anotácie `@Parser` hlavnú (koreňovú) doménovú triedu (v jazyku aritmetických výrazov je to výraz `Expression`), meno triedy vygenerovaného jazykového procesora (`tuke.pargen.test.expression.parser.ExpressionParser`), lexikálne symboly a biele znaky pre konkrétnu syntax nasledovne.

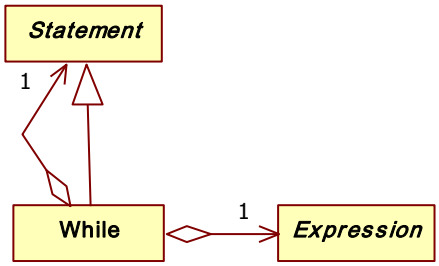
```
    @Parser(
        className =
          "tuke.pargen.test.expression.parser.ExpressionParser",
        mainNode = "Expression",
        tokens = {
            @TokenDef(name = "PLUS",    regexp = "+"),
            @TokenDef(name = "MINUS",   regexp = "-"),
            @TokenDef(name = "STAR",    regexp = "*"),
            @TokenDef(name = "SLASH",   regexp = "/"),
            @TokenDef(name = "LPAR",    regexp = "("),
            @TokenDef(name = "RPAR",    regexp = ")"),
```

```
            @TokenDef(name = "VALUE",   regexp = "[0-9]+")
        },
        skips = {
            @Skip(" "),
            @Skip("\\t"),
            @Skip("\\n"),
            @Skip("\\r")
        }
)
```

5.2.6. Anotačné vzory

Abstraktná syntax je okrem doménových tried definovaná v doménovom modeli aj vzťahom „je“ (konštukcia `extends` a `implements` v jazyku Java) a vzťahom „má“ (premenné objektu definované v triede v jazyku Java). Vzhľadom na to, že konkrétna syntax nemusí byť len jednoduchým rozšírením abstraktnej syntaxe, v navrhnutej metóde sú použité pre definíciu konkrétnej syntaxe konštruktory. Toto riešenie poskytuje dostatočnú flexibilitu pre definíciu konkrétnej syntaxe. Zatiaľ čo vzťah „je“ má rovnaký význam v abstraktnej aj konkrétnej syntaxi, vzťah „má“ je reprezentovaný rôzne v abstraktnej aj konkrétnej syntaxi. Na reprezentáciu vzťahu vlastníctva sú použité výhradne konštruktory.

Rozdiel medzi zápisom abstraktnej a konkrétnej syntaxe je ukázaný v nasledujúcom príkaze cyklu `while` z imperatívneho jazyka. Abstraktná syntax pre príkaz cyklu `while` je zobrazená diagramom tried a taktiež zapísaná zodpovedajúcim produkčným pravidlom BNF v Tab. 2.

Diagram tried	Produkčné pravidlo
	`Statement ::= While | ...` `While ::= Epression Statement`

Tab. 2: Abstraktná syntax pre cyklus `while`

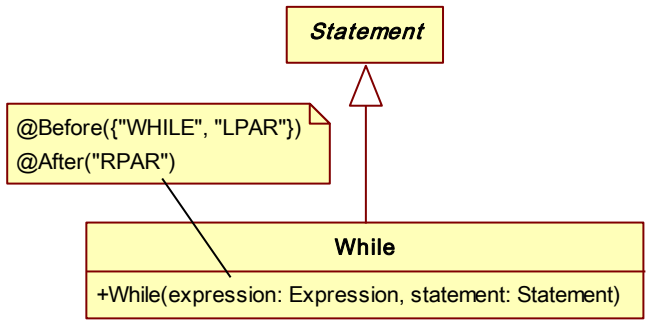
V jazyku Java je abstaktná syntax pre pojem cyklu zapísaná nasledovne.

```
class While extends Statement {
    Statement statement;

    Expression expression;
}
```

Konkrétnu syntax pre pojem cyklu `while` je možné zapísať prostredníctvom diagramu tried a produkčného pravidla podľa Tab. 3. Nasledujúce zápisy použité v definícii konkrétnej syntaxe prostredníctvom produkčných pravidiel sú ekvivalentné.

BNF zápis	Ekvivalentý zápis
`P ::= ε | x`	`P ::= x?`
`P ::= ε | P x`	`P ::= x*`
`P ::= x | P x`	`P ::= x+`

Lexikálne symboly sú označené v špeciálnych zátvorkách prostredníctvom mena lexikálneho symbolu (napr. <LPAR>) alebo v úvodzovkách priamo zápisom lexikálneho symbolu (napr. '(').

Diagram tried	Produkčné pravidlo
	Statement ::= While | ... While ::= 'while' '(' Expression ')' Statement

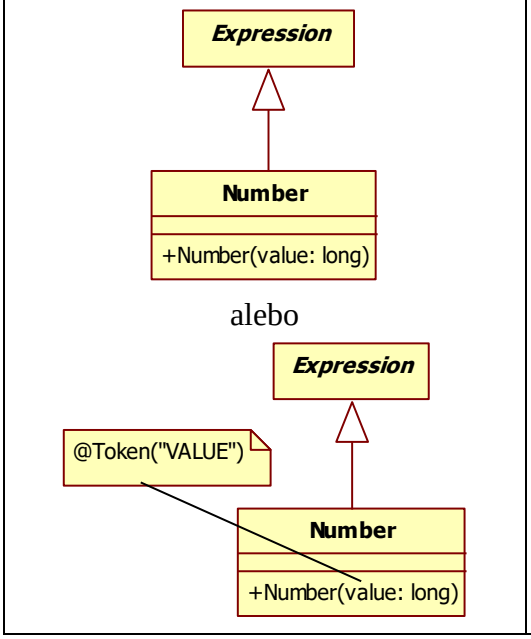
Tab. 3: Konkrétna syntax pre cyklus `while`

Pre explicitnú reprezentáciu anotácií v diagramoch tried je použitý zápis prostredníctvom komentárov, ktorý je najvizuálnejší [14] a umožňuje anotovať aj parametre konštruktora. Čiara spája anotáciu vo forme poznámky s anotovaným elementom. V jazyku Java je konkrétna syntax pre pojem cyklu zapísaná nasledovne.

```
class While extends Statement {
    Expression expression;

    Statement statement;

    While(@Before({"WHILE", "LPAR"})
          @After("RPAR") Expression expression,
          Statement statement) {
        this.expression = expression;
        this.statement = statement;
    }
}
```

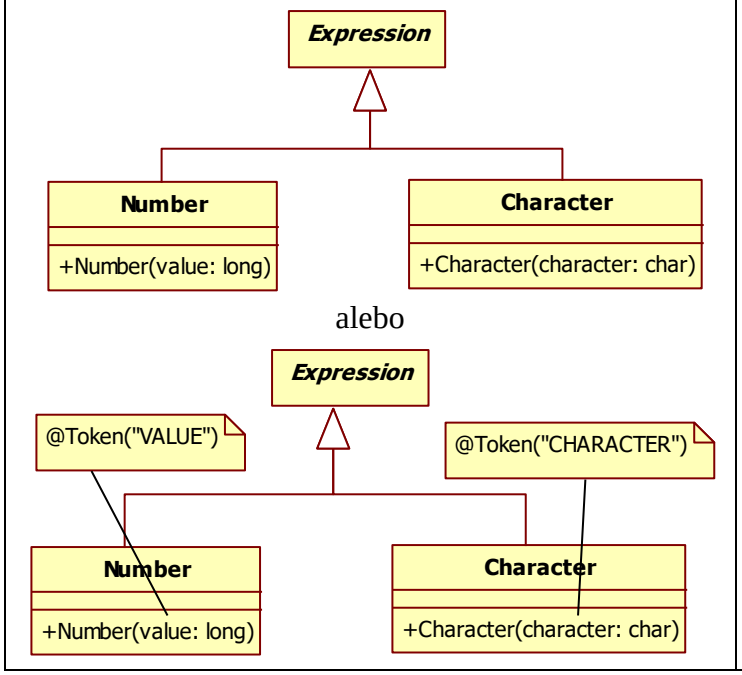
Vzťah anotovaných doménových modelov vyjadrených vo forme diagramov tried s explicitnými atribútmi a produkčnými pravidlami bezkontextových gramatík bude prezentovaný na množine príkladov, ktoré vyjadrujú vzory použitia definovaných anotácií. Príklady uvedené v Tab. 4, Tab. 5 a Tab. 6 sú príkladmi použitia anotácií `@Token`.

Tab. 7, Tab. 8, Tab. 9, Tab. 10 obsahujú príklady pre použitie anotácií `@Before` a `@After` zobrazené prostredníctvom diagramov tried s uvedením ekvivalentného produkčného pravidla bezkontextovej gramatiky.

Diagram tried	Produkčné pravidlo
	`Expression ::= Number` `Number ::= <VALUE>`

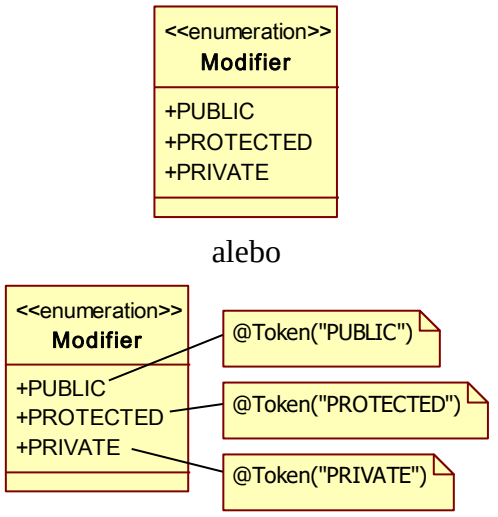
Tab. 4: Vzor použitia anotácie `@Token` pre parameter I.

Diagram tried	Produkčné pravidlo

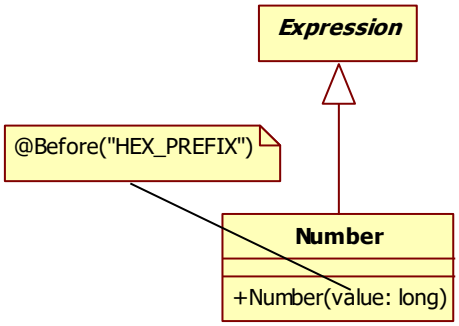
	`Expression ::= Number`
	`  | Character`
	`Number ::= <VALUE>`
	`Character ::= <CHARACTER>`

Tab. 5: Vzor použitia anotácie `@Token` pre parameter II.

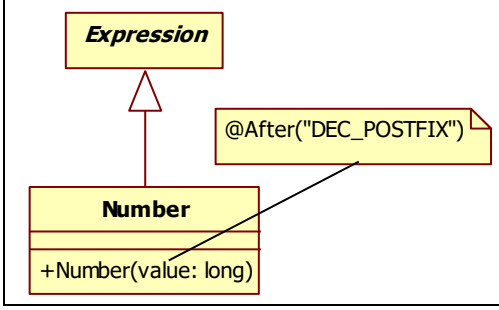
Diagram tried	Produkčné pravidlo

	`Modifier ::= <PUBLIC>`
	`| <PROTECTED>`
	`| <PRIVATE>`

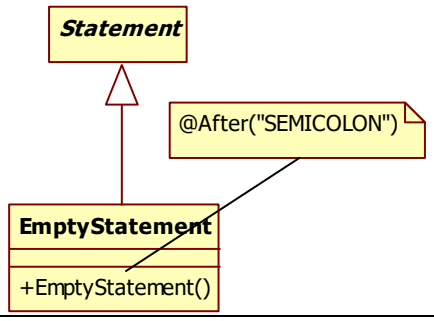
Tab. 6: Vzor použitia anotácie `@Token` pre enumeračný typ

Diagram tried	Produkčné pravidlo
	`Expression ::= Number` `Number ::= <HEX_PREFIX> <VALUE>`

Tab. 7: Vzor použitia anotácie `@Before` pre parameter

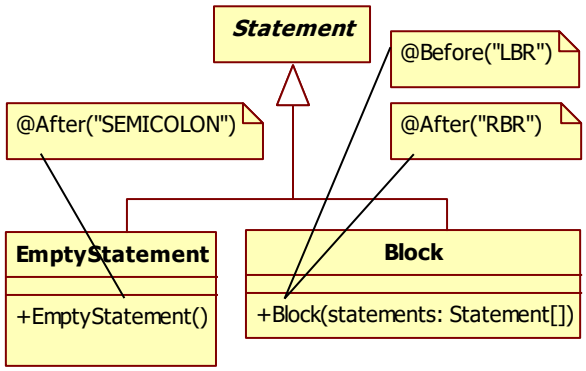
Diagram tried	Produkčné pravidlo
	`Expression ::= Number` `Number ::= <VALUE> <DEC_POSTFIX>`

Tab. 8: Vzor použitia anotácie `@After` pre parameter

Diagram tried	Produkčné pravidlo
	`Statement ::= EmptyStatement` `EmptyStatement ::= <SEMICOLON>`

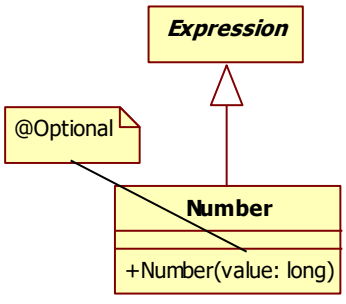
Tab. 9: Vzor použitia anotácie `@After` pre konštruktor

Diagram tried	Produkčné pravidlo

	`Statement ::= EmptyStatement`
	`| Block`
	`EmptyStatement ::= <SEMICOLON>`
	`Block ::= <LBR> (Statement)* <RBR>`

Tab. 10: Vzor použitia anotácie `@Before`, `@After` pre konštruktor

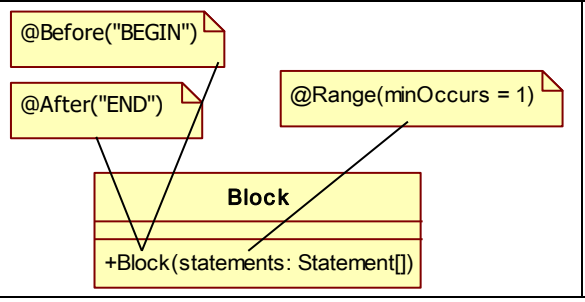
Príklad použitia anotácie `@Optional` vyjadrený diagramom tried aj s ekvivalentným produkčným pravidlom je uvedený v Tab. 11.

Diagram tried	Produkčné pravidlo
	`Expression ::= Number` `Number ::= <VALUE>?`

Tab. 11: Vzor použitia anotácie `@Optional`

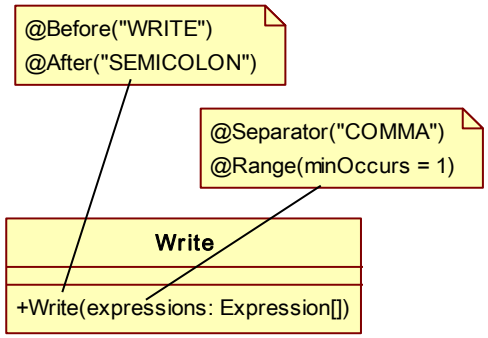
V tabuľkách Tab. 12 a Tab. 13 sú uvedené príklady použitia anotácií `@Range` a `@Separator`.

Diagram tried	Produkčné pravidlo

	`Block ::= <BEGIN>`
	`Statement+`
	`<END>`

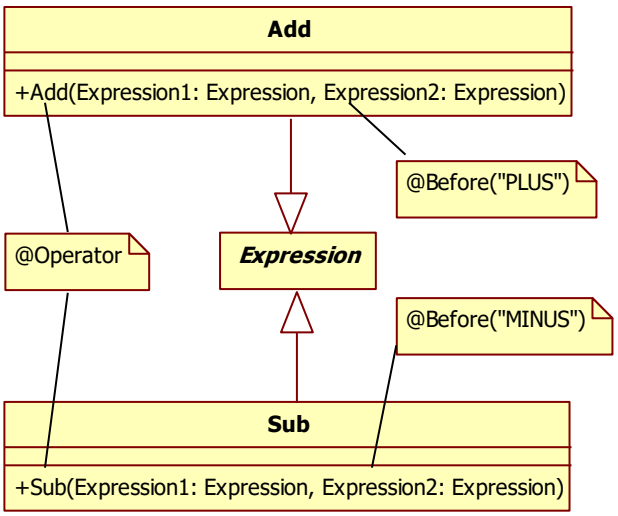
Tab. 12: Vzor použitia anotácie `@Range`

Diagram tried	Produkčné pravidlo

	`Write ::= <WRITE> Expression`
	`(<COMMA> Expression0)*`
	`<SEMICOLON>`

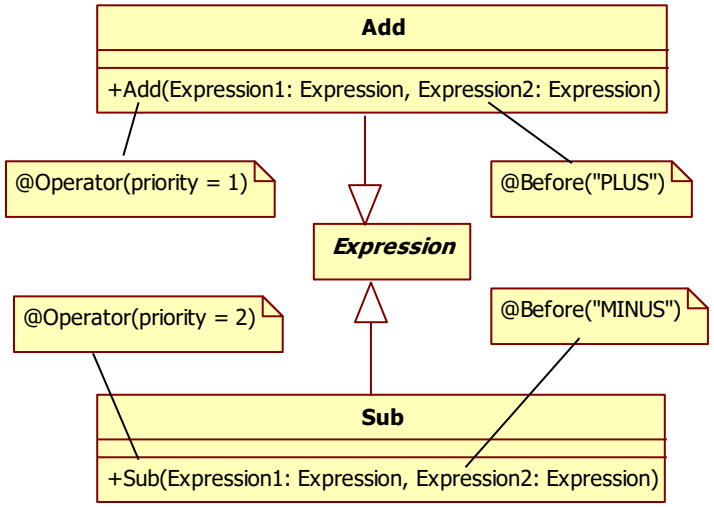
Tab. 13: Vzor použitia anotácie `@Range` spolu s anotáciou `@Separator`

Použitie anotácie `@Operator` je uvedené v tabuľkách Tab. 14, Tab. 15 a Tab. 16.

Diagram tried	

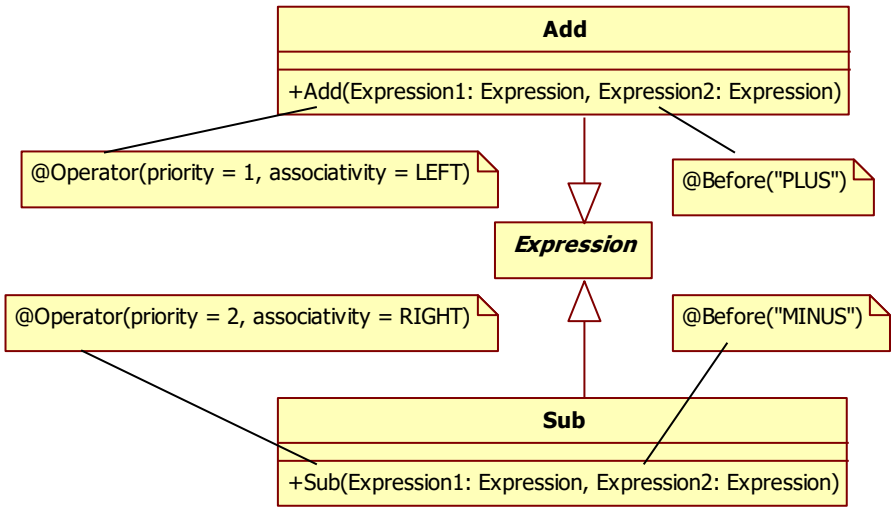
Produkčné pravidlo	`Expression1 ::= Expression`
	`((<PLUS> | <MINUS>) Expression)*`
	`Expression ::= ...`

Tab. 14: Vzor použitia anotácie `@Operator` I.

Diagram tried	

Produkčné pravidlo	`Expression1 ::= Expression2 (<PLUS> Expression2)*`
	`Expression2 ::= Expression (<MINUS> Expression)*`
	`Expression ::= ...`

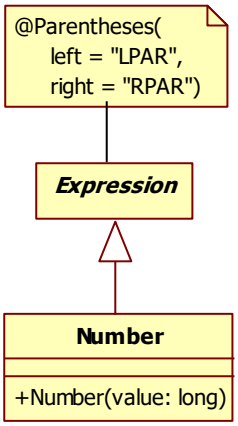
Tab. 15: Vzor použitia anotácie `@Operator` II.

Diagram tried	

Produkčné pravidlo	`Expression1 ::= Expression2 (<PLUS> Expression2)*`
	`Expression2 ::= Expression (<MINUS> Expression2)?`
	`Expression ::= ...`

Tab. 16: Vzor použitia anotácie `@Operator` III.

Posledný príklad uvedený v Tab. 17 obsahuje použitie anotácie `@Parentheses` spolu príslušným produkčným pravidlom.

Diagram tried	Produkčné pravidlo
	`Expression ::= Number | <LPAR> Expression <RPAR>` `Number ::= <VALUE>`

Tab. 17: Vzor použitia anotácie `@Parentheses`

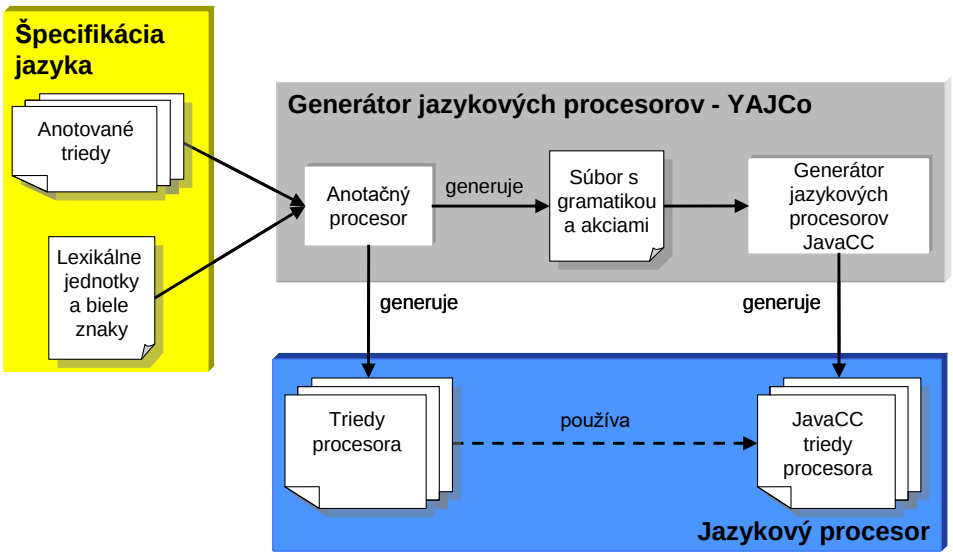
5.2.7. Generátor jazykových procesorov *YAJCo*

Na základe navrhnutej metódy špecifikácie jazyka a jazykového procesora bol implementovaný generátor jazykových procesorov *YAJCo* (Yet Another Java Compiler Compiler). Pre tento nástroj je doménový model zapísaný v jazyku Java a je označený anotáciami pre špecifikáciu konkrétnej syntaxe. Vytvorený nástroj prechádza určené zdrojové kódy a na základe ich štruktúry a anotácií generuje jazykový procesor. Zatiaľ čo anotácie ako aj doménový model sú nezávislé od technológie rozpoznania viet z jazyka, vygenerovaný jazykový procesor rozpoznáva vetu z jazyka konkrétnou technológiou. V prípade experimentálneho riešenia je použitý nástroj JavaCC, ktorý umožňuje určiť konkrétnu syntax bezkontextovou gramatikou LL(∞). Výhodou nezávislosti vytvoreného generátora jazykových procesorov je jednoduchý prechod na ľubovoľnú technológiu rozpoznania vety založenú na teórii bezkontextových gramatík [1][40][98][116] (SLR, LALR, LR, GLR) bez modifikácie anotovaného doménového modelu. V súčasnosti používané technológie rozpoznania viet poskytujú minimálnu možnosť migrácie medzi technológiami. Výhodou navrhnutého riešenia je možná voľba technológie rozpoznania viet jazyka na základe charakteru jazyka.

Vygenerovaný jazykový procesor rozpoznáva vety z jazyka a konštruuje abstraktný syntaktický strom. Skonštruovaný abstraktný syntaktický strom obsahuje uzly vo forme objektov doménových tried. Abstraktný syntaktický strom je konštruovaný od listov smerom ku koreňu. V čase konštruovania objektu uzla sú už skonštruované všetky objekty reprezentujúce podstromy tohto uzla.

Sémantické funkcie sú definované pre jednotlivé doménové triedy podľa uváženia dizajnéra jazyka a sú realizované priamo v jazyku Java bez použitia špeciálnej syntaxe ako obyčajné metódy objektov. V tomto riešení teda nie je nutné použiť špeciálnu syntax pre zápis sémantických funkcií.

Na Obr. 24 je zobrazená architektúra implementovaného generátora jazykových procesorov *YAJCo*. Vstupom generátora je doménový model jazyka zapísaný prostredníctvom anotovaných tried v jazyku Java. Výstupom generátora je množina tried a rozhraní v jazyku Java, ktoré reprezentujú vygenerovaný jazykový procesor.



Obr. 24: Implementovaný generátor jazykových procesorov

Vygenerovaný jazykový procesor spĺňa definíciu nasledujúceho polymorfného rozhrania v jazyku Java s typovým parametrom, ktorý je v konkrétnej implementácií stotožnený s typom hlavného (koreňového) uzla abstraktného syntaktického stromu.

```
public interface ParserInterface<T> {
  T parse(String text) throws ParseException;

  T parse(java.io.Reader reader) throws ParseException;
}
```

Jazykový procesor je reprezentovaný vygenerovanou triedou implementujúcou rozhranie `ParserInterface`. Vygenerovaná trieda sa používa na rozpoznávanie viet z jazyka zapísaných v textovej forme.

```
public class ExpressionParser implements ParserInterface<Expression> {
  public Expression parse(String text) throws ParseException {
    return parse(new java.io.StringReader(text));
  }

  public Expression parse(java.io.Reader reader) throws ParseException {
     //Použitie technológie rozpoznania viet
     //Konštrukcia abstraktného syntaktického grafu
  }
}
```

Vygenerovaný jazykový procesor je jednoducho použiteľný na spracovanie viet jazyka zo špecifikovaného vstupu. V nasledujúcom príklade je rozpoznaná veta `1 + 2 * 7` z jazyka aritmetických výrazov prostredníctvom jazykového procesora, pre ktorú je skonštruovaný abstraktný syntaktický strom s definovaným typom koreňového uzla `Expression`.

```
public class Main {
    public static void main(String[] args) throws Exception {
        String expr = "1 + 2 * 7";
        Expression expression = new ExpressionParser().parse(expr);
        System.out.println(expression.eval());
        System.out.println(expression.code());
```

```
        }
    }
```

Vygenerovaná konkrétna syntax jazyka aritmetických výrazov, ktorý je definovaný anotovaným doménovým modelom (časť 5.2.5) má nasledujúci tvar:

```
Expression1 ::= Expression2 (('-' | '+') Expression2)*
Expression2 ::= Expression3 (('*' | '/') Expression3)*
Expression3 ::= '-' Expression3 | Expression
Expression ::= Number | 'LPAR' Expression1 'RPAR'
Number ::= <VALUE>
```

Implementovaný nástroj pre generovanie jazykových procesorov bol konštruovaný s cieľom minimalizácie používania anotácií v doménovom modeli. Anotácie je nutné použiť len v prípade, keď nie je možné konkrétnu syntax jednoznačne odvodiť alebo definovať predvolenú stratégiu odvodenia. Vďaka takejto implementácii napríklad nie je nutné vždy používať anotácie `@Token`, `@Separator` alebo `@Range` ak to je možné odvodiť, respektíve určiť predvolené správanie generátora.

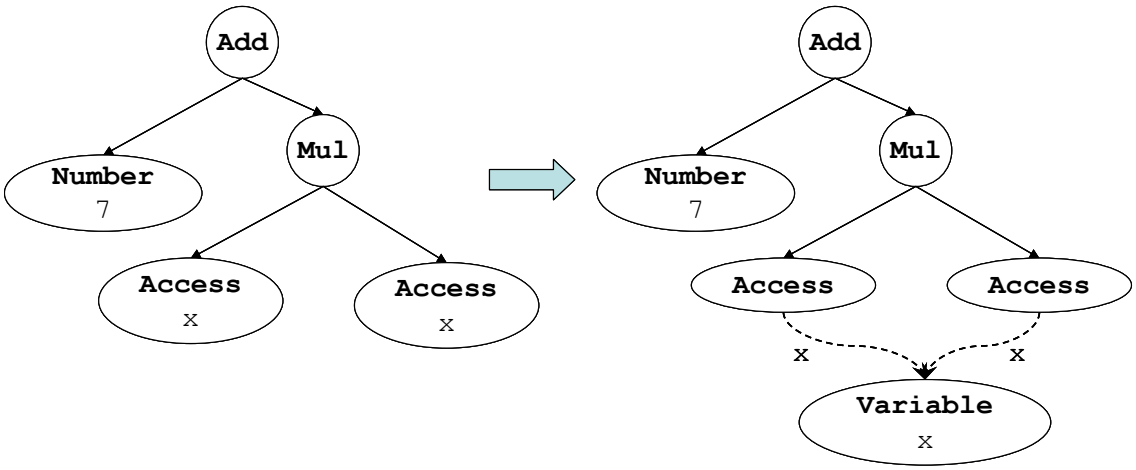
Zapisovať doménový model v programovacom jazyku Java nie je jediné možné riešenie. Táto možnosť bola zvolená na základe vysokej popularity jazyka Java ako aj množstva používateľov [13][98][99]. Navrhované riešenie definovania konkrétnej syntaxe je aplikovateľné pre ľubovoľný jazyk umožňujúci definovať doménový model, ktorý je rozšíriteľný o anotácie na požadovanej úrovni štruktúrovanosti. Výhodou voľby jazyka Java je existujúce množstvo nástrojov pre spracovanie viet jazykov ako aj podpora atribútového programovania v jazyku Java. Vďaka definícii doménového modelu v jazyku Java a anotáciám, ktoré umožňujú priamo v programe definovať metaúdaje je minimalizované duplikovanie kódov.

5.3. Transformácia na abstraktný syntaktický graf

V textovej forme jazykov sú vety jazyka zapísané ako postupnosť znakov, ktoré vytvárajú lexikálne symboly (slová). Vzhľadom na tento charakter textovej formy jazykov je nutné pre jazyky, v ktorých vety nie sú vyjadriteľné stromom ale všeobecným grafom, vytvárať textové odkazy reprezentujúce hrany v grafe. Tieto odkazy sú realizované vo forme použitia identifikátorov. Výskytom pojmov v jazyku (napr. premenné, funkcie) sú priradené mená (identifikátory), prostredníctvom ktorých je možné na inom mieste vo vete definovať odkaz na výskyt tohto pojmu. Referencie a identifikátory umožňujú zdieľanie výskytov pojmov vo vete. Identifikátor musí jednoznačne identifikovať daný výskyt pojmu. To ale neznamená, že rôzne výskyty aj tých istých pojmov musia mať rôzne identifikátory. Príkladom môže byť rovnaké meno rôznych premenných v dvoch rôznych procedúrach. Výskyt pojmu je teda identifikovaný nielen svojim menom ale aj kontextom, v ktorom sa vyskytuje. Iba globálne pojmy, ktoré majú oblasť platnosti v celej vete, musia mať jedinečné mená, pretože ich kontextom je samotná veta.

V uvedenom jazyku aritmetických výrazov je možné pre každú vetu jazyka vytvoriť abstraktný syntaktický strom. Vzhľadom na to, že v tomto jazyku nie sú použité identifikátory, je abstraktný syntaktický strom možné považovať za abstraktný syntaktický graf a transformácia na graf je teda identitou. Ak by bol jazyk aritmetických výrazov rozšírený o pojem premennej, ktorá bude používaná na viacerých miestach vo vete, bude nutné do jazyka pridať identifikátor konkrétneho výskytu pojmu a definovať aj samotný pojem premennej. Na Obr. 25 je vľavo zobrazený abstraktný syntaktický strom vety `7 + x * x` z jazyka aritmetických výrazov, ktorý je rozšírený o pojem premennej. Uzol

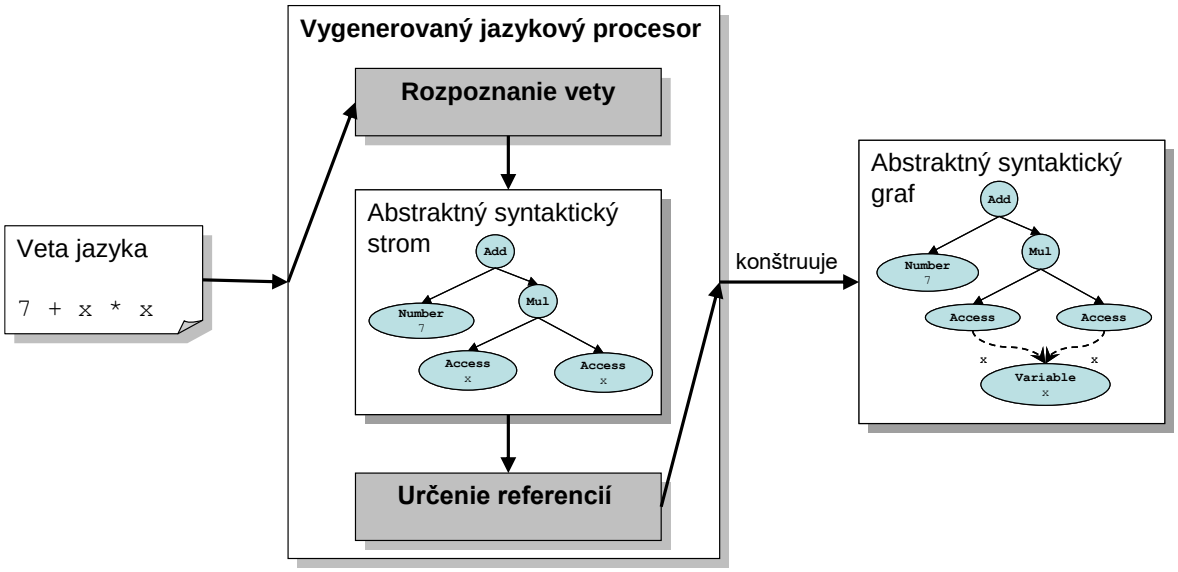
označený `Access` označuje prístup k premennej (hodnota premennej). Vpravo v obrázku je abstraktný syntaktický graf, ktorý vyjadruje použitie toho istého výskytu pojmu (premennej `x`) na základe referencií. Hrany reprezentujúce vytvorené referencie sú na obrázku zobrazené prerušovanou čiarou.



Obr. 25: Príklad transformácia abstraktného syntaktického stromu na graf

Vytvorenie dodatočných hrán nahrádzajúcich použitie identifikátorov na odkazovanie zjednodušuje následné spracovanie abstraktného syntaktického grafu a definovanie sémantiky.

Pre automatizované vytvorenie abstraktného syntaktického grafu z abstraktného syntaktického stromu bol navrhnutý a implementovaný nástroj realizujúci túto transformáciu, ktorý je súčasťou vygenerovaného jazykového procesora vytvoreného generátorom *YAJCo*. Tento nástroj je označený ako vyhľadávač referencií a kontext jeho použitia je zobrazený na Obr. 26.



Obr. 26: Určovanie referencií v jazykovom procesore

Pre potreby definovania identifikátorov a referencií boli do generátora pridané ďalšie dva anotačné typy. Anotačný typ `@Identifier` je určený na označenie

jednoznačného identifikátora uzla v abstraktnom syntaktickom strome. Zároveň umožňuje pre neglobálne pojmy definovať kontext, v ktorom má byť identifikátor jedinečný prostredníctvom hodnoty atribútu `unique` (XPath výraz [125]).

```
public @interface Identifier {
    String unique() default "";
}
```

Druhý anotačný typ `@References` umožňuje špecifikovať referencie na výskyty pojmov. Každý pojem je špecifikovaný doménovou triedou. Táto anotácia zároveň definuje spôsob uloženia referencie na objekt reprezentujúci uzol v abstraktnom syntaktickom strome.

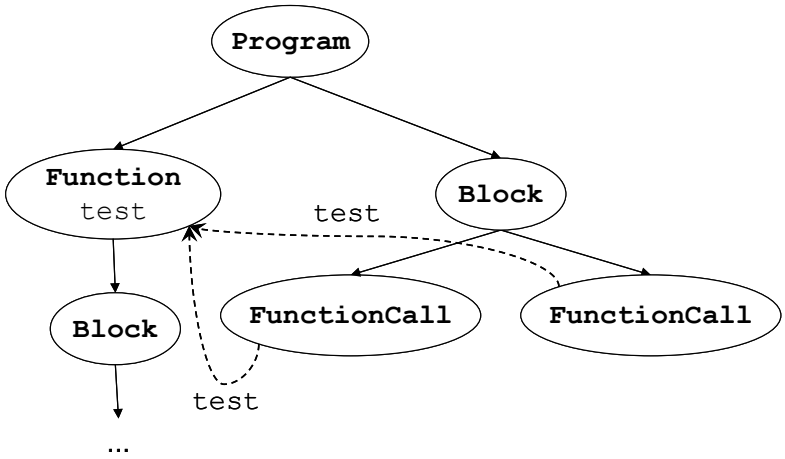
```
public @interface References {
    Class value();

    String field() default "";

    String path() default "";

    boolean create() default false;
}
```

Pre vytvorenie referencie je použitá reflexia [30] v jazyku Java. Požadovaný objekt reprezentujúci uzol v abstraktnom syntaktickom strome je uložený do objektu doménovej triedy, ktorá obsahovala anotáciu `@References`. Vyhľadanie a uloženie referencie je vykonané podľa vzoru injektáže závislostí [31] prostredníctvom `set` metódy. Pre globálne pojmy stačí uviesť triedu pojmu pre nájdenie (atribút `value`). V prípade, že by určenie objektovej premennej na uloženie referencie nebolo jednoznačné, je možné špecifikovať meno premennej, do ktorej sa má uložiť odkaz prostredníctvom atribútu `field`. V prípade neglobálnych pojmov je možné špecifikovať kontext vyhľadávania prostredníctvom XPath výrazu atribútom `path`. Atribút `create` anotácie `@References` zabezpečí vytvorenie výskytu špecifikovaného pojmu v prípade, že daný pojem ešte neexistuje.

Použitie anotácií na vytváranie abstraktného syntaktického grafu je možné prezentovať na príklade procedurálneho jazyka s procedúrami a funkciami (časť 7.1.3). Nasledujúci príklad je ukážkou vety z takéhoto jazyka. Táto veta definuje funkciu `test` s implicitným typom návratovej hodnoty a hlavné telo, ktoré túto funkciu dvakrát vykoná.

```
test() {
...
}

{
  test();
  test();
}
```

Vytvorený abstraktný syntaktický graf uvedenej vety je na Obr. 27. Prerušovanými čiarami sú naznačené vytvorené referencie na základe identifikátorov.



Obr. 27: Abstraktný syntaktický graf vety z procedurálneho jazyka

Doménová trieda `Function` definuje pojem funkcie, ktorý je identifikovateľný menom, čo je označené anotáciou `@Identifier`. Táto anotácia určuje premennú objektu `name` ako premennú v ktorej je uložený identifikátor výskytu pojmu.

```
public class Function {
    @Identifier
    private final String name;

    private final Parameter[] parameters;

    private final Block body;

    public Function(
            String ident,
            @Before("LPAR") @After("RPAR")
            @Separator("COMMA") Parameter[] parameters,
            Block body) {
        this.name = ident;
        this.parameters = parameters;
        this.body = body;
    }

    //...
}
```

Výraz volania funkcie je reprezentovaný doménovou triedou `FunctionCall`. Pri volaní funkcie (v uvedenom príklade `test()`) je menom funkcie určené, o ktorú funkciu sa jedná.

```
public class FunctionCall implements Expression {
    private Function function;

    private final Expression[] expressions;

    public FunctionCall(
            @References(Function.class) String ident,
            @Before("LPAR") @After("RPAR")
            @Separator("COMMA") Expression[] expressions) {
        this.expressions = expressions;
    }
    //...
```

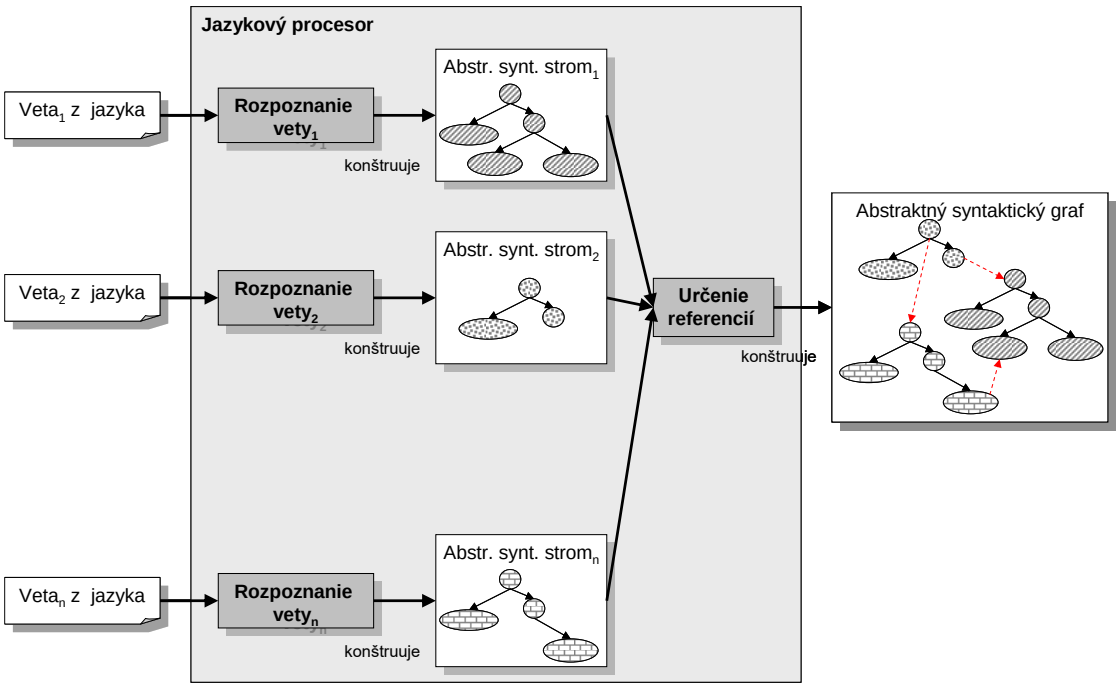
```
}
```

Prostredníctvom anotácie `@References` je označený v konštruktore triedy `FunctionCall` prvý parameter `ident`, ktorý obsahuje meno výskytu pojmu, na základe ktorého bude po vytvorení abstraktného syntaktického stromu nájdený príslušný výskyt pojmu doménovej triedy `Function`. Nájdený objekt bude uložený do premennej `function`, pretože táto premenná je premennou typu `Function`. Uvedený konštruktor neobsahuje uloženie referencie do premennej objektu `function` ani uloženie hodnoty parametra `ident`. Uloženie referencie do objektovej premennej `function` bude realizované vyhľadávačom referencií po skonštruovaní abstraktného syntaktického stromu automaticky na základe typu premennej.

Vyhľadávač referencií podporuje aj štandardný anotačný typ jazyka Java `@PostConstruct`. Anotácia slúži na označenie metódy, ktorá bude zavolaná vyhľadávačom referencií po nastavení všetkých požadovaných referencií v objekte doménovej triedy. Ukážka použitia tejto anotácie je uvedená v časti 7.1.5.

5.4. Kompozícia viet

Softvérový systém je zvyčajne špecifikovaný ako viacero viet z jedného alebo viacerých jazykov. Výsledný softvérový systém vzniká kompozíciou jednotlivých viet počítačových jazykov. Príkladom je kompozícia systému z tried zapísaných v súboroch v jazyku Java, kompozícia zo súborov modulov pre jazyk Haskell. Jednotlivé vety sú rozpoznávané jazykovým procesorom a následne komponované za účelom získania výsledného systému. Pri komponovaní sa používajú identifikátory určujúce konkrétne výskyty pojmov (napr. triedy, moduly). Navrhnutý a implementovaný generátor jazykových procesorov *YAJCo* umožňuje komponovať jednotlivé abstraktné syntaktické stromy do výsledného abstraktného syntaktického grafu práve prostredníctvom vyhľadávača referencií. Na Obr. 28 je zobrazený princíp kompozície viet z jazyka do výsledného abstraktného syntaktického grafu.



Obr. 28: Kompozícia viet z jazyka

5.5. Kompozícia jazykov

Jedna z výhod navrhnutého riešenia je modularizácia jazyka na základe abstraktnej syntaxe. Doménové triedy vyjadrujúce syntaktické oblasti a sémantické funkcie sú umiestnené do samostatných jednotiek, ktorými sú triedy v jazyku Java. Kompozícia jazykov definovaných prostredníctvom doménových modelov spočíva práve v kompozícii doménových modelov. Jazykový generátor pri generovaní jazykového procesora postupuje od určenej hlavnej triedy a prechádza všetky triedy, na ktoré existuje väzba. Väzba vzniká pre existenciu relácie špecializácie (relácia „je“) a relácie vlastníctva (relácia „má“). Prostredníctvom anotácií pre vyhľadávač referencií je určený ďalší typ väzby, ktorý je zohľadnený pri transformácií na abstraktný syntaktický graf.

Vytvorený generátor jazykových procesorov podporuje tri základné spôsoby kompozície jazykových konštrukcií definovaných jazykov:

1. *Vloženie jazyka* – kompozícia na úrovni konkrétnej syntaxe.
2. *Rozšírenie jazyka* – kompozícia na úrovni konkrétnej aj abstraktnej syntaxe.
3. *Jazyková referencia* – kompozícia na úrovni abstraktnej syntaxe.

Pri kompozícií jazykov je možné použiť súčasne viacero spôsobov kompozície.

5.5.1. Kompozícia jazykov vložením

Pri tomto spôsobe je vkladaný jazyk definovaný prostredníctvom anotovaného doménového modelu komponovaný s hostiteľským jazykom na úrovni konkrétnej syntaxe. To znamená, že vety z hostiteľského jazyka budú obsahovať časti (podvety), ktoré sú vetami jazyka vkladaného. Táto kompozícia je realizovaná na úrovni konkrétnej syntaxe, lebo jazyk vytvorený kompozíciou obsahuje konštrukcie konkrétnej syntaxe oboch jazykov. Vkladaný jazyk je nezávislý od hostiteľského jazyka a môže byť vyvíjaný nezávisle od hostiteľského jazyka. Príkladom je kompozícia jazyka pre výrazy (vkladaný jazyk) a imperatívneho jazyka (hostiteľský jazyk), kompozícia jazyka anotácií (vkladaný jazyk) a programovacieho jazyka Java (hostiteľský jazyk).

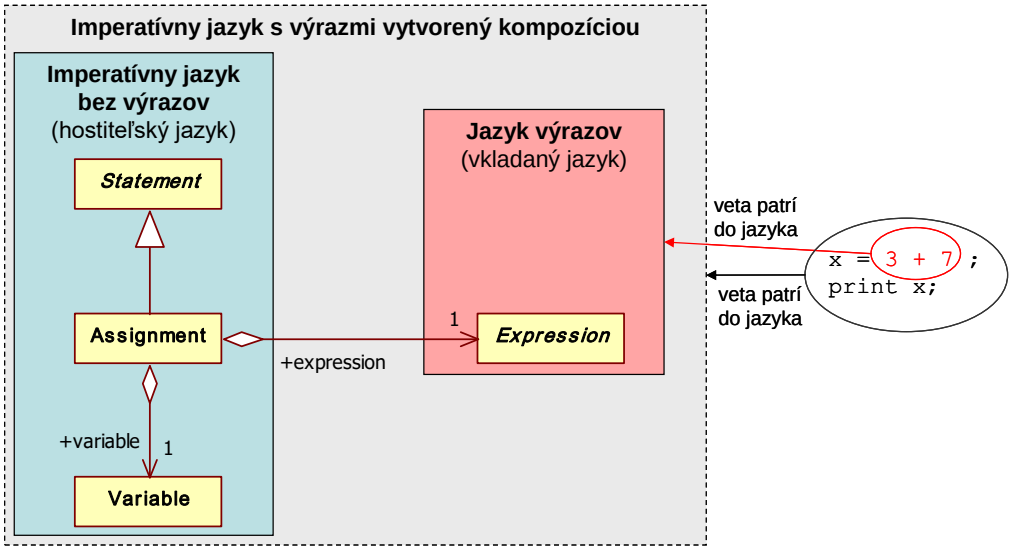
Kompozícia jazykov vložením je realizovaná v navrhnutom generátore jazykových procesorov vo forme zápisu parametrov v konštruktoroch. Aj keď vo väčšine prípadov väzba jazykov na úrovni konkrétnej syntaxe bude spätá s väzbou na úrovni abstraktnej, nie je to nevyhnutné. Generátor nepredurčuje väzbu jazykov komponovaných vložením na abstraktnej úrovni. Väzba abstraktnej syntaxe je na dizajnérovi hostiteľského jazyka, ktorý túto väzbu určuje v kontexte definície sémantiky jazyka.

Príklad kompozície vložením je zobrazený na Obr. 29. Do imperatívneho jazyka príkazov bez výrazov (hostiteľský jazyk) je vložený jazyk aritmetických výrazov (vkladaný jazyk).

Kompozícia vzniká na základe definovania väzby v konštruktore použitím doménovej triedy `Assignment` z jazyka aritmetických výrazov.

```
public class Assignment extends Statement {
    private Variable variable;

    private final Expression expression;

    @After("SEMICOLON")
    public Assignment(
            @References(Variable.class) String ident,
            @Before("ASSIGN") Expression expression) {
        this.expression = expression;
```

```
        }
        //...
}
```

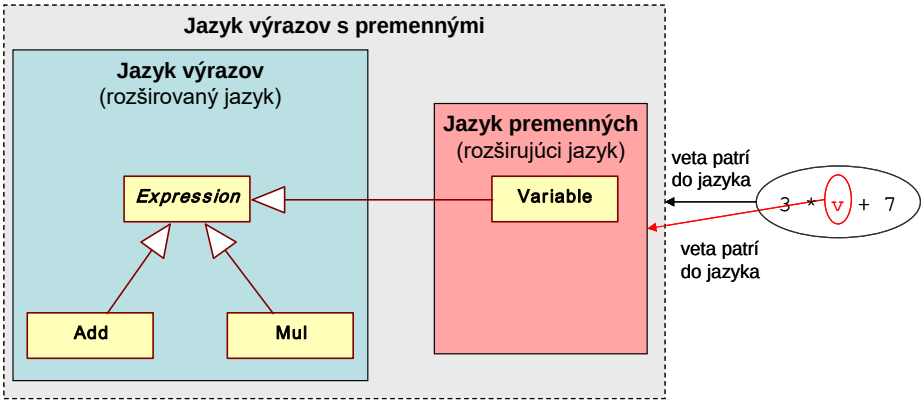


Obr. 29: Príklad kompozície jazykov vkladaním

V tomto prípade je väzba na úrovni konkrétnej syntaxe prenesená aj do väzby v abstraktnej syntaxi, pretože doménová trieda `Assignment` definuje premennú typu `Expression`, ktorej hodnota je určená v konštruktore na základe konkrétnej syntaxe.

5.5.2. Kompozícia rozširovaním jazyka

Pri tomto spôsobe kompozície je definovaný jazyk rozšírený o doménové triedy iného jazyka (rozširujúceho) dedením od doménových tried rozširovaného jazyka. Základným bodom rozšírenia je teda abstraktná alebo konkrétna doménová trieda. Vzhľadom na charakter spracovania doménového modelu generátorom *YAJCo*, tento spôsob kompozície jazykov má vplyv na konkrétnu aj abstraktnú syntax. Takéto rozšírenie umožňuje používať vety z rozširujúceho jazyka vo vetách rozširovaného jazyka. Príkladom môže byť rozšírenie jazyka aritmetických výrazov na jazyk aritmetických výrazov s premenenými (uvedené na Obr. 30).



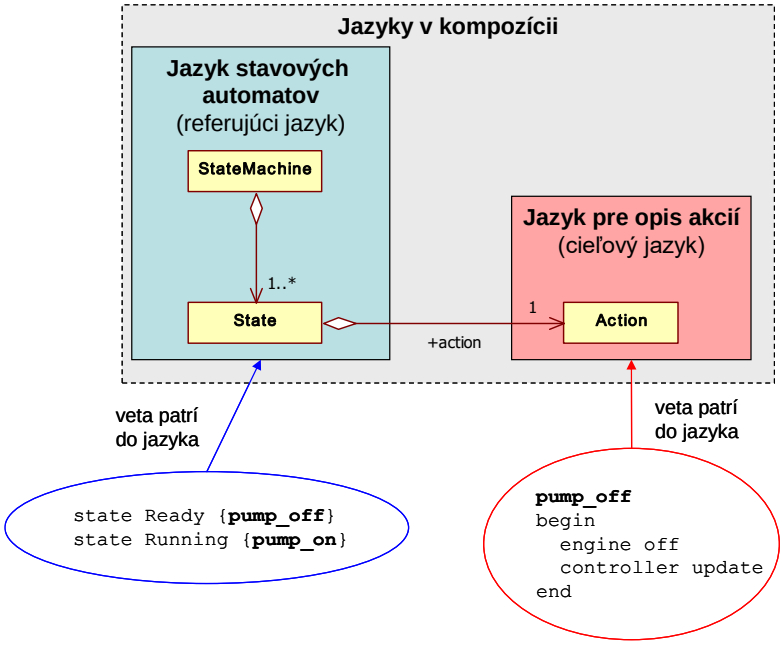
Obr. 30: Príklad kompozície jazykov rozširovaním

Kompozícia vzniká na základe definovania rozširovania doménovej triedy `Expression` triedou `Variable`. Pri takomto rozširovaní je rozširujúci jazyk viazaný na jazyk rozširovaný dedením od doménových tried rozširovaného jazyka. Generátor *YAJCo* umožňuje generovať pre oba jazyky samostatné jazykové procesory ale umožňuje vytvoriť aj jazykový procesor pre jazyk v kompozícii.

5.5.3. Kompozícia jazykov referenciou

Kompozícia prostredníctvom referencie je zovšeobecnením kompozície viet z jazyka opísanej v časti na úrovni jazykov. Pri tejto kompozícii sú komponované vety z rôznych jazykov vyhľadávačom referencií za vzniku abstraktného syntaktického grafu. Táto kompozícia je kompozíciou len na úrovni abstraktnej syntaxe. Veta z jazyka v kompozícii v tomto prípade nie je podvetou iného jazyka v kompozícii. Komponovanie je realizované na základe mien identifikátorov. Jeden jazyk v kompozícii (referujúci jazyk) obsahuje odkaz na konštrukciu jazyka (cieľový jazyk) na základe identifikátora. Vytvorený generátor *YAJCo* nepožaduje aby bol zápis identifikátora v referujúcom jazyku rovnaký ako v jazyku cieľovom, no musí existovať transformačná funkcia na transformáciu zápisu identifikátora z referujúceho jazyka do jazyka cieľového.

Príkladom takejto kompozície je kompozícia jazyka stavových automatov a jazyka opisu činností akcií uvedená na Obr. 31. Pri prechode automatu do stavu má byť vykonaná akcia, ktorá je identifikovaná menom.



Obr. 31: Príklad kompozície jazykov referenciou

5.6. Zhrnutie

Vytvorená metóda pre špecifikáciu jazykov a implementáciu jazykových procesorov spája výhody oboch prístupov: opis jazyka prostredníctvom bezkontextových gramatík aj metamodelov. Špecifikácia konkrétnej syntaxe jazyka je nezávislá od konkrétnej technológie spracovania vety bez vplyvu na samotnú efektivitu rozpoznania vety. V tejto metóde je kľúčová abstraktná syntax a sémantika jazyka. Konkrétna syntax

jazyka je v tejto metóde určená na základe abstraktnej syntaxe umožňujúc vytvárať viacero foriem konkrétnej syntaxe pre jeden jazyk. Navyše na základe charakteru generovanej gramatiky pre konkrétnu syntax je možné odporúčať zmeny v konkrétnej syntaxi dizajnérovi jazyka s cieľom využitia konkrétneho spôsobu spracovania vety. Na druhej strane je v tejto metóde možné použiť bežné nástroje a prostriedky implementácie softvérových systémov založené na použití objektových princípov vrátane diagramov tried. Doménovo-špecifický jazyk pre definovanie konkrétnej syntaxe je jednoduchý z pohľadu počtu konštrukcií. Obsahuje len 8 anotácií pre určenie štruktúry a väzby lexikálnych symbolov na jazykové konštrukcie. Predpokladom použitia tohto jazyka je znalosť objektovo-orientovaného programovania a základných princípov atribútovo-orientovaného programovania. Vďaka použitiu konštruktorov ostáva dostatočná voľnosť pre špecifikáciu konkrétneho zápisu jazykovej konštrukcie. Za pozitívnu vlastnosť navrhnutej metódy je možné považovať aj jednoduchú komponovateľnosť jazykov na syntaktickej úrovni ako aj separáciu jazykových elementov na základe pojmov jazyka.

6 Prípadová štúdia – jazyk didaktických testov

V tejto kapitole si ukážeme prípadovú štúdiu návrhu doménovo-špecifického jazyka pre oblasť didaktických textov. Našim cieľom bude navrhnúť jazyk pre túto doménu a implementovať nástrojovú podporu pre tvorbu a použitie viet v tomto jazyku.

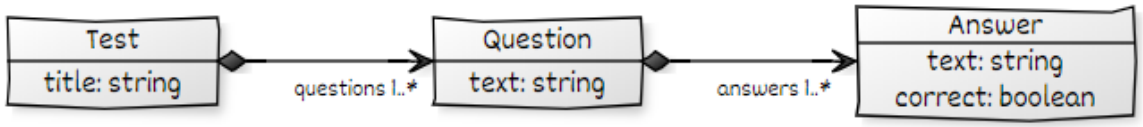
6.1. Abstraktná syntax

Pri tvorbe jazyka začíname doménovou analýzou, jednoducho povedané pochopením domény. Bez pochopenia domény a porozumenia expertom v doméne sú šance na vytvorenie aspoň uspokojujúceho jazyka minimálne. Zamýšľame sa nad kľúčovými pojmami v doméne a ich použitím, pýtame sa na vlastnosti, činnosti a snažíme sa definovať rozsah jazyka. Nápomocné v tejto fáze nám sú rozhovory s doménovými expertmi, ale aj existujúca doménová literatúra (napr. postupy, návody, príručky) a taktiež dokumenty používané na komunikáciu v doméne (napr. tlačivá).

6.1.1. Doménový model

Pri realizácii doménovej analýzy identifikujeme pojmy a vzťahy a vytvárame tak model domény. Proces tvorby doménového modelu a jeho dôležitosť by sme mohli prirovnať k procesu tvorby údajového modelu pri vývoji informačných systémoch.

Poďme sa teraz pozrieť na doménu didaktických testov a pokúsme sa identifikovať kľúčové pojmy a vzťahy medzi týmito pojmami. Ako už názov domény hovorí, kľúčový pojem bude *test*. Existuje viacero druhov didaktických testov. Kvôli jednoduchosti sa sústreďme na didaktické testy s otázkami s uzavretou odpoveďou. V takýchto testoch testovaný pri otázke volí jednu alebo viac odpovedí, o ktorých sa domnieva, že sú správnymi odpoveďami na položenú otázku. V predošlých vetách teda môže identifikovať hneď niekoľko ďalších kľúčových pojmov pre doménu, ktorými sú *otázka* a *odpoveď* a zároveň aj niekoľko vzťahov, napríklad, že test obsahuje otázky (je zložený z otázok) a otázka obsahuje odpovede (je zložená z odpovedí). Každá otázka aj odpoveď má svoje znenie - text. Okrem toho, je ešte odpoveď možno označiť ako správnu pre danú otázku. Aby sme vedeli testy rozlíšiť, bude tiež vhodné aby mal test svoj názov. Nasledujúci obrázok je modelom našej domény. Na vyjadrenie sme využili UML diagram tried, v našom prípade je to teda metamodelovací jazyk (jazyk na vyjadrenie modelu). Tento jazyk zároveň reprezentuje abstraktnú syntax jazyka.



Obr. 32: Doménový model pre didaktický test

Pre potreby implementácie nástrojovej podpory pre tvorbu a použite viet z jazyka didaktických testov, zapíšme tento model v jazyku Java. Jednotlivé pojmy budeme reprezentovať prostredníctvom tried a ich vlastností, tak ako sme to robili v predošlej kapitole. Toto vyjadrenie pojmov v jazyku Java budeme potom ďalej používať napríklad pri validácii, definícii konkrétnej syntaxe a generovaní/interpretovaní. Jednotlivé objekty z takto definovaných tried budú tvoriť vrcholy abstraktného syntaktického grafu vety z jazyka.

```
public class Test {
    private String title;

    private Question[] questions;

    public Test() {}

    public Test(String title, Question[] questions) {
        this.title = title;
        this.questions = questions;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    //Ďalšie get a set metódy ...
}

public class Question {
    private String text;

    private Answer[] answers;

    public Question() {}

    public Question(String text, Answer[] answers) {
        this.text = text;
        this.answers = answers;
    }

    public String getText() {
        return text;
    }

    //Get a set metódy ...
}

public class Answer {
    private String text;

    private boolean correct;

    public Answer() {}

    public Answer(String text, boolean correct) {
        this.text = text;
        this.correct = correct;
    }

    //Get a set metódy ...
}
```

6.1.2. Validácia

V jazyku Java sme prostredníctvom tried definovali doménový model. Takáto definícia nám ale nezaručí, že vytvárané vety budú korektné v zmysle obmedzení vytváraného doménovo-špecifického jazyka. Pokúsme sa identifikovať nejaké zmysluplné obmedzenia pre vety z jazyka. Bolo by vhodné aby každá veta z jazyka, čo je v našom prípade jeden test, mala aspoň jednu otázku. Veď aký zmysel by mal test bez otázok? Taktiež by bolo vhodné, aby každá otázka mala aspoň 2 odpovede, aby bolo z čoho vyberať. Zo samotných odpovedí by mala byť správna v otázke aspoň jedna odpoveď.

Keďže ako metamodelovací jazyk používame v našom prípade jazyka Java, môže jednoducho definovať validáciu vety prostredníctvom metód a to buď priamo v doménových triedach alebo mimo nich, napr. použitím známeho objektového vzoru Návštevník (angl. Visitor). V nasledujúcom zdrojovom kóde validáciu implementujeme priamo v metódach doménových tried s názvom `validate`.

```
public class Test {
    //Vlastnosti, konštruktory, get a set metódy ...

    public void validate() {
        if(questions == null || questions.length == 0)
            throw new TestLanguageException("Test musí mať aspoň otázku");

        for(Question question : questions)
            question.validate();
    }
}

public class Question {
    //Vlastnosti, konštruktory, get a set metódy ...

    public void validate() {
        if (answers == null || answers.length < 2) {
            throw new TestLanguageException(
                        "Otázka musí mať aspoň dve odpovede");
        }

        if (Arrays.stream(answers).allMatch(a -> !a.isCorrect())) {
            throw new TestLanguageException(
                        "Otázka musí mať aspoň jednu správnu odpoveď");
        }
    }
}
```

6.2. Generovanie a interpretácia

Pre náš doménovo-špecifický jazyk didaktických testov musíme definovať aj sémantiku. Sémantiku v jazykových procesoroch najčastejšie definujeme generatívnym alebo interpretačným prístupom. Pri generovaní sa z vety v doménovo-špecifickom jazyku na vstupe vygeneruje ako výstup veta v inom jazyku. Pri interpretácii sa veta zo vstupu priamo interpretuje (vyhodnocuje), dôsledkom čoho je vykonanie akcií. To, či je pre jazyk definovaná sémantika interpretovaním alebo generovaním, nepredurčuje jeho abstraktná ani konkrétna syntax, ale rozhodnutie autora nástrojovej podpory pre jazyk. Navyše, samotné rozlíšenie toho či sa jedná o interpretáciu alebo generovanie nemusí byť pri

doménovo-špecifických jazykoch jednoznačné. V našom jazyku môžeme sémantiku definovať rôzne podľa okolností použitia. Jeden zo spôsobov môže byť použitie generovania, ktoré na základe zapísanej vety v jazyku didaktických testov vygeneruje test v HTML podobe, ktorý môžeme študentov vytlačiť. Iným prístupom môže byť interpretácia, pri ktorej bude študent priamo skúšaný na základe zapísaného didaktického testu.

Poďme si teda ukázať príklad zapísania sémantiky generovaním, v ktorom použijeme metódy v jazyku Java na zápis transformácie. Tieto metódy opisujú ako sa transformuje vstupná veta vyjadrená abstraktným syntaktickým grafom postupne po pojmoch na výstup, ktorý reprezentuje postupnosť znakov. Táto postupnosť znakov reprezentuje výstupnú vetu v jazyku HTML.

```
public class TestGeneratorJava {
    public void generate(Test test, Writer writer) {
        PrintWriter out = new PrintWriter(writer);
        out.println("<html>");
        out.printf("<head><title>%s</title></head>\n", test.getTitle());
        out.println("<body>");

        out.printf("<h1>Test: %s</h1>\n", test.getTitle());
        out.println("<ol>");
        for (Question question : test.getQuestions())
            generate(question, out);
        out.println("</ol>");

        out.println("</body>");
        out.println("</html>");
    }

    private void generate(Question question, PrintWriter out) {
        out.println("<li class='question'>" + question.getText());

        out.println("<ol type='A'>");
        for (Answer answer : question.getAnswers())
            generate(answer, out);

        out.println("</ol>");
    }

    private void generate(Answer answer, PrintWriter out) {
        String style = answer.isCorrect() ? "correct_answer" : "wrong_answer";
        out.printf("<li class='%s'>%s\n", style, answer.getText());
    }
}
```

Iným spôsobom implementácie generovania je použitie šablónovacích systémov. Tieto systémy sú vhodné najmä vtedy, ak je veľká časť výstupnej vety nemenná v závislosti od vstupnej vety. Navyše je v šablóne zvyčajne ľahšie identifikovateľná výstupná veta, čo nám umožňuje pozerať sa na riešenie z pohľadu výsledku. Príklad takéhoto generovania si uvedieme s použitím šablónovacieho nástroja Velocity (https://velocity.apache.org). Pre našu prípadovú štúdiu didaktických testov by mohla vyzerať šablóna v jazyku Velocity generujúca test do jazyka HTML nasledovne.

```
<html>
    <head><title>$test.title</title></head>
<body>
<h1>Test: $test.title</h1>

<ol>
    #foreach($question in $test.questions)
        <li class='question'>$question.text</li>
        <ol type='A'>
            #foreach($answer in $question.answers)
                <li
class='#if($answer.correct)correct_answer#{else}wrong_answer#end'>
                 $answer.text
            #end
        </ol>
    #end
</ol>
</body>
</html>
```

Samotný nástroj Velocity môžeme spolu so šablónou použiť nasledovne. Po voľbe šablóny, ktorú použijeme na spracovanie, pripravíme kontext pre transformáciu, ktorý je v našom prípade objekt testu. Následne transformáciu vykonáme.

```
void generate(Test test, Writer writer) throws IOException {
    //Zvoľme šablónu
    Template template = Velocity.getTemplate("templates/test.html.vm");

    //Pripravme parametre pre šablónu - objekt testu
    Map<String, Object> params = new HashMap<>();
    params.put("test", test);

    //Vykonajme transformáciu s použitím zvolenej šablóny
    VelocityContext context = new VelocityContext(params);
    template.merge(context, writer);
}
```

6.3. Konkrétna syntax v hostiteľskom jazyku Java

Aby bolo možné zapisovať respektíve spracovať vety v jazyku didaktických testov, musíme pre tieto činnosti implementovať nástroje. Najprv sa vyberieme cestou interných doménovo-špecifických jazykov, pri ktorých sa snažíme použiť existujúce prostredie a vytvorené nástroje pre takzvané hostiteľské jazyky. Motiváciou pre tento prístup je ušetriť náklady znovupoužitím už vytvorených nástrojov. Prvým krokov je výber hostiteľského jazyka a s ním spätých nástrojov. Na celkovú efektivitu tvorby viet v internom doménovo-špecifickom jazyku totiž nemá význam len hostiteľský jazyk ale aj nástroje, ktoré boli preň vytvorené. Pri tomto prístupe to ale bude o kompromisoch medzi predstavou autora o ideálnom zápise a tým čo nám umožní hostiteľský jazyk. V príkladoch sme zvolili ako hostiteľský jazyk programovací jazyk Java. Možno to nie je ideálna voľba z pohľadu jazyka samotného vzhľadom na jeho konkrétnu syntax (zátvorky, bodkočiarky, povinné časti), no v prospech jazyka Java hovorí množstvo programátorov, ktorí tento jazyk ovládajú, ako aj množstvo veľmi kvalitných nástrojov na podporu tvorby viet v jazyku Java.

V rámci nasledujúcich častí si ukážeme viacero riešení (vzorov) implementácie interných doménovo-špecifických jazykov pri implementáciu nášho jazyka didaktických testov.

6.3.1. Vzor postupnosť funkcií

Prvým prezentovaným vzorom implementácie interných doménovo-špecifických jazykov je vzor postupnosť funkcií (angl. function sequence). Tento vzor prezentuje nasledujúca ukážka, v ktorej je veta zapísaná ako postupnosť volaní definovaných funkcií.

```
test("Astronomia");

question("Kolko planet ma Slnecna sustava?");
    wrong_answer("6");
    wrong_answer("7");
    correct_answer("8");
    wrong_answer("9");

question("Zem je v poradi od Slnka");
    wrong_answer("2");
    wrong_answer("4");
    correct_answer("3");
    wrong_answer("5");

question("Mars je v poradi od Slnka");
    wrong_answer("6");
    wrong_answer("7");
    correct_answer("4");
    wrong_answer("9");
```

Takúto formu zápisu nám jazyk Java umožňuje použiť vďaka statickým importom. Pre lepšie pochopenie príkladu uvádzame aj zdrojový kód triedy `FunctionSequenceBuilder`, ktorého metódy sú použité v predošlom príklade.

```
public class FunctionSequenceBuilder {
    private static String testTitle;

    private static String questionText;

    private static List<Question> questions = new ArrayList<>();

    private static List<Answer> answers;

    public static void test(String title) {
        testTitle = title;
    }

    public static void question(String text) {
        if(questionText != null) {
            createQuestion();
        }
        answers = new ArrayList<>();
        questionText = text;
    }
```

```
    public static void correct_answer(String text) {
        answers.add(new Answer(text, true));
    }

    public static void wrong_answer(String text) {
        answers.add(new Answer(text, false));
    }

    public static Test getTest() {
        createQuestion();
        return new Test(testTitle, questions.toArray(new Question[] {}));
    }

    private static void createQuestion() {
        questions.add(new Question(questionText,
                answers.toArray(new Answer[]{})));
    }
}
```

6.3.2. Vzor vnorená funkcia

Druhým prezentovaným vzorom implementácie interných doménovo-špecifických jazykov je vzor vnorená funkcia (angl. nested function). Tento vzor prezentuje nasledujúca ukážka, v ktorej je veta zapísaná ako postupnosť volaní vnorených funkcií.

```
test("Astronomia",
        question("Kolko planet ma Slnecna sustava?",
                wrong_answer("6"),
                wrong_answer("7"),
                correct_answer("8"),
                wrong_answer("9")
        ),

        question("Zem je v poradi od Slnka",
                wrong_answer("2"),
                wrong_answer("4"),
                correct_answer("3"),
                wrong_answer("5")
        ),

        question("Mars je v poradi od Slnka",
                wrong_answer("6"),
                wrong_answer("7"),
                correct_answer("4"),
                wrong_answer("9")
        )
);
```

Pre lepšie pochopenie príkladu uvádzame aj zdrojový kód triedy `NestedFunctionBuilder`, ktorého metódy sú použitý v predošlom príklade.

```
public class NestedFunctionBuilder {
    private static Test test;

    public static void test(String title, Question... questions) {
        test = new Test(title, questions);
```

```
    }

    public static Question question(String text, Answer... answers) {
        return new Question(text, answers);
    }

    public static Answer wrong_answer(String text) {
        return new Answer(text, false);
    }

    public static Answer correct_answer(String text) {
        return new Answer(text, true);
    }

    public static Test getTest() {
        return test;
    }
}
```

6.3.3. Vzor zreťazenie metód

Tretím vzorom implementácie interných doménovo-špecifických jazykov, na ktorý sa pozrieme, je vzor zreťazenie metód (angl. method chaining). Tento vzor prezentuje nasledujúca ukážka, v ktorej je veta zapísaná ako zreťazenie volania metód.

```
Test test = ((MethodChainingBuilder)

test("Astronomia").

question("Kolko planet ma Slnecna sustava?").
    correct_answer("8").
    wrong_answer("6").
    wrong_answer("7").
    wrong_answer("9").

question("Zem je v poradi od Slnka").
    correct_answer("3").
    wrong_answer("2").
    wrong_answer("4").
    wrong_answer("5").

question("Mars je v poradi od Slnka").
    correct_answer("4").
    wrong_answer("6").
    wrong_answer("7").
    wrong_answer("9")

).getTest();
```

Zdrojový kód triedy `MethodChainingBuilder` a príslušných implementujúcich rozhraní je uvedený na nasledujúcich riadkoch.

```
public interface QuestionInterface {
    FirstAnswerInterface question(String text);
}
```

```
public interface FirstAnswerInterface {
    SecondAnswerInterface correct_answer(String text);
}

public interface SecondAnswerInterface {
    FirstAnswerInterface question(String text);

    SecondAnswerInterface correct_answer(String text);

    SecondAnswerInterface wrong_answer(String text);
}

public class MethodChainingBuilder
 implements QuestionInterface, SecondAnswerInterface, FirstAnswerInterface {
    private static String testTitle;

    private static String questionText;

    private static List<Question> questions = new ArrayList<>();

    private static List<Answer> answers;

    public static QuestionInterface test(String title) {
        MethodChainingBuilder builder = new MethodChainingBuilder();
        builder.testTitle = title;
        return builder;
    }

    public FirstAnswerInterface question(String text) {
        if(questionText != null) {
            createQuestion();
        }
        answers = new ArrayList<>();
        questionText = text;
        return this;
    }

    public MethodChainingBuilder correct_answer(String text) {
        answers.add(new Answer(text, true));
        return this;
    }

    public MethodChainingBuilder wrong_answer(String text) {
        answers.add(new Answer(text, false));
        return this;
    }

    public Test getTest() {
        createQuestion();
        return new Test(testTitle, questions.toArray(new Question[] {}));
    }

    private static void createQuestion() {
        questions.add(new Question(questionText,
                answers.toArray(new Answer[]{})));
    }
}
```

6.4. Konkrétna syntax v jazykoch XML, JSON a YAML

Alternatívou k použitiu hostiteľského jazyka Java sú jazyky XML, JSON a YAML, ktorých použitie na definíciu konkrétnej syntaxe doménovo-špecifického jazyka si uvedieme v nasledujúcich príkladoch. Prvý príklad je jedným z možných zápisov viet nášho jazyka didaktických testov v jazyku XML.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<test>
    <title>Astronomia</title>
    <question>
        <text>Kolko planet ma Slnecna sustava?</text>
        <answer correct="false">
            <text>6</text>
        </answer>
        <answer correct="false">
            <text>7</text>
        </answer>
        <answer correct="true">
            <text>8</text>
        </answer>
        <answer correct="false">
            <text>9</text>
        </answer>
    </question>
    <question>
        <text>Zem je v poradi od Slnka</text>
        <answer correct="false">
            <text>2</text>
        </answer>
        <answer correct="false">
            <text>4</text>
        </answer>
        <answer correct="true">
            <text>3</text>
        </answer>
        <answer correct="false">
            <text>5</text>
        </answer>
    </question>
    <question>
        <text>Mars je v poradi od Slnka</text>
        <answer correct="false">
            <text>6</text>
        </answer>
        <answer correct="false">
            <text>7</text>
        </answer>
        <answer correct="true">
            <text>4</text>
        </answer>
        <answer correct="false">
            <text>9</text>
        </answer>
    </question>
</test>
```

Na spracovanie XML v jazyku Java môžeme použiť viacero technológií, uveďme si použitie technológie JAXB, ktorá je súčasťou platformy Java.

```
public class XMLSerializer {
    public Test load(Reader reader) throws JAXBException {
        JAXBContext jc = JAXBContext.newInstance("sk.tuke.dsl.testlang.model");
        Unmarshaller unmarshaller = jc.createUnmarshaller();
        return (Test) unmarshaller.unmarshal(reader);
    }

    public void save(Writer writer, Test test) throws JAXBException {
        JAXBContext jc = JAXBContext.newInstance("sk.tuke.dsl.testlang.model");
        Marshaller marshaller = jc.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
        marshaller.marshal(test, writer);
    }
}
```

Aby táto technológia fungovala správne, musíme označiť doménové triedy `Test`, `Question` a `Answer` anotáciami technológie JAXB `@XmlRootElement` a `@XmlElement`. Predpokladáme, že doménové triedy sú umiestnené v Java balíku `sk.tuke.dsl.testlang.model`.

```
@XmlRootElement(namespace = "http://kpi.fei.tuke.sk/testlang")
public class Test {
    //...

    @XmlElement(name = "question")
    public Question[] getQuestions() {
        return questions;
    }
}

public class Question implements Serializable {
    //...

    @XmlElement(name = "answer")
    public Answer[] getAnswers() {
        return answers;
    }
}
```

V nasledujúcom príklade je ukázané použitie formátu JSON na zápis viet doménovo-špecifických jazykov, ktorý plní rovnakú úlohu na zápis konkrétnej syntaxe ako jazyk XML. Porovnanie ich rozdielov nechávame na čitateľa.

```
{
  "title": "Astronomia",
  "questions": [
    {
      "text": "Kolko planet ma Slnecna sustava?",
      "answers": [
        { "text": "6",
          "correct": false
```

```
        },
        { "text": "7",
          "correct": false
        },
        { "text": "8",
          "correct": true
        },
        { "text": "9",
          "correct": false
        }
      ]
    },
    { "text": "Zem je v poradi od Slnka",
      "answers": [
        { "text": "2",
          "correct": false
        },
        { "text": "4",
          "correct": false
        },
        { "text": "3",
          "correct": true
        },
        { "text": "5",
          "correct": false
        }
      ]
    },
    { "text": "Mars je v poradi od Slnka",
      "answers": [
        { "text": "6",
          "correct": false
        },
        { "text": "7",
          "correct": false
        },
        { "text": "4",
          "correct": true
        },
        { "text": "9",
          "correct": false
        }
      ]
    }
  ]
}
```

Na spracovanie formátov JSON a YAML v jazyku Java môžeme použiť napríklad knižnicu Jackson (https://github.com/FasterXML/jackson). Nasledujúci zdrojový kód ukazuje ako načítať a zapísať objekty z/do súboru formátu JSON.

```
public class JSONSerializer {
    public Test load(Reader reader) throws IOException {
        ObjectMapper objectMapper = new ObjectMapper();
        return objectMapper.readValue(reader, Test.class);
    }
```

```
    public void save(Writer writer, Test test) throws IOException {
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.writeValue(writer, test);
    }
}
```

V poslednom príklade ukazujeme použitie formátu YAML na zápis viet doménovo-špecifických jazykov.

```
title: "Astronomia"
questions:
  - text: "Kolko planet ma Slnecna sustava?"
    answers:
      - text: "6"
        correct: false
      - text: "7"
        correct: false
      - text: "8"
        correct: true
      - text: "9"
        correct: false
  - text: "Zem je v poradi od Slnka"
    answers:
      - text: "2"
        correct: false
      - text: "4"
        correct: false
      - text: "3"
        correct: true
      - text: "5"
        correct: false
  - text: "Mars je v poradi od Slnka"
    answers:
      - text: "6"
        correct: false
      - text: "7"
        correct: false
      - text: "4"
        correct: true
      - text: "9"
        correct: false
```

Nasledujúci zdrojový kód ukazuje ako načítať a zapísať objekty z/do súboru formátu YAML a to opäť použitím technológie Jackson.

```
public class YAMLSerializer {
    public Test load(Reader reader) throws IOException {
        ObjectMapper objectMapper = new ObjectMapper(new YAMLFactory());
        return objectMapper.readValue(reader, Test.class);
    }

    public void save(Writer writer, Test test) throws IOException {
        ObjectMapper objectMapper = new ObjectMapper(new YAMLFactory());
        objectMapper.writeValue(writer, test);
```

```
    }
}
```

6.5. Konkrétna syntax bez obmedzení

Doteraz sme prechádzali možnosti pre definíciu konkrétnej syntaxe, ktoré využívali existujúce hostiteľské jazyky a ich prostriedky. Na jednej strane sa nám podarilo využiť existujúce jazykové nástroje, no na druhej strane sme boli obmedzený syntaxou hostiteľského jazyka. Takto definovaná konkrétna syntax obsahuje nežiadúce elementy, tzv. syntaktický šum. Teraz sa pozrieme na definíciu konkrétnej syntaxe, ktorá nebude závislá od žiadneho jazyka a bude plne zodpovedať našej predstave. V príkladoch budeme používať generátor jazykových procesorov ANTLR (https://www.antlr.org). V nasledujúcom texte zapisujeme ukážku vety pre náš jazyk didaktických testov.

```
"Test z astronomie"

question "Ktora planeta je obyvana?" {
    - "Merkur",
    - "Venusa",
    + "Zem",
    - "Jupiter"
}

question "Kolko planet ma Slnecna sustava?" {
    - "6",
    - "7",
    + "8",
    - "9"
}
```

Vychádzajúc z uvedenej vety, môžeme napísať gramatiku pre generátor jazykových procesorov v požadovanom formáte pre ANTLR nasledovne.

```
grammar Test1;

test : title question+;

title : STRING;

question : 'question' STRING '{' answer (',' answer)+ '}';

answer : '-' STRING  # Incorrect
       | '+' STRING  # Correct
       ;

STRING : '"' .*? '"';
WS : [ \r\t\n]+ -> skip;
```

Pokúsme sa teraz zmeniť konkrétnu syntax nášho jazyka tak, že odstránime úvodzovky a zátvorky a použijeme nové riadky na určenie konca textu otázky a odpovede. Vďaka tomu sa stane zápis vety stručnejší a jeho forma sa ešte viac priblíži tomu ako testy zvyčajne zapisujeme.

```
Test z astronomie

Ktora planéta je obývana?
- Merkúr
- Venuša
+ Zem
- Mars

Kolko planet ma Slnecna sustava?
- 6
- 7
+ 8
- 9
```

Definujme teraz gramatiku pre konkrétnu syntax jazyka didaktických testov vo formáte pre ANTLR podľa predošlého príkladu vety.

```
grammar Test2;

test : title question+;

title : STRING NL+;

question : STRING NL+ answer answer+ NL*;

answer : '-' STRING NL # Incorrect
       | '+' STRING NL # Correct
       ;

STRING : [a-zA-Z0-9]~[-+\n]*;
NL : '\r'? '\n';
WS : [ \t]+ -> skip;
```

Na základe uvedenej gramatiky vygeneruje ANTLR jazykový procesor, ktorý použijeme na spracovanie textového vstupu. Tento jazykový procesor zároveň vybuduje zo vstupu abstraktný syntaktický strom na základe našej špecifikácie. Pre potreby definovania akcií vytvárajúcich abstraktný syntaktický strom nadefinujme `Test2ParserListener` podľa požiadaviek nástroja ANTLR. Abstraktnú triedu `Test2BaseListener` vygeneruje nástroj ANTLR na základe gramatiky s názvom `Test2`. Jednotlivé metódy sú odvodené z názvov neterminálnych symbolov, resp. názvov pravidiel uvedenej gramatiky a sú volané pri spravovaní vstupnej vety (metódy s prefixom `enter` a `exit`).

```
public class Test2ParserListener extends Test2BaseListener {
    private List<Answer> answers;
    private List<Question> questions;
    private Test test;

    @Override
    public void enterTest(Test2Parser.TestContext ctx) {
        questions = new ArrayList<>();
    }

    @Override
    public void enterQuestion(Test2Parser.QuestionContext ctx) {
```

```
        answers = new ArrayList<>();
    }

    @Override
    public void exitCorrect(Test2Parser.CorrectContext ctx) {
        Answer answer = new Answer(ctx.STRING().getText().trim(), true);
        answers.add(answer);
    }

    @Override
    public void exitIncorrect(Test2Parser.IncorrectContext ctx) {
        Answer answer = new Answer(ctx.STRING().getText().trim(), false);
        answers.add(answer);
    }

    @Override
    public void exitQuestion(Test2Parser.QuestionContext ctx) {
        Question question = new Question(ctx.STRING().getText().trim(),
                answers.toArray(new Answer[answers.size()]));
        questions.add(question);
    }

    @Override
    public void exitTest(Test2Parser.TestContext ctx) {
        test = new Test(ctx.title().getText().trim(),
                questions.toArray(new Question[questions.size()]));
    }

    public Test getTest() {
        return test;
    }
}
```

Teraz nám stačí použiť vygenerovaný jazykový procesor priamo v zdrojovom kóde na spracovanie vstupnej vety nasledovne.

```
Test2Lexer lexer = new Test2Lexer(CharStreams.fromFileName("test2.test"));
CommonTokenStream tokens = new CommonTokenStream(lexer);
Test2Parser parser = new Test2Parser(tokens);
ParseTree tree = parser.test();
ParseTreeWalker treeWalker = new ParseTreeWalker();
Test2ParserListener listener = new Test2ParserListener();
treeWalker.walk(listener, tree);

Test test = listener.getTest();

test.validate();

TestPrinter printer = new TestPrinter();
printer.print(test);
```

Okrem spracovania viet použitím objektov typu `Listnener`, podporuje ANTLR aj prístup k spracovaniu vstupu prostredníctvom vzoru Návštevník (angl. Visitor), ktorý prezentuje nasledujúci zdrojový kód. V uvedenom príklade pomocou vzoru Návštevník rátame počet odpovedí v teste.

```
public class AnswerCountVisitor extends Test2BaseVisitor<Void> {
    private int answerCount;

    @Override
    public Void visitCorrect(Test2Parser.CorrectContext ctx) {
        answerCount++;
        return super.visitCorrect(ctx);
    }

    @Override
    public Void visitIncorrect(Test2Parser.IncorrectContext ctx) {
        answerCount++;
        return super.visitIncorrect(ctx);
    }

    public int getAnswerCount() {
        return answerCount;
    }
}
```

Vytvorenú triedu pre počítanie odpvoedí v didaktickom teste môžeme použiť nasledovne.

```
Test2Lexer lexer = new Test2Lexer(CharStreams.fromFileName("test2.test"));
CommonTokenStream tokens = new CommonTokenStream(lexer);
Test2Parser parser = new Test2Parser(tokens);
ParseTree tree = parser.test();
AnswerCountVisitor visitor = new AnswerCountVisitor();
visitor.visit(tree);
System.out.println(visitor.getAnswerCount());
```

V rámci prípadovej štúdie jazyka didaktických testov sme sa prostredníctvom príkladov zoznámili s viacerými technológiami a prístupmi, ktoré nám umožňujú vytvárať doménovo-špecifické jazyky a nástroje preň. Takto definované jazyky a nástroje je možné začleniť priamo do komplexnejšieho softvérového riešenia prípadne do procesu jeho vytvárania (v prípade modelom riadeného vývoja). Uvedené príklady samozrejme neukazujú všetky možnosti, prístupy a technológie pre tvorbu doménovo-špecifických jazykov a počítačových jazykov vo všeobecnosti. Ďalšie štúdium a spoznávanie témy vývoja jazykov ponechávame na čitateľa.

7 Príklady použitia nástroja YAJCo

Táto kapitola opisuje príklady použitia nástroja *YAJCo*. Zároveň je ukážkou potenciálu vytvoreného generátora jazykových procesorov. Doménové modely sú zapísané vo forme diagramov tried UML. Tieto modely sú vytvorené vo vývojovom prostredí Netbeans IDE [11] zo zápisu v jazyku Java.

7.1. Príklady implementovaných jazykov

Pre potreby experimentálneho overenia navrhnutého prístup k tvorbe jazykov bolo vytvorených sedem počítačových jazykov, ktoré boli následne implementované prostredníctvom prototypu nástroja. Jazyky boli vytvárané s dôrazom na prezentáciu možností navrhnutého nástroja. Zároveň je výber jazykov ukážkou všestranného použitia nástroja pre implementáciu doménovo-špecifických jazykov ako aj jazykov všeobecného použitia. Zvolené a implementované boli nasledujúce jazyky:

1. jednoduchý jazyk aritmetických výrazov (SAL – Simple Arithmetic Language), časť 5.1.1 a 5.2.5,
2. jazyk aritmetických výrazov (AL – Arithmetic Language), časť 7.1.1,
3. štruktúrovaný imperatívny jazyk s typmi (SIL – Structured Imperative Language), časť 7.1.2,
4. procedurálny imperatívny jazyk (PIL – Procedural Imperative Language), časť 7.1.3,
5. jazyk pre riadenie grafického rozhrania (GUIIL – Graphical User Interface Interaction Language), časť 7.1.4,
6. jazyk stavových automatov (SML – State Machine Language), časť 7.1.5,
7. jazyk pre kompozíciu anotácií (LAD – Language of Annotation Designator), časť 7.1.6.

Jazyky 3, 4 predstavujú zástupcov programovacích jazykov všeobecného použitia. Jazyky 1, 2 sú jazyky určené pre zápis celočíselných aritmetických výrazov, jazyky 5, 6, 7 sú doménovo-špecifické jazyky určené pre konkrétne domény. Pri experimentálnych jazykoch je uvedený aj dôvod ich návrhu a implementácie v kontexte generátora jazykových procesorov. V jednotlivých príkladoch jazykov je uvedená konkrétna syntax vo forme zápisu gramatiky používanom v JavaCC vytvorená generátorom jazykových procesorov.

Terminálne symboly reprezentujúce lexikálne symboly sú uvedené v apostrofoch (napr. `'+'`). V prípade, že je použité meno lexikálnej jednotky, toto meno je uvedené v zátvorkách < > (napr. `<NUMBER>`).

7.1.1. Jazyk aritmetických výrazov

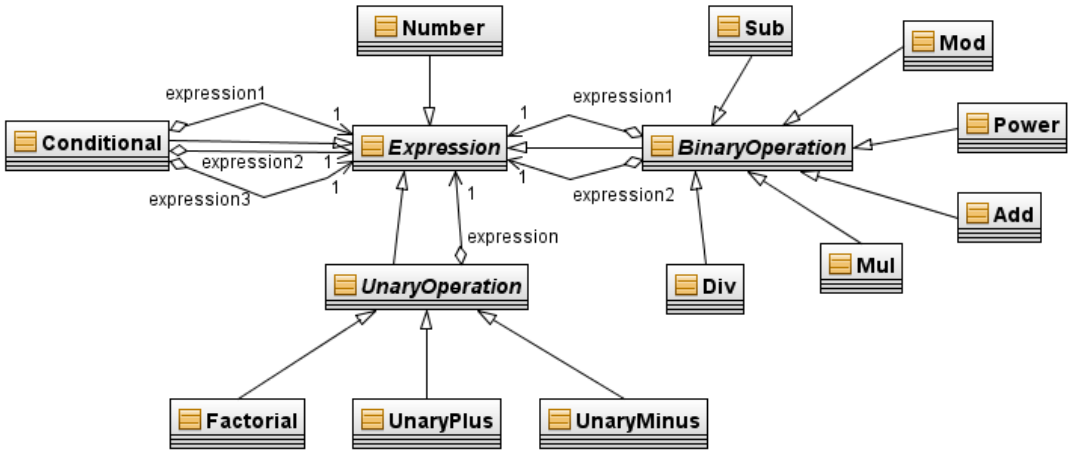
Jazyk celočíselných aritmetických výrazov je rozšírením jednoduchého jazyka aritmetických výrazov s operáciami súčtu (trieda `Add`, symbol +), rozdielu (trieda `Sub`, symbol -), súčinu (trieda `Mul`, symbol *), podielu (trieda `Div`, symbol /) a určenia záporného čísla (trieda `UnaryMinus`, symbol -) o ďalšie operácie. Týmito operáciami sú unárne operácie výpočtu faktoriálu (trieda `Factorial`, symbol !) a unárne plus (trieda `UnaryPlus`, symbol +), binárne operácie určenia zvyšku po celočíselnom delení (trieda `Mod`, symbol %) a operácia určenia mocniny (trieda `Power`, symbol ^) a ternárna operácia

podmieneného výpočtu výrazu (trieda `Conditional`, symboly ? a :) známa z jazyka C. Priorita a asociatívnosť operátorov je uvedená v Tab. 18. V tomto jazyku sa vyskytujú aj dve abstraktné triedy `UnaryOperation` a `BinaryOperation`, ktoré reprezentujú abstrakciu unárnej respektíve binárnej operácie.

Operátor	Priorita	Asociatívnosť
?:	0 (najnižšia)	doprava
+, - (binárne)	1	doľava
*, /, %, ^	2	doľava
-, + (unárne)	3	doprava
!	4 (najvyššia)	doľava

Tab. 18: Priorita a asociatívnosť operátorov v jazyku aritmetických výrazov

Doménový model jazyka vyjadrujúci abstraktnú syntax je znázornený na obrázku Obr. 33. Tento jazyk je ukážkou možností rozšírenia jazyka definovaného prostredníctvom doménového modelu. Zároveň poukazuje na možnosť definície abstraktných doménových tried vyjadrujúcich doménové pojmy. Doménové triedy `UnaryOperation` a `BinaryOperation` neobsahujú žiadne anotácie pre generátor *YAJCo*.



Obr. 33: Jazyk aritmetických výrazov

Použitie abstraktných tried v doménovom modeli je prezentované na operácii celočíselného delenia, ktorá je binárnou operáciu (trieda `Mod`), čo je postihnuté v doménovom modeli dedením od triedy `BinaryOperation`.

```
public final class Mod extends BinaryOperation {
    @Operator(priority = 2)
    public Mod(Expression expression1,
            @Before("PERC") Expression expression2) {
        super(expression1, expression2);
    }

    //Sémantická funkcia
    public long eval() {
        return getExpression1().eval() % getExpression2().eval();
    }
```

```
}
```

Konkrétna syntax generovaná z anotovaného doménového modelu generátorom *YAJCo* je zapísaná bezkontextovou gramatikou v EBNF a obsahuje nasledujúce pravidlá.

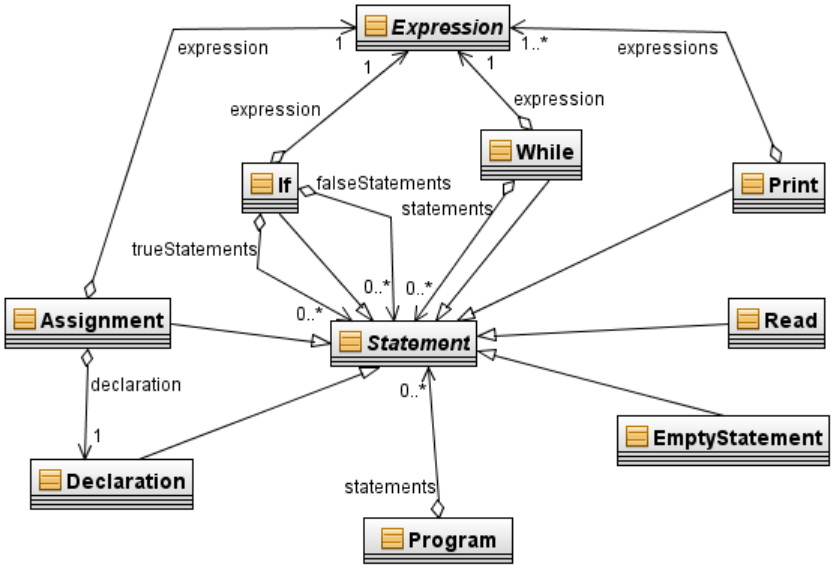
```
Expression0 ::= Expression1 ('?' Expression0 ':' Expression0)?
Expression1 ::= Expression2 (('+' | '-') Expression2)*
Expression2 ::= Expression3 (('*' | '/'| '%'| '^') Expression3)*
Expression3 ::= ('+' | '-') Expression3 | Expression4
Expression4 ::= Expression ('!')*
Expression ::= Number | '(' Expression0 ')'
Number ::= [0-9]+
```

7.1.2. Štruktúrovaný imperatívny jazyk

Ďalším implementovaným jazykom je štruktúrovaný imperatívny jazyk (SIL). Tento jazyk je inšpirovaný zadaním z predmetu „Programovacie jazyky a prekladače“ z FEI VŠB-TU Ostrava [6]. Pri vypracovaní zadania je požadované použiť generátor jazykových procesorov a preto je zaujímavé overiť vhodnosť vytvoreného generátora *YAJCo* pre implementáciu jazyka SIL. Implementácia tohto jazyka overuje použiteľnosť navrhovanej metódy a implementovaného nástroja *YAJCo* vo výučbe.

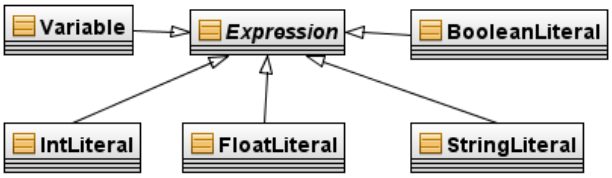
V štruktúrovanom imperatívnom jazyku je program tvorený postupnosťou príkazov. SIL obsahuje výrazy a definuje aritmetické, relačné a logické operátory. V jazyku je definovaný pojem premennej, ktorá má určený typ. Jazyk SIL definuje štyri údajové typy: `int`, `float`, `boolean` a `String`. SIL je jazyk so statickým definovaním typov. Množinu typov nie je možné rozšíriť. Jazyk definuje nasledujúce príkazy, ktoré sú uvedené aj v doménovom modeli na Obr. 34:

- prázdny príkaz (trieda `EmptyStatement`) so zápisom `;`,
- príkaz deklarácie premenných (trieda `Declaration`) so zápisom *typ premenná*$_1$`, ... ,` *premenná*$_n$`;`,
- priraďovací príkaz (trieda `Assignment`) so zápisom *premenná* `=` *výraz*`;`,
- príkaz `read` pre čítanie zo štandardného vstupu a priradenie prečítaných hodnôt do premenných (trieda `Read`) so zápisom `read` *premenná*$_1$`, ... ,` *premenná*$_n$`;`,
- príkaz `print` pre výpis hodnoty výrazov na štandardný výstup (trieda `Print`) so zápisom `print` *výraz*$_1$`, ... ,` *výraz*$_n$`;`,
- podmienený príkaz `if-else-end` (trieda `If`) so zápisom `if(`*podmienka*`) :` *príkazy*$_1$ `else :` *príkazy*$_2$ `end;`, s nepovinnou časťou `else`,
- príkaz cyklu `while` (trieda `While`) so zápisom `while(`*podmienka*`) :` *príkazy* `end;`.



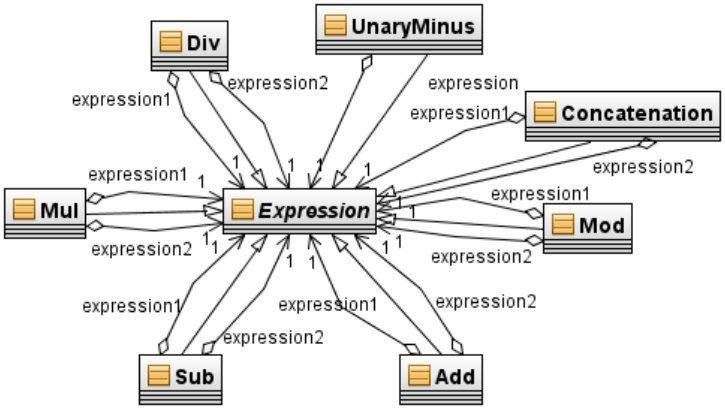
Obr. 34: Štruktúrovaný imperatívny jazyk – príkazy

SIL obsahuje výrazy (trieda `Expression`). Jednoduchými výrazmi (Obr. 35) sú výraz sprístupnenia hodnoty premennej (trieda `Variable`), a literály pre hodnoty typu `boolean`, ktorými sú `True` a `False` (trieda `BooleanLiteral`), literály pre celé čísla (trieda `IntLiteral`), literály pre reálne čísla (trieda `FloatLiteral`) a literály pre reťazce (trieda `StringLiteral`).



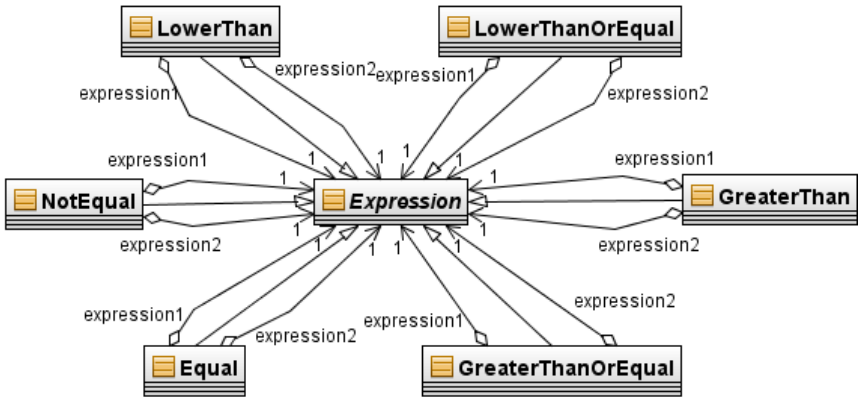
Obr. 35: Štruktúrovaný imperatívny jazyk – jednoduché výrazy

SIL definuje aritmetické operátory (Obr. 36) pre súčet (trieda `Add`), rozdiel (trieda `Sub`), súčin (trieda `Mul`), podiel (trieda `Div`), zvyšok po celočíselnom delení (trieda `Mod`) a určenie záporného čísla (trieda `UnaryMinus`). Na spájanie reťazcov je určený binárny operátor definovaný triedou `Concatenation` (symbol .).



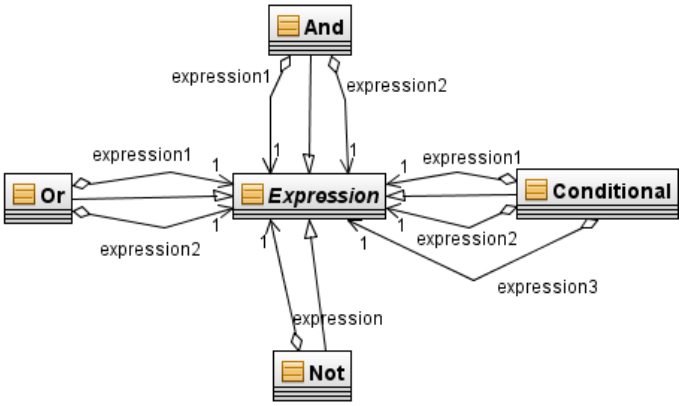
Obr. 36: Štruktúrovaný imperatívny jazyk – aritmetické operátory a spájanie reťazcov

Ďalším druhom definovaných operátorov sú relačné operátory: rovný (trieda `Equal`, zápis ==), rôzny (trieda `NotEqual`, zápis !=), menší ako (trieda `LowerThan`, zápis <), menší alebo rovný ako (trieda `LowerThanOrEqual`, zápis <=), väčší ako (trieda `GreaterThan`, zápis >), väčší alebo rovný ako (trieda `GreaterThanOrEqual`, zápis >=). Doménový model relačných operátorov je zobrazený na Obr. 37.



Obr. 37: Štruktúrovaný imperatívny jazyk – relačné operátory

Posledným druhom operácií sú logické operácie: operátor konjunkcie (trieda `And`, zápis &&), disjunkcie (trieda `Or`, zápis ||), negácie (trieda `Not`, zápis !), a podmienený výraz (trieda `Conditional`, zápis ? :).



Obr. 38: Štruktúrovaný imperatívny jazyk – logické operátory

Priorita a asociatívnosť operátorov je definovaná analogickým spôsobom s jazykom aritmetických výrazov (časť 7.1.2). Konkrétna syntax generovaná z anotovaného doménového modelu generátorom *YAJCo* obsahuje nasledujúce pravidlá:

```
Program ::= Statement*

Statement ::= EmptyStatement | Declaration | Assignment | Read
                | Print | If | While
EmptyStatement ::= ';'
Declaration ::= Type <IDENT> (',' <IDENT>)* ';'
Assignment ::= <IDENT> '=' Expression0 ';'
Read ::= 'read' <IDENT> (',' <IDENT>)* ';'
Print ::= 'print' Expression0 (',' Expression0)* ';'
If ::= 'if' '(' Expression0 ')' ':' Statement*
```

```
            ('else' ':' Statement*)? 'end' ';'
While ::= 'while' '(' Expression0 ')' ':' Statement* 'end' ';'

Expression0 ::= Expression1 ('?' Expression0 ':' Expression1)*
Expression1 ::= Expression2 ('||' Expression2)*
Expression2 ::= Expression4 ('&&' Expression4)*
Expression4 ::= Expression5 (('+' | '-' | '.') Expression5)*
Expression5 ::= Expression6 (('*' | '/' | '%') Expression6)*
Expression6 ::= ('!' | '-') Expression6) | Expression8
Expression8 ::= Expression9 (('==' | '!=') Expression9)*
Expression9 ::= Expression (('<' | '<=' | '>' | '>=') Expression)*
Expression ::= BooleanLiteral | IntLiteral | FloatLiteral | StringLiteral
                  | Variable | '(' Expression0 ')'

BooleanLiteral ::= <BOOLEAN_VALUE>
IntLiteral ::= <INT_VALUE>
FloatLiteral ::= <FLOAT_VALUE>
StringLiteral ::= <STRING_VALUE>
Variable ::= <IDENT>

Type ::= 'int' | 'float' | 'booelan' | 'String'
```

7.1.3. Procedurálny imperatívny jazyk

Procedurálny imperatívny jazyk (PIL) je ďalším implementovaným experimentálnym jazykom. Program v jazyku pozostáva z množiny funkcií a hlavného bloku programu. Napriek názvu procedurálny jazyk obsahuje len funkcie. PIL ale nie je funkcionálny jazyk lebo neumožňuje používať funkcie ako hodnoty. Jazyk PIL podporuje len jeden údajový typ (celé čísla) a neumožňuje rozšíriť množinu typov o ďalší údajový typ. Všetky funkcie majú implicitný typ návratovej hodnoty (celé číslo), preto nie je nutné tento typ pri funkciách uvádzať. Z tohto dôvodu tiež nie je nutné deklarovať typ premenných. Syntax jazyka PIL je podobná jazyku C. Procedurálny imperatívny jazyk definuje nasledujúce príkazy (Obr. 39):

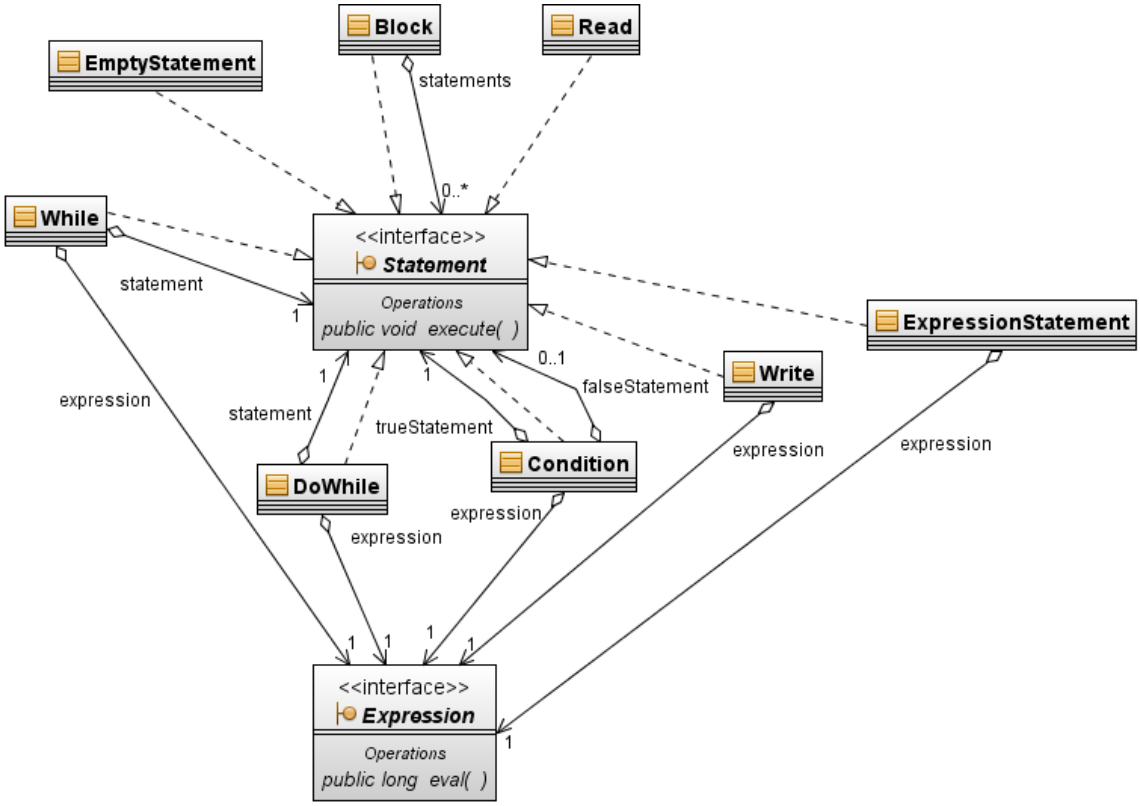
- prázdny príkaz (trieda `EmptyStatement`) so zápisom `;`,
- príkaz tvorený výrazom (trieda `ExpressionStatement`) so zápisom *`výraz`*`;`,
- blok (trieda `Block`) so zápisom `{`*`príkaz`*$_1$ ... *`príkaz`*$_n$`}`,
- príkaz pre načítanie celočíselnej hodnoty zo štandardného vstupu do premennej (trieda `Read`) so zápisom `read` *`premenná`*`;`,
- príkaz pre výpis celočíselnej hodnoty výrazu na štandardný výstup (trieda `Write`) so zápisom `write` *`výraz`*`;`,
- podmienený príkaz (trieda `Condition`) so zápisom `if(`*`výraz`*`)` *`príkaz`*$_1$ `else` *`príkaz`*$_2$, s nepovinnou časťou `else`,
- príkaz cyklu s podmienkou na začiatku (trieda `While`) so zápisom `while(`*`výraz`*`)` *`príkaz`*,
- príkaz cyklu s podmienkou na konci (trieda `DoWhile`) so zápisom `do` *`príkaz`* `while(`*`výraz`*`);`,
- príkaz návratu z funkcie (trieda `Return`) so zápisom `return` *`výraz`*`;`.

V tomto jazyku sú pre definíciu abstraktných pojmov výraz a príkaz použité rozhrania (rozhranie `Expression` pre výraz, rozhranie `Statement` pre príkaz) na rozdiel od predošlých jazykov, kde boli použité abstraktné triedy. Rozhranie umožňuje deklarovať metódy bez ich implementácie, čo je postačujúce v tomto prípade pre definíciu uvedených abstraktných pojmov. Rozhranie `Expression` pre vyjadrenie výrazu v doménovom modeli má nasledujúci tvar.

```
@Parentheses
public interface Expression {
    long eval();
}
```

Rozhranie `Statement` definujúce príkaz v abstraktnej syntaxi jazyka má nasledujúce vyjadrenie.

```
public interface Statement {
    void execute();
}
```



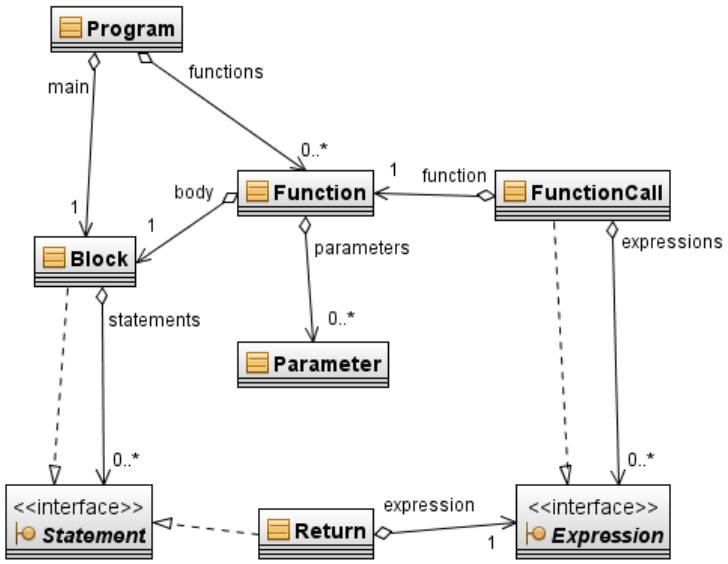
Obr. 39: Procedurálny imperatívny jazyk – príkazy

Oproti predošlým uvedeným jazykom PIL definuje pojem funkcie (trieda `Function`), parametra (trieda `Parameter`), výraz volania funkcie (trieda `FunctionCall`) a príkaz návratu z funkcie (trieda `Return`). Vzťah uvedených doménových tried je definovaný doménovým modelom na Obr. 40. Zápis doménových tried `Function` a `FunctionCall` v jazyku Java s anotáciami bol uvedený v časti 5.3 ako príklad pre vysvetlenie transformácie abstraktného syntaktického stromu na graf. Doménová trieda `Parameter` je definovaná nasledovne:

```
public class Parameter {
```

```
        @Identifier(unique = "../Parameter")
        private final String name;

        public Parameter(String ident) {
            this.name = ident;
        }

        public String getName() {
            return name;
        }
    }
```

Prostredníctvom anotácie `@Identifier` je určené, že meno parametra musí byť jedinečné v kontexte definície funkcie. Na určenie jedinečnosti je použitý XPath výraz `../Parameter`, ktorý určuje, že v kontexte rodičovského uzla (uzol typu `Function`) musí mať parameter jedinečné meno.



Obr. 40: Procedurálny imperatívny jazyk – funkcie

Konkrétna syntax jazyka PIL generovaná z anotovaného doménového modelu generátorom *YAJCo* obsahuje nasledujúce pravidlá:

```
Program ::= ((Function)* Block)

Function ::= <IDENT> '(' (Parameter (',' Parameter)*)? ')' Block
Parameter ::= <IDENT>

Statement ::= EmptyStatement | ExpressionStatement | Block | Read | Write
                | Condition | While | DoWhile | Return
EmptyStatement ::= ';'
ExpressionStatement ::= Expression1 ';'
Block ::= '{' (Statement)* '}'
Read ::= 'read' <IDENT> ';'
Write ::= 'write' Expression1 ';'
Condition ::= 'if' '(' Expression1 ')' Statement ('else' Statement)?
While ::= 'while' '(' Expression1 ')' Statement
DoWhile ::= 'do' Statement 'while' '(' Expression1 ')' ';'
Return ::= 'return' Expression1 ';'

Expression1 ::= Expression2 (('=' | '+=' | '-=' | '*=' | '/='
```

```
                                                | '%=' Expression1)?
Expression2 ::= Expression3 ('?' Expression1 ':' Expression3)*
Expression3 ::= Expression4 ('||' Expression4)*
Expression4 ::= Expression8 ('&&' Expression8)*
Expression8 ::= Expression9 (('==' | '!=') Expression9)*
Expression9 ::= Expression11 (('<' | '<=' | '>' | '>=') Expression11)*
Expression11 ::= Expression12 (('+' | '-') Expression12)*
Expression12 ::= Expression13 (('*' | '/' | '%') Expression13)*
Expression13 ::= ('+' | '-' | '!') Expression13 | Expression14
Expression14 ::= Expression ('--' | '++')? | ('++' | '--') Expression
Expression ::= Number | Variable | FunctionCall | '(' Expression1 ')'

Number ::= <VALUE>
Variable ::= <IDENT>
FunctionCall ::= <IDENT> '(' (Expression1 (',' Expression1)*)? ')'
```

Ako jednoduchý príklad vety z jazyka PIL je možné uviesť program na výpis absolútnej hodnoty čísel od `-10`, po `10`. V uvedenom programe je definovaná funkcia `abs` a uvedený hlavný blok programu.

```
abs(x) {
  return x < 0 ? - x : x;
}

{
  i = -10;
  while ( i <= 10 ) {
    write abs(i);
    i = i + 1;
  }
}
```

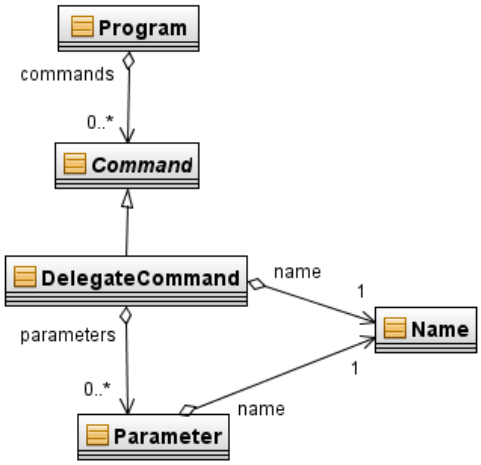
7.1.4. Jazyk pre riadenie grafického rozhrania

Za počítačový jazyk je možné považovať aj jazyk, ktorý je vyjadrený konkrétnym softvérovým systémom. Používateľské rozhranie systému teda definuje jednu z možných konkrétnych foriem jazyka. Takto definovaný jazyk je doménovo-špecifický jazyk, ktorý je určený pre rovnakú doménu ako samotný systém. Textový editor teda definuje jazyk manipulácie s textovými dokumentmi (zmeniť písmo, vložiť text, odstrániť odstavec), medicínsky informačný systém používa pojmy z domény zdravotníctva a definuje formálny jazyk v tejto doméne (vyšetriť pacienta, určiť diagnózu, predpísať liek). Zatiaľ čo používateľské rozhranie definuje syntax jazyka, sémantika je definovaná implementovaným systémom.

Vzhľadom na to, že jazyk môže mať viacero rôznych konkrétnych reprezentácií, je možné pre jazyk definovaný softvérovým systémom vytvoriť inú konkrétnu formu ako je forma definovaná používateľským rozhraním. V tomto príklade to bude jednoduchá textová forma. Prostredníctvom tejto formy bude možné definovať vety, ktoré budú zodpovedať použitiu používateľského rozhrania s cieľom vykonať tú istú činnosť. Takto definovaná forma konkrétnej syntaxe umožní jednoduchšiu kompozíciu viet z jazyka, prenositeľnosť a vytváranie abstraktných pojmov na základe vytvorených viet.

Jazyk pre riadenie grafického rozhrania (GUIIL) je jednoduchý doménovo-špecifický jazyk. Veta v tomto jazyk pozostáva z postupnosti príkazov (trieda `Command`), ktoré je možné vykonať. Predpokladom pri návrhu jazyka je jeho postupné rozširovanie v závislosti od charakteru používateľských rozhraní softvérových systémov, v ktorých má

byť jazyk aplikovaný. Doménový model vyjadrujúci abstraktnú syntax jazyka je zobrazený na Obr. 41.



Obr. 41: Jazyk pre riadenie grafického rozhrania

Základným bodom rozšírenia navrhnutého jazyka je abstraktná trieda `Command`, ktorá môže byť implementovaná aj ako rozhranie bez vplyvu na generovanú konkrétnu syntax.

```
public abstract class Command {
    public abstract <T> boolean execute(T component);
}
```

Trieda `DelegateCommand` je konkrétnou implementáciou príkazu pre používateľské rozhranie. Umožňuje zapísať pomenovaný parametrizovaný príkaz. Parametre sú oddelené čiarkou a sú uvedené v okrúhlych zátvorkách.

```
public class DelegateCommand extends Command {
    private final Name name;

    private final Parameter[] parameters;

    public DelegateCommand(Name name,
            @Optional
            @Before("LPAR")
            @After("RPAR")
            @Separator("COMMA") Parameter[] parameters) {
        this.name = name;
        this.parameters = parameters;
    }
    //...
}
```

Konkrétna syntax jazyka GUIIL vygenerovaná z anotovaného doménového modelu nástrojom *YAJCo* ma nasledujúci tvar:

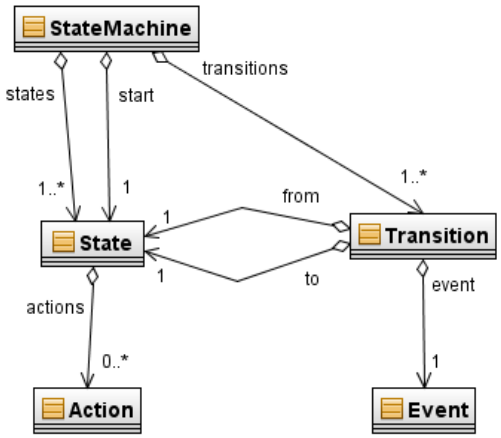
```
Program ::= (Command)*
Command ::= DelegateCommand
DelegateCommand ::= Name ('(' (Parameter (',' Parameter)*)? ')')?
Parameter ::= Name '=' <VALUE>
Name ::= <NAME>
```

Ako príklad vety z jazyka je možné uviesť nasledujúci zápis reprezentujúci postupnosť príkazov pre aplikáciu kalkulačky.

```
1 2 - 3 4 + 5 [=]
[About Calculator]
[ OK ]
Exit
```

7.1.5. Jazyk stavových automatov

Jazyk stavových automatov (SML) je klasický príklad na doménovo-špecifické jazyky. Napriek tomu, že SML je jednoduchý jazyk z pohľadu počtu doménových tried, umožňuje dobre prezentovať navrhnutú metódu a možnosti vytvoreného generátora. Prostredníctvom vety v tomto jazyku je možné definovať stavový automat (trieda `StateMachine`), ktorý má konečný počet stavov, prechodov medzi stavmi, definuje vykonateľné akcie a udalosti, ktoré aktivujú prechod. Každý stav (trieda `State`) má svoje jedinečné meno. Stav tiež môže definovať akcie (trieda `Action`), ktoré majú byť vykonané pri zmene stavu. Prechody zo stavu do stavu (trieda `Transition`) sú aktivované výskytom konkrétnej pomenovanej udalosti (trieda `Event`). Stavový automat má definovaný jeden počiatočný stav. Abstraktná syntax vyjadrená doménovým modelom je zobrazená na Obr. 42.



Obr. 42: Jazyk stavových automatov

Nasledujúci príklad je vetou z jazyka SML.

```
start Running

#Definovanie stavov
state Ready {pump_off}
state Running {pump_on}
state Unsafe {pump_off}

#Definovanie prechodov
transition from Ready to Running when water_high
transition from Running to Ready when water_low
transition from Running to Unsafe when metan_high
transition from Unsafe to Running when metan_low
```

Doménová trieda `State` je definovaná v jazyku Java prostredníctvom anotovanej triedy nasledovne:

```
public class State {
    @Identifier
    private final String name;
```

```
        private final Action[] actions;

        private List<Transition> incomingTransitions;

        private List<Transition> outgoingTransitions;

        @Before("STATE")
        public State(String name,
                @Optional
                @Before("LBR")
                @After("RBR") Action[] actions) {
            this.name = name;
            this.actions = actions;
        }

        //...
    }
```

Doménová trieda `Transition` je definovaná v jazyku Java prostredníctvom anotovanej triedy nasledovne:

```
public class Transition {
    private State from;

    private State to;

    private final Event event;

    @Before("TRANSITION")
    public Transition(
            @Before("FROM")
            @Token("NAME")
            @References(value = State.class, field = "from") String from,
            @Before("TO")
            @Token("NAME")
            @References(value = State.class, field = "to") String to,
            @Before("WHEN") Event event) {
        this.event = event;
    }

    //...
}
```

Jazyk SML je zároveň ukážkou použitia vyhľadávača referencií. Stav má svoj identifikátor a prechod obsahuje zdrojový a cieľový stav. Jazyk SML definuje grafové štruktúry, ktoré sú konštruované práve vyhľadávačom referencií. Hlavnou (koreňovou) triedou je trieda `StateMachine`. Táto doménová trieda obsahuje anotáciu `@PostConstruct`, ktorá zabezpeční zavolanie anotovanej metódy `init()` po vytvorení objektu reprezentujúceho stavový automat. V tejto metóde je nastavený aktuálny stav na počiatočný stav.

```
    public class StateMachine {
        private State start;

        private State[] states;

        private Transition[] transitions;
```

```
    private State currentState;

    public StateMachine(
            @Before("START")
            @Token("NAME")
            @References(State.class) String start,
            @Range(minOccurs = 1) State[] states,
            @Range(minOccurs = 1) Transition[] transitions) {
        this.states = states;
        this.transitions = transitions;
    }

    @PostConstruct
    public void init() {
        changeState(start);
    }

    //...
}
```

Konkrétna syntax jazyka PIL generovaná z anotovaného doménového modelu generátorom *YAJCo* má nasledujúci tvar:

```
StateMachine ::= 'start' <NAME> State+ Transition+
State ::= 'state' <NAME> ('{' Action* '}')?
Transition ::= 'transition' 'from' <NAME> 'to' <NAME> 'when' Event
Action ::= <NAME>
Event ::= <NAME>
```

Poslednou ukážkou z jazyka SML je možnosť rozšírenia akcií o akciu vyjadrenú doménovou triedou `DiagnoseAction`. Pre toto rozšírenie stačí vytvoriť anotovanú doménovú triedu, ktorá bude dediť od triedy `Action`.

```
public class DiagnoseAction extends Action {
    @Before("DIAGNOSE")
    public DiagnoseAction() {
        super("diagnose");
    }

    //...
}
```

Táto modifikácia doménového modelu si nevyžiada žiadne iné modifikácie doménového modelu. Vygenerovaná konkrétna syntax jazyka po tejto modifikácii doménového modelu bude obsahovať dve zmeny. Prvou zmenou je jedno nové pravidlo `DiagnoseAction` a druhou zmenou je modifikácia pravidla `Action`.

```
Action ::= <NAME> | DiagnoseAction
DiagnoseAction ::= 'diagnose'
```

7.1.6. Jazyk pre kompozíciu anotácií

Jazyk pre kompozíciu anotácií (LAD) zohral kľúčovú rolu pri tvorbe generátora jazykových procesorov *YAJCo*. Práve pri návrhu a implementácii tohto jazyka vznikla myšlienka na vytvorenie generátora jazykových procesorov z dôvodu nízkej flexibility existujúcich nástrojov a prístupov na tvorbu jazykových procesorov. Tento jazyk bol postupne rozširovaný evolučnými krokmi o ďalšie konštrukcie, čo pri súčasných

prístupoch a nástrojoch pre tvorbu jazykových procesorov znamenalo časté zásahy do existujúcich artefaktov aj vďaka nesymetrickej separácií pojmov v artefaktoch. Vytvorený generátor *YAJCo* zjednodušil vývoj a implementáciu jazyka pre kompozíciu anotácií ako aj jazykového procesora a umožnil sústrediť sa na abstraktnú syntax a sémantiku, ktoré sú kľúčové pri návrhu jazyka. Navrhnutá metóda a generátor taktiež minimalizovali vplyv konkrétnych technológií spracovania viet jazyka.

Jazyk pre kompozíciu anotácií (LAD) umožňuje definovať obmedzenia pre vkladanie anotácií do zdrojových kódov jazyka Java. Prostredníctvom viet z jazyka LAD je možné pre vytvorený anotačný typ definovať, ktoré elementy jazyka Java je možné anotáciou definovaného typu označiť. Vytvorený jazyk LAD je určený pre nástroje pre podporu atribútovo-orientovaného programovania [118]. Na rozdiel od iných riešení umožňujúcich definovanie obmedzení pri používaní anotácií [15][90], navrhnuté riešenie je založené na jednoduchom a rozšíriteľnom jazyku s textovou konkrétnou syntaxou. Konkrétna syntax jazyka LAD je čiastočne inšpirovaná aspektovým jazykom AspectJ, konkrétne opisom bodov prierezov [37][64].

Význam a použite jazyka je možné prezentovať na jednoduchom príklade anotačného typu `PostCreate`, ktorý je definovaný platformou Java [36]. Tento definovaný anotačný typ sa používa na označenie metódy, ktorá bude zavolaná kontajnerom spravujúcim objekty a zabezpečujúcim injektáž zdrojov po vytvorení objektu. V štandardnej dokumentácii k tomuto anotačnému typu je neformálne definované, kde môže byť anotácia použitá [109]:

> *„Anotáciu `@PostConstruct` je možné použiť iba na označenie metódy. Označená metóda nesmie mať parametre, okrem prípadu použitia EJB zachytávačov, kde je jediným povoleným typom parametra typ `InvocationContext`. Typ návratovej hodnoty musí byť `void`. Metóda nesmie špecifikovať vznik žiadnej kontrolovanej výnimky a nesmie byť označená modifikátorom `static`.“*

Prostredníctvom vety z jazyka LAD je možné explicitne uviesť tieto požiadavky v textovej podobe dobre čitateľnej pre programátorov a zároveň vhodnej pre automatizované spracovanie prostriedkami podporujúcimi návrh a vývoj aplikácií. Automatizovaná kontrola anotácií pri vývoji zjednodušuje vytváranie systému a znižuje požiadavky kladené na programátora. Pre anotačný typ `PostCreate` je možné uvedené neformálne požiadavky zapísať formálne vetou v jazyku LAD nasledovne:

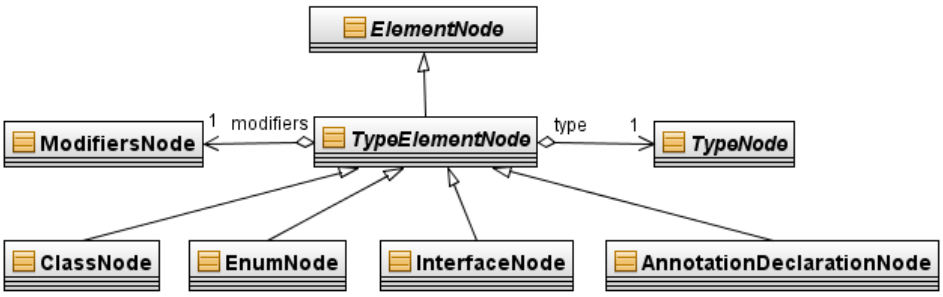
```
method(!static void *())
|| method(!static void *(javax.ejb.InvocationContext *))
```

Tento zápis definuje dve prípustné formy metód, ktoré je možné označiť anotáciou `@PostCreate`. Prvá forma opisuje bezparametrické metódy a je definovaná predikátom `method(!static void *())`, ktorý umožňuje zisiť, či je anotácia použitá na označenie nestatickej bezparametrickej metódy s návratovou hodnotou typu `void`. Druhá forma metódy je definovaná predikátom `method(!static void *(javax.ejb.InvocationContext *))`, ktorý umožňuje zistiť, či je anotácia použitá na označenie nestatickej metódy s jedným parametrom typu `InvocationContext`. Symbol `*` zastupuje ľubovoľné meno metódy. Operátor || predstavuje operáciu disjunkcie, unárny operátor ! je operátor logickej negácie. Základom vytvoreného jazyka LAD sú konštrukcie označenia elementov jazyka Java, ktoré môžu byť

označené anotáciami. Jazyk LAD definuje nasledujúce predikáty pre charakterizovanie elementov jazyka Java:

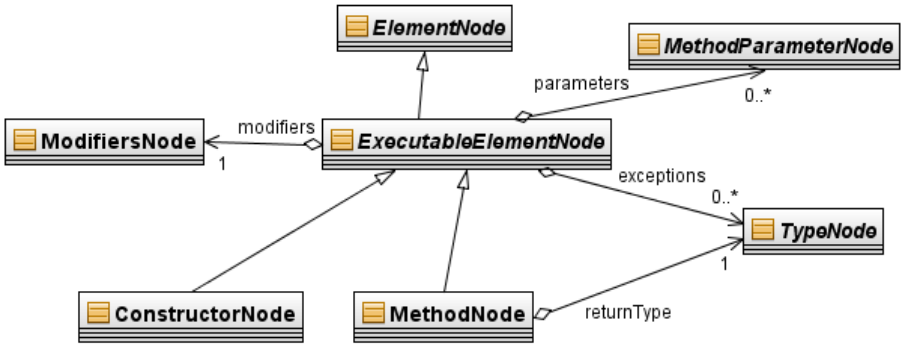
- element opisujúci triedu jazyka Java (trieda `ClassNode`) so zápisom `class(`*`modifkátory typ`*`)`; napríklad predikát `class(public abstract javax.swing.*)` určujúci všetky verejné abstraktné triedy v balíku `javax.swing`,
- element opisujúci enumeračný typ (trieda `EnumNode`) so zápisom `enum(`*`modifkátory typ`*`)`; napríklad predikát `enum(javax.lang.model.element.*)` určujúci všetky enumeračné typy v balíku `javax.lang.model.element`,
- element opisujúci rozhranie jazyka Java (trieda `InterfaceNode`) so zápisom `interface(`*`modifkátory typ`*`)`; napríklad predikát `interface(public java.io.Serializable)` určujúci verejné rozhranie `javax.lang.model.element`,
- element opisujúci anotačný typ (trieda `AnnotationDeclarationNode`) so zápisom `annotation(`*`modifkátory typ`*`)`; napríklad predikát `annotation(javax.annotation.R*)` určujúci všetky anotácie v balíku `javax.annotation` začínajúce na písmeno R,
- element opisujúci konštruktor triedy (trieda `ConstructorNode`) so zápisom `constructor(`*`modifkátory`* `new(`*`parametre`*`) throws `*`výnimky`*`)`; napríklad predikát `constructor(!public new())` určujúci bezparametrický neverejný konštruktor,
- element opisujúci metódu (trieda `MethodNode`) so zápisom `method(`*`modifkátory typ meno`*`(`*`parametre`*`) throws `*`výnimky`*`)`; napríklad predikát `method(public set*(int *))` určujúci verejnú metódu s menom začínajúcim na `set` a jediným parametrom typu `int`,
- element opisujúci premennú triedy alebo objektu (trieda `FieldNode`) so zápisom `field(`*`modifkátory typ meno`*`)`; napríklad predikát `field(private int *)` určujúci súkromnú premennú triedu s ľubovoľným menom typu `int`,
- element opisujúci parameter metódy alebo konštruktora (trieda `ParameterNode`) so zápisom `param(`*`modifkátory typ meno`*`)`; napríklad predikát `param(final String name)` určujúci parameter s menom `name` typu `String`,
- element opisujúci hodnotu enumeračného typu (trieda `EnumConstantNode`) so zápisom `enum_constant(`*`meno`*`)`; napríklad predikát `enum_constant(PRIVATE)` určujúci všetky hodnoty enumeračného typu s názvom `PRIVATE`,
- element opisujúci balík (trieda `PackageNode`) so zápisom `package(`*`meno`*`)`; napríklad predikát `package(java.*)` určujúci všetky balíky začínajúce prefixom `java`,
- element umožňujúci rozšírenie jazyka vo forme používateľom definovanej funkcie (trieda `UserFunctionNode`).

Hlavnou doménovou triedou je abstraktná trieda `ElementNode`. Táto trieda reprezentuje predikáty umožňujúce overovať správnosť väzby anotácií na elementy jazyka Java. Vzťahy doménových tried, ktoré reprezentujú predikáty typov v jazyku Java – triedy (trieda `ClassNode`), rozhrania (trieda `InterfaceNode`), enumeračné typy (trieda `EnumNode`) a anotačné typy (trieda `AnnotationDeclarationNode`), sú zobrazené na obrázku Obr. 43.



Obr. 43: Jazyk pre riadenie kompozície anotácií – elementy tried

Doménové triedy reprezentujúce predikáty pre konštruktor (trieda `ConstructorNode`) a metódu (trieda `MethodNode`) sú zobrazené na Obr. 44. Predikát pre konštruktor a predikát pre metódu môže určovať ľubovoľný počet parametrov (trieda `MethodParameterNode`) a výnimiek.

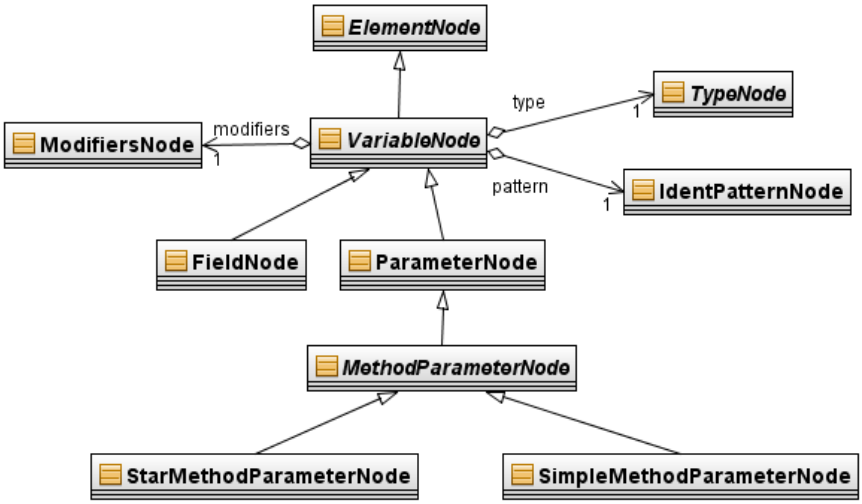


Obr. 44: Jazyk pre riadenie kompozície anotácií – elementy metód

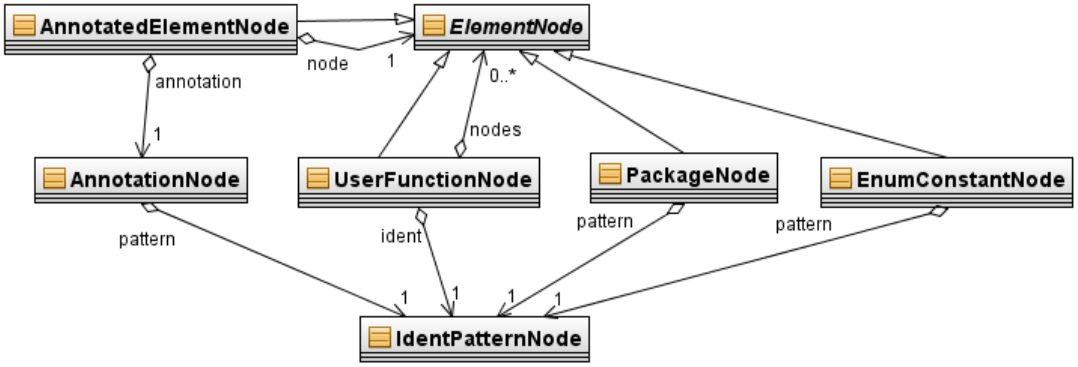
Doménové triedy reprezentujúce predikáty pre premenné tried a objektov (trieda `FieldNode`) a parametre (trieda `ParameterNode`) sú zobrazné na Obr. 45. Parameter môže byť zapísaný symbolom `*` (trieda `StarMethodParameterNode`), ktorý reprezentuje parameter bez modifikátora s ľubovoľným menom a typom. Ďalšou možnosťou je zápis parametra s modifikátorom, menom alebo typom (trieda `SimpleMethodParameterNode`).

Doménové triedy pre predikáty balíka (trieda `PackageNode`), hodnoty enumeračného typu (trieda `EnumConstantNode`), používateľom definovanú funkciu (trieda `UserFunctionNode`) a anotovanie elementov (trieda `AnnotationNode`) sú zobrazené na Obr. 46.

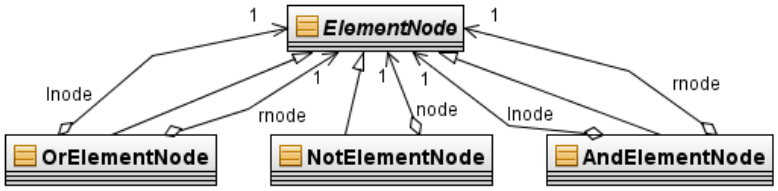
Jednotlivé predikáty je možné kombinovať prostredníctvom logických operácií negácie (trieda `NotElementNode`), konjunkcie (trieda `AndElementNode`) a disjunkcie (trieda `OrElementNode`) (Obr. 47).



Obr. 45: Jazyk pre riadenie kompozície anotácií – elementy premenných

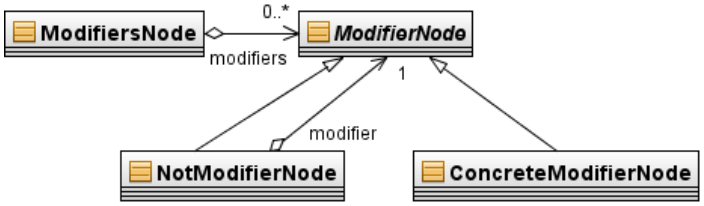


Obr. 46: Jazyk pre riadenie kompozície anotácií – ostatné elementy



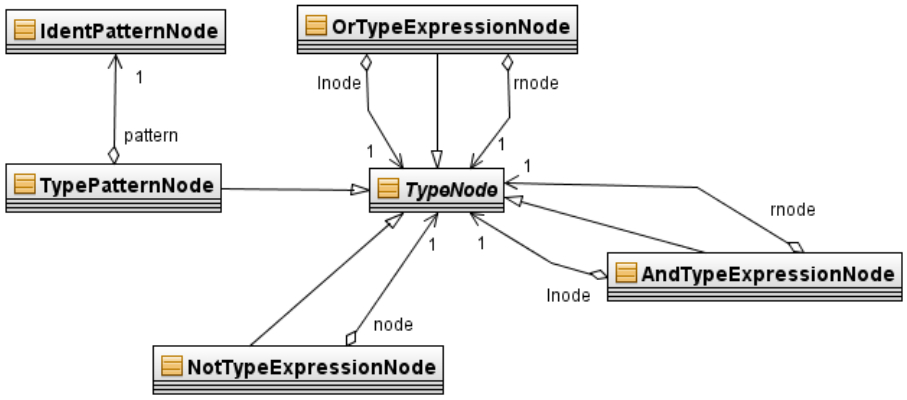
Obr. 47: Jazyk pre riadenie kompozície anotácií – operátory

Vzťahy predikátov pre modifikátory sú uvedené na Obr. 48.



Obr. 48: Jazyk pre riadenie kompozície anotácií – modifikátory

Triedy predikátov umožňujúcich overovanie typov a operácie pre kompozíciu týchto predikátov sú zobrazené na Obr. 49.



Obr. 49: Jazyk pre riadenie kompozície anotácií – typové výrazy

Konkrétna syntax jazyka LAD generovaná prostredníctvom generátora *YAJCo* z anotovaného doménového modelu má nasledujúci tvar:

```
ElementNode1 ::= ElementNode2 (('&&' | '||') ElementNode2)*
ElementNode2 ::= '!' ElementNode2 | ElementNode3
ElementNode3 ::= AnnotationNode ElementNode3 | ElementNode
ElementNode ::= ClassNode | EnumNode | InterfaceNode
            | AnnotationDeclarationNode | ConstructorNode | MethodNode
            | FieldNode | ParameterNode | EnumConstantNode | PackageNode
            | UserFunctionNode | '(' ElementNode1 ')'

ClassNode ::= 'class' '(' ModifiersNode TypeNode1 ')'
EnumNode ::= 'enum' '(' ModifiersNode TypeNode1 ')'
InterfaceNode ::= 'interface' '(' ModifiersNode TypeNode1 ')'
AnnotationDeclarationNode ::= 'annotation' '(' ModifiersNode TypeNode1 ')'
ConstructorNode ::= 'constructor' '(' ModifiersNode 'new'
      '(' (MethodParameterNode (',' MethodParameterNode)*)? ')'
      ('throws' TypeNode1 (',' TypeNode1)*)? ')'
MethodNode ::= 'method' '(' ModifiersNode TypeNode1 IdentPatternNode
      '(' (MethodParameterNode (',' MethodParameterNode)*)? ')'
      ('throws' TypeNode1 (',' TypeNode1)*)? ')'
FieldNode ::= 'field' '(' ModifiersNode TypeNode1 IdentPatternNode ')'
ParameterNode ::= 'param' '(' ModifiersNode TypeNode1 IdentPatternNode ')'
EnumConstantNode ::= 'enum_constant' '(' IdentPatternNode ')'
PackageNode ::= 'package' '(' IdentPatternNode ')'
UserFunctionNode ::= IdentPatternNode '(' (ElementNode1 (',' ElementNode1)*)? ')'

AnnotationNode ::= '@' IdentPatternNode

MethodParameterNode ::= SimpleMethodParameterNode | StarMethodParameterNode
SimpleMethodParameterNode ::= ModifiersNode TypeNode1 IdentPatternNode
StarMethodParameterNode ::= IdentPatternNode

ModifiersNode ::= ModifierNode1*
ModifierNode1 ::= '!' ModifierNode1 | ModifierNode
ModifierNode ::= ConcreteModifierNode
ConcreteModifierNode ::= 'public' | 'protected' | 'private' | 'abstract'
                       | 'static' | 'final' | 'transient' | 'volatile'
                       | 'synchronized' | 'native' | 'strictfp'

TypeNode1 ::= TypeNode2 (('&&' | '||') TypeNode2)*
TypeNode2 ::= '!' TypeNode2 | TypeNode
TypeNode ::= TypePatternNode | '(' TypeNode1 ')'
TypePatternNode ::= IdentPatternNode '+'? '[]'*
```

```
IdentPatternNode ::= <PATTERN>
```

Prostredníctvom vytvoreného jazyka pre kompozíciu anotácií s elementmi jazyka Java je možné určiť, ktoré elementy jazyka Java (napr. konštruktory, parametre, triedy) môžu byť anotované anotáciami generátora jazykových procesorov. V Tab. 19 je definované prostredníctvom viet z jazyka LAD ako je možné komponovať anotácie generátora jazykových procesorov *YAJCo* s doménovým modelom zapísaným v jazyku Java.

Anotačný typ	LAD
`Optional`	`param(* *) && parent(constructor(public new(..)))` `  && parent(parent(class(public !abstract *)))`
`Range` `Separator`	`param(* *[]) && parent(constructor(public new(..)))` `  && parent(parent(class(public !abstract *)))`

`Before`
`After`
	`param(* *) && parent(constructor(public new(..)))`
	`  && parent(parent(class(public !abstract *)))`
	`||`
	`constructor(public new(..))`
	`  && parent(class(public !abstract *))`

`Operator`	`constructor(public new(..))` `  && parent(class(public !abstract *))`
`Parentheses`	`class(abstract *) || interface(*)`

`Token`	`param(* *) && parent(constructor(public new(..)))`
	`  && parent(parent(class(public !abstract *)))`
	`||`
	`enum_constant(*)`

`Parser`	`package(*)`

`TokenDef`
`Skip`
`Option`
	`none()`

`Exclude`	`class(*) || interface(*)`

Tab. 19: Definovanie kompozície pre anotačné typy generátora jazykových procesorov

Napríklad anotáciu `@Operator` je možné použiť len na označenie verejného konštruktora s ľubovoľným počtom parametrov (`constructor(public new(..))`) verejnej triedy, ktorá nie je označená modifikátorom asbtract (`class(public !abstract *)`). Anotácie `@TokenDef`, `@Skip` a `@Option` nie je možné použiť na označenie žiadneho elementu.

7.2. Experimentálne výsledky

Cieľom tvorby experimentálnych jazykov bolo empirické zhodnotenie možností použitia vytvoreného generátora jazykových procesorov na implementáciu počítačových jazykov rôzneho charakteru.

V tejto časti sú uvedené výsledky meraní implementovaných jazykov uvedených v predošlých častiach. Merania boli realizované štandardnými nástrojmi umožňujúcimi spočítať počet riadkov a znakov v súbore ako aj implementovaným anotačným procesorom umožňujúcim určiť počet a typ anotácií v zdrojových kódoch doménových tried a typ doménových tried.

Pre jednotlivé jazyky boli merané nasledujúce charakteristiky:

1. Počet doménových tried v jazyku:
   - konkrétne triedy,
   - abstraktné triedy,
   - rozhrania a
   - enumeračné typy.
2. Počet použitých anotácií (kategorizovaný podľa anotačných typov) na označenie elementov doménových tried.
3. Porovnanie počtu doménových tried s anotáciami a bez anotácií, určenie priemerného počtu anotácií na jednu doménovú triedu,
4. Počet definovaných lexikálnych jednotiek pre jazyk.
5. Charakteristika generovaných kódov:
   - počet pravidiel vo vygenerovanej bezkontextovej gramatike opisujúcej konkrétnu syntax,
   - počet riadkov vygenerovaných generátorom *YAJCo* v súbore, ktorý je vstupom pre generátor jazykových procesorov JavaCC,
   - počet znakov vygenerovaných generátorom *YAJCo* v súbore, ktorý je vstupom pre generátor jazykových procesorov JavaCC,
   - celkový počet riadkov v súboroch generovaných nástrojom JavaCC,
   - celkový počet znakov v súboroch generovaných nástrojom JavaCC.

Jednotlivé namerané hodnoty umožňujú vytvorenie predstavy o možnostiach navrhnutej metódy tvorby počítačových jazykov a implementovanom generátore jazykových procesorov *YAJCo*.

Prvou hodnotenou charakteristikou je počet použitých doménových tried pre definíciu abstraktnej syntaxe jazyka. Tento údaj umožňuje zhodnotiť zložitosť jazyka, pretože väčšie množstvo pojmov v jazyku zvyčajne znamená zložitejšie pochopenie. Jednotlivé pojmy sú reprezentované doménovými triedami. Doménové triedy postihujú konkrétne aj abstraktné pojmy. Namerané údaje sú zobrazené v Tab. 20. V zmysle týchto údajov je možné za najzložitejší z uvedených jazykov považovať procedurálny imperatívny jazyk (PIL). Na druhej strane sú najjednoduchšími jazykmi z pohľadu počtu pojmov jazyk stavových automatov (SML) a jazyk pre riadenie grafického rozhrania (GUIIL). Podľa charakteru implementácie doménových tried sú najviac používané konkrétne triedy a abstraktné triedy. Pri definícii jazyka sú abstraktné triedy plne nahraditeľné rozhraniami, ktoré majú rovnaké možnosti pre definíciu pojmov jazyka.

	Počet definovaných typov						
Druh	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
Konkrétne triedy	6	11	30	42	4	6	26
Abstraktné triedy	1	3	2	3	1		7
Rozhrania				2			
Enumeračné typy*			1				1
Spolu	7	14	33	47	5	6	34
* V jazyku LAD je použitý enumeračný typ pre modifikátory definovaný v Java SE							

Tab. 20: Počty definovaných typov v jazykoch

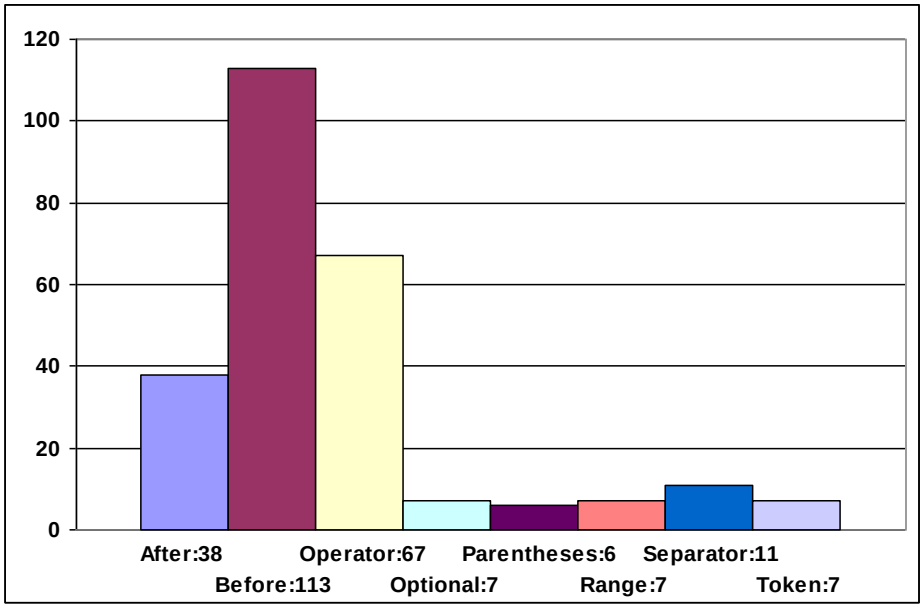
V Tab. 21 sú zobrazené počty anotácií generátora jazykových procesorov *YAJCo* použité na označenie elementov v doménových triedach jednotlivých experimentálnych jazykov. V prípade, že je možné použiť anotáciu na označenie dvoch rôznych elementov, tabuľka uvádza početnosť použitia anotácie aj s uvedením elementu (pre anotácie `@Before`, `@After` a `@Token`).

		Počet výskytov							
Anotácia	**Element**	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**	**Spolu**
`After`	Konštruktor			7	2			10	19
`After`	Parameter		1	2	11	1	1	3	19
`After`	Všetky		1	9	13	1	1	13	**38**
`Before`	Konštruktor			4	1		3	11	19
`Before`	Parameter	5	10	24	36	2	5	12	94
`Before`	Všetky	5	10	28	37	2	8	23	**113**
`Operator`		5	10	17	27			8	**67**
`Optional`				1	1	1	1	3	**7**
`Parentheses`		1	1	1	1			2	**6**
`Range`				3			2	2	**7**
`Separator`				3	2	1		5	**11**
`Token`	Hodnota enum.								0
`Token`	Parameter			2			3	2	7
`Token`	Všetky			2			3	2	**7**
Spolu		**11**	**22**	**64**	**81**	**5**	**15**	**58**	**256**

Tab. 21: Použitie anotácií v experimentálnych jazykoch

Najviac používanou anotáciou v experimentálnych jazykoch je anotácia `@Before`. Z dvoch možných foriem použitia (konštruktor, parameter konštruktora) je častejšou formou použitie na označenie parametra konštruktora. Časté použitie anotácie `@Before` je opodstatnené vzhľadom na to, že táto anotácia určuje zoznam lexikálnych symbolov pred označenou konštrukciou vo vete jazyka. Tieto lexikálne symboly označujú výskyt jazykových pojmov vo vete a slúžia na rozpoznanie tejto konštrukcie. To, že lexikálny symbol označujúci konštrukciu je pred danou konštrukciou, zodpovedá spôsobu nášho čítania viet zľava doprava a umožňuje rýchle a jednoznačné rozpoznanie výskytu pojmu. Vysoký počet anotácií `@Operator` je vzhľadom na charakter implementovaných experimentálnych jazykov opodstatnený. Štyri jazyky priamo definujú výrazy s operátormi (SAL, AL, SIL, PIL) a jazyk pre kompozíciu anotácií (LAD) definuje logické operácie. Na druhej strane nízky počet anotácií `@Token` je spôsobený automatickým odvádzaním mena lexikálneho symbolu z anotovaného parametra alebo hodnoty enumeračného typu. Pri hodnotách enumeračného typu dokonca nebola použitá ani jedna anotácia `@Token`.

Graf zobrazený na Obr. 50 je celkovým zhrnutím použitého počtu anotácií v experimentálnych jazykoch.



Obr. 50: Celkový počet použitých anotácií

Ďalším uskutočneným meraním bolo určenie počtu doménových tried, ktoré obsahujú elementy označené anotáciami generátora *YAJCo* a počtu doménových tried bez anotácií pre jednotlivé experimentálne jazyky. Na základe nameraných hodnôt a počtu anotácií pre jednotlivé experimentálne jazyky boli určené priemerné počty anotácií na doménovú triedu pre experimentálne jazyky ako aj celkový priemer. Výsledky merania sú zobrazené v Tab. 22.

Kategória	SAL	AL	SIL	PIL	GUIIL	SML	LAD	Spolu
Počet typov s anotáciami	6	11	25	39	2	4	23	110
Počet typov bez anotácií	1	3	8	8	3	2	10	35
Celkový počet doménových tried	7	14	33	47	5	6	34	146
Počet anotácií	11	22	64	81	5	15	58	256
Priemerný počet anotácií na triedu	1,57	1,57	1,94	1,72	1,00	2,50	1,71	1,75
Podiel tried bez anotácií ku celkovému počtu doménových tried	0,14	0,21	0,24	0,17	0,60	0,33	0,29	0,24

Tab. 22: Porovnanie počtu typov s anotáciami a bez anotácií

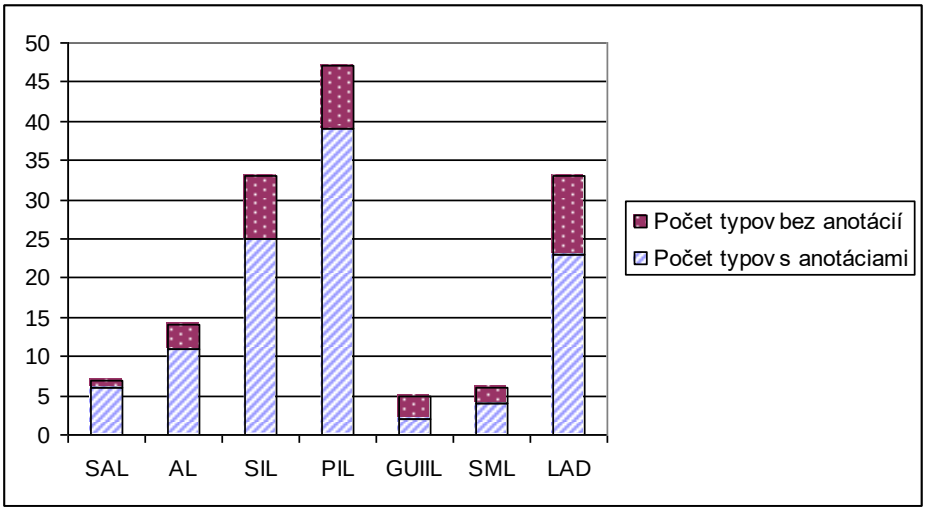
Priemerný počet anotácií na jednu doménovú triedu určený z experimentálnych jazykov je 1,75. Najvyššiu nameranú hodnotu priemerného počtu anotácií na jednu doménovú triedu spomedzi experimentálnych jazykov má jazyk stavových automatov (SML). Je to z dôvodu definície väčšieho množstva lexikálnych symbolov pre oddeľovanie a určovanie jazykových konštrukcií. Príkladom je časť vety z tohto jazyka, ktorá definuje prechod medzi dvoma stavmi a akciu, ktorá sa má vykonať.

```
transition from Ready to Running when water_high
```

Hrubým písmom sú zvýraznené lexikálne symboly, ktoré slúžia na oddelenie konštrukcií ale pre sémantiku vety nemajú význam. Na druhej strane ale použitie týchto symbolov

navodzuje podobnosť s prirodzeným jazykom, čo môže znamenať jednoduchšie (intuitívnejšie) pochopenie vety z jazyka.

Graf na Obr. 51 zobrazuje celkový prehľad počtu doménových tried, ktoré obsahujú elementy označené anotáciami a počtu doménových tried bez anotácií pre jednotlivé experimentálne jazyky.



Obr. 51: Počet typov s anotáciami a bez anotácií

V Tab. 23 sú uvedené počty definícii lexikálnych symbolov. Táto tabuľka obsahuje taktiež výsledky meraní charakterizujúce generované zdrojové kódy. Prvým meraným údajom charakterizujúcim generovaný jazykový procesor je počet pravidiel pre konkrétnu syntax zapísanú v EBNF. Ďalším meraným údajom je počet riadkov a celkový počet znakov v generovanom súbore pre nástroj JavaCC. Tento súbor je generovaný vytvoreným generátorom jazykových procesorov *YAJCo*. Posledným meraným údajom je počet riadkov a celkový počet znakov v súboroch, ktoré sú generované nástrojom JavaCC na základe vstupného súboru.

Druh	SAL	AL	SIL	PIL	GUIIL	SML	LAD
Počet definícií lexikálnych symbolov	8	16	37	40	7	11	37
Počet pravidiel v EBNF	5	7	24	27	5	6	29
Počet riadkov vygenerovaných z *YAJCo*	128	187	570	655	124	168	693
Počet znakov vygenerovaných z *YAJCo*	2458	3775	15912	17554	2854	4245	19559
Počet riadkov vygenerovaných z JavaCC	1487	1603	2567	2580	1516	1843	3849
Počet znakov vygenerovaných z JavaCC	42103	45370	78450	78687	43869	52950	114002

Tab. 23: Charakteristika generovaných kódov

Literatúra

[1] AHO, A. V. – LAM, M. S. – SETHI, R. – ULLMAN, J. D.: *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison Wesley, 2006. 1000 p. ISBN 978-0321486813.

[2] ALLISON, L.: *A Practical Introduction to Denotational Seman*tics. Cambridge University Press, 1987. 148 p. ISBN 0521314232.

[3] ARMSTRONG, J: *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. 536 p. ISBN 978-1934356005.

[4] BĚHÁLEK, M. – ŠALOUN, P.: *Paralelization of Process Functional Language*. In: Proceedings of ECI'2006, 7-th International Scientific Conf. on Electronic Computers and Informatics, Košice-Herľany, Slovakia, September 20-22, FEII TU Košice, 2006, p. 168-173, ISBN 80-8073-150-0.

[5] BĚHÁLEK, M. – ŠALOUN, P.: *Parallel Process Functional Language*. In: SOFSEM 2007 SRF, Prague:INSTITUTE of Computer Science AS CR, 2007, p. 1-13, ISBN 80-903298-9-6.

[6] BĚHÁLEK, M.: *Zadanie z predmetu Programovací jazyky a překladače – štruktúrovaný imperatívny jazyk*. FEI VŠB-TU Ostrava, 2008, http://www.cs.vsb.cz/behalek/vyuka/pjp/projekt2/popis.php, prístupné 15.5.2008.

[7] BENTLEY, J. L.: *Programming pearls: Little languages*. Comm. ACM 29, 8 (1986), p. 711–721.

[8] BERGIN, T. J. – GIBSON, R. G., ed.: *History of Programming Languages II*. ACM Press, 1996.

[9] BEZEM, M. et al.: *Term Rewriting Systems*. Cambridge University Press, 2003. 884 p. ISBN 0521391156.

[10] BIRD, R. J. – WADLER, P.: *Introduction to Functional Programming*. Prentice Hall International, 1988, 300 p.

[11] BOUDREAU, T. – GLICK, J. – GREENE, S. – WOEHR, J. – SPURLIN, V.: *NetBeans: The Definitive Guide*. O'Reilly Media, Inc., 2002. 672 p. ISBN 978-0596002800.

[12] BRAINERD, W. S. – LANDWEBER, L. H.: *Theory of Computation*. Wiley, 1974. ISBN 0471095850.

[13] CEPA, V.: *Attribute Enabled Software Development*. VDM Verlag Dr. Muller e.K., 2007. 216 p. ISBN 3836410168.

[14] CEPA, V. – KLOPPENBURG, S.: *Representing Explicit Attributes in UML*. In: 7th International. Workshop on Aspect-Oriented Modeling (AOM), 2005.

[15] CEPA, V. – MEZINI, M.: *Declaring and enforcing dependencies between .NET custom attributes*. In G. Karsai and E. Visser, editors, Lecture Notes in Computer Science 3286, Springer, 2004. p. 283–297.

[16] COOK, S. – JONES, G. – KENT, S. – WILLS, A. C.: *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007, 576 p.

[17] CZARNECKI, K. – EISENECKER, U.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000. 864 p. ISBN 978-0201309775.

[18] DYBVIG, K. R.: The *Scheme Programming Language*. 3rd Edition. Prentice Hall, 2003. 307 p. ISBN 978-0262541480.

[19] DEGROOT, D. – LINDSTROM, G.: *Functional and Logic Programming*. Prentice Hall, 1985.
[20] DIOS, Y. C. – MILI, R. – LAN WU, A. – WANG, K.: *An empirical study of programming language trends*. IEEE Software 22(3), 2005, p. 72- 79, ISSN 0740-7459.
[21] DMITRIEV, S.: *Language Oriented Programming: The Next Programming Paradigm*. http://www.onboard.jetbrains.com/is1/articles/04/10/lop, prístupné 15.5.2008.
[22] DONNELLY, C. – STALLMAN, R.: *Bison: The Yacc-compatible Parser Generator*. 2006, http://www.gnu.org/software/bison/manual/pdf/bison.pdf, prístupné 15.5.2008.
[23] EASTWOOD, A.: *Firm Fires Shots at Legacy Systems*. Computing Canada 19 (2), 1993, p. 17.
[24] ELRAD, T. – FILMAN, R. E. – BADER, A.: *Aspect-oriented programming: Introduction*. Commun. ACM 44(10), 2001. p. 29–32.
[25] ERLIKH, L.: *Leveraging Legacy System Dollars for E-Business*. In: IT Professional. 2000, p. 17-23. ISSN 1520-9202.
[26] EVJEN, B. et al.: *Professional XML*. Wrox, 2007, 856 p. ISBN 0471777773.
[27] FELLBAUM, C. (ed.): *WordNet: An Electronic Lexical Database*. The MIT Press, 1998. 423 p. ISBN 026206197X.
[28] FIELD, A. J. – HARRISON, P. G.: *Functional Programming*, Addison-Wesley, 1988.
[29] FILMAN, R. E. – ELRAD, T. – CLARKE, S. – AKSIT, M.: *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004. 800 p. ISBN 978-0321219763.
[30] FORMAN, I. R. – FORMAN, N.: *Java Reflection in Action*. Manning Publications, 2004. 300 p. ISBN 1932394184.
[31] FOWLER, M.: *Inversion of Control Containers and the Dependency Injection pattern*. 2004. http://martinfowler.com/articles/injection.html, prístupné 15.5.2008.
[32] FOWLER, M.: *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. http://www.martinfowler.com/articles/languageWorkbench.html, prístupné 15.5.2008.
[33] FOWLER, M.: *Domain Specific Languages*. 2008, http://martinfowler.com/dslwip, prístupné 15.5.2008.
[34] FRANKEL, D. S.: *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003. 352 p. ISBN 978-0471319207.
[35] FREEMAN, S. – PRYCE, N.: *Evolving an Embedded Domain-Specific Language in Java*. In: OOPSLA'06 October 22-26, 2006.
[36] GOSLING, J. – JOY, B. – STEELE, G. – BRACHA, G.: *The Java Language Specification. Third Edition*. Addison-Wesley, 2005. 688 p. ISBN 0321246780.
[37] GRADECKI, J. D. – LESIECKI, N.: *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003. 456 p. ISBN 0471431044.
[38] GRAND, M.: *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*. 2nd Edition, Volume 1. Wiley, 2002. 544 p. ISBN 978-0471227298.
[39] GREENFIELD, J. – SHORT, K. – COOK, S. – KENT, S. – CRUPI, J.: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004. 500 p. ISBN 0471202843.

[40] GRUNE , D. – JACOBS, C. J. H.: *Parsing Techniques: A Practical Guide*. 2nd Edition, Springer, 2007. 662 p. ISBN 978-0387202488.
[41] HARROP, J.: *F# for Scientists*. Wiley-Interscience, 2008. 368 p. ISBN-13: 978-0470242117.
[42] HENDERSON, P.: *Functional Programming: Application and Implementation*. Prentice-Hall, Englewood Cliffs, N.J., 1980.
[43] HOPCROFT, J. E. – MOTWANI, R. – ULLMAN, J. D.: *Introduction to Automata Theory, Languages, and Computation*. 3rd Edition, Addison Wesley, 2006. 750 p. ISBN 978-0321455369.
[44] HUDAK, P.: *Conception, evolution, and application of functional programming languages*. ACM Computing Surveys 21 (3), 1993, p. 359–411.
[45] HUDAK, P.: Building domain-specific embedded languages. ACM Computing Surveys, Vol. 28, Issue 4, 1996, ISSN 0360-0300.
[46] CHURCH, A.: *The Calculi of Lambda Conversion*. Princeton University Press, 1941, reprinted in 1963 by University Microfilms Inc., 1963.
[47] *International Conference on Software Language Engineering* (SLE), http://planet-sl.org/sle2008, 2008, prístupné 15.5.2008.
[48] JARZABEK, S.: Effective Software Maintenance and Evolution A Reuse-Based Approach. AUERBACH, 2007, 424 pp. ISBN 0849335922.
[49] JOHNSON, S. C.: *YACC: Yet Another Compiler-Compiler*. Unix Programmer's Manual Volume 2b, 1979, http://www2.informatik.uni-erlangen.de/Lehre/WS200304/Compilerbau/Uebungen/yacc.pdf.
[50] JONES, S. L. P.: *The Implementation of Functional Programmnig Languages*, Prentice Hall, 1987, 445 p.
[51] JONES, S. L. P. (ed.): *Haskell 98 Language and Libraries: The Revised Report*. 2002, http://haskell.org/onlinereport/, prístupné 15.5.2008.
[52] KELLY, S. – LYYTINEN, K. – ROSSI, M.: *MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment*. In: Proceedings of CAiSE'96, 8th Intl. Conference on Advanced Information Systems Engineering, Lecture Notes in Computer Science 1080, Springer-Verlag, p. 1-21, 1996.
[53] KLEPPE, A.: *A Language Description is More than a Metamodel*. In: the 4th International Workshop on (Software) Language Engineering. 2007.
[54] KLINT, P. – LÄMMEL, R. – VERHOEF, C.: *Toward an engineering discipline for grammarware*. In: ACM Transactions on Software Engineering and Methodology, Vol. 14, Issue 3, 2005, p. 331-380. ISSN 1049-331X.
[55] KNUTH, D. E.: *Semantics of context-free languages*. In: Journal Theory of Computing Systems, Vol. 2, No. 2, 1968, Springer New York, p. 127-145, ISSN 1432-4350.
[56] KNUTH, D. E.: *The genesis of attribute grammars*. In: Proceedings of the international conference on Attribute grammars and their applications, 1990, p. 1–12.
[57] KOLLÁR, J.: *Funkcionálne programovanie*. Elfa, 1995. ISBN 80-88786-14-2.
[58] KOLLÁR, J.: Process *Functional Programming*. In: Proc. 33rd Spring International Conference MOSIS'99 - ISM'99, Information Systems Modeling, Rožnov pod Radhoštem, Czech Republic, ACTA MOSIS No. 74, 1999, p. 41-48.
[59] KOLLÁR, J.: Process Functional Arrays. In: Proc. of EMES'99 Conference on Engeneering Modern Electric Systems, Felix-Spa, Romania, 1999. p. 201-206, ISSN-1223-2106.

[60] KOLLÁR, J.: PFL Expressions for Imperative Control Structures, Proc. of Computer Engineering and Informatics Scientific Conference, Herlany, Slovakia, 1999. p. 23-28. ISBN 80-88922-05-4.
[61] KOLLÁR, J.: Comprehending Loops in a Process Functional Programming Language. Computers and AI 19, 2000. p. 373-388.
[62] KOLLÁR, J.: Object Modelling using Process Functional Paradigm. In: Proc. 34th Spring International Conference Information Systems Modelling, Rožnov pod Radhoštem, Czech Republic, ACTA MOSIS No. 80, 2000, p. 203-208, ISBN 80-85988-45-3.
[63] KOLLÁR, J.: Control-driven Data Flow. Journal of Electrical Engineering 51 (3-4), 2000, p. 67-74.
[64] LADDAD, R.: *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003. 512 p. ISBN 1930110936.
[65] LEDECZI A. et al.: *GME 2000 Users Manual (v2.0)*, 2001, http://www.isis.vanderbilt.edu/sites/default/files/GMEUMan.pdf, prístupné 15.5. 2008.
[66] LEDECZI A. et al.: *Composing Domain-Specific Design Environments*. In. Computer Vol. 34, Issue 11, p. 44-51, November, 2001, ISSN 0018-9162.
[67] LEENHEER, P. – MENS, T.: *Ontology Evolution: State of the art and future directions*. In: Ontology Management: Semantic Web, Semantic Web Services, and Business Applications, Springer, 2008, p. 131-176. ISBN 038769899.
[68] LEHMAN, M. M.: *Approach to a Theory of Software Evolution*. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution. 2005, p. 135. ISSN 1550-4077.
[69] LEHMAN, M. M. - RAMIL, J. F.: *Software Evolution and Software Evolution Processes*. In: Annals of Software Engineering. 2002, s. 275-309. ISSN 1022-7091.
[70] LENZ, G. – WIENANDS, C.: *Practical Software Factories in .NET*. Apress, 2006. 240 p. ISBN 159059665X.
[71] LISKOV, B.: *Data Abstraction and Hierarchy*. ACM SIGPLAN Notices, 23 (5): p. 17-34, 1987.
[72] LISKOV, B. – WING, J. M.: *Behavioural subtyping using invariants and constraints*. In: Formal methods for distributed processing: a survey of object-oriented approaches, Cambridge University Press, 2001. p. 254 - 280.
[73] LOCHMANN, H. – BRAEUER, M.: *Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations*. In: the 4th International Workshop on (Software) Language Engineering. 2007.
[74] LUBERS, M. – POTTS, C. – RICHTER, C.: *A Review of the State of the Practice*. In: Requirements Modeling, Proc. International Requirements Engineering Symposium. IEEE Computer Society Press, 1993, p. 2-14.
[75] MARTIN, J.: *Fourth-Generation Languages. Vol. I: Principles, Vol II: Representative 4GLs*. Prentice-Hall, 1985.
[76] *Merriam-Webster's Collegiate Dictionary*, http://www.merriam-webster.com, prístupné 15.5.2008.
[77] MENS, T. – VAN GORP, P.: *A Taxonomy of Model Transformation*. In: Electronic Notes in Theoretical Computer Science, Vol. 152, No. 27, 2006, p. 125-142.
[78] MENS, T. – DEMEYER, S.: *Introduction and Roadmap: History and Challenges of Software Evolution*. In: Software Evolution. Springer, 2008, p. 1-11. ISBN 978-3-540-76439-7.

[79] MENS, T. – WERMELINGER, M. – DUCASSE, S. – DEMEYER, S. – HIRSCHVELD, R. – JAZAYERI, M.: *Challenges in software evolution*. In: Proc. of International Workshop on Principles of Software Evolution (IWPSE 2005), 2005, p. 13-22. ISSN 1550-4077.

[80] MENS, T. – TOURWE, T.: *A survey of software refactoring*. In: IEEE Transactions on Software Engineering, Vol. 30, Issue 2, p. 126-139, 2004.

[81] MENS, T. – DEMEYER, S. – DU BOIS, B. – STENTEN, H. – VAN GORP, P.: *Refactoring: Current research and future trends*. In: Third Workshop on Language Descriptions, Tools and Applications (LDTA), Satellite event of ETAPS, Warsaw, April 6th, 2003.

[82] MERNIK, M. – HEERING, J. – SLOANE, A. M.: *When and How to Develop Domain-Specific Languages*. ACM Computing Surveys, Vol. 37, No. 4, December 2005, p. 316–344.

[83] MEYER, B.: *Reality: a cousin twice removed*. In: Computer, Vol. 29, Issue 7, 1996. p. 96-97. ISSN 0018-9162.

[84] MEYER, B.: *Object-Oriented Software Construction*. 2nd Edition, Prentice Hall PTR, 2000. 1296 p. ISBN 978-0136291558.

[85] MILNER, R. – TOFTE, M. – HARPER, R. – MACQUEEN, D.: *The Definition of Standard ML - Revised*. The MIT Press, 1997. 128 p. ISBN 978-0262631815.

[86] MITCHELL, J. C.: *Concepts in Programming Languages*. Cambridge University Press, 2003, 529 p. ISBN 0521780985.

[87] NARDI, B. A.: *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.

[88] NIELSON, H. R. – NIELSON, F.: *Semantics with Applications: A Formal Introduction*. John Wiley & Sons Inc, 1992. 252 p. ISBN 0471929808.

[89] NOGUERA, C. – DUCHIEN, L.: *Annotation Framework Validation Using Domain Models*. In: Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications, Springer-Verlag, 2008. p. 48 - 62.

[90] NOGUERA, C. – PAWLAK, R.: *AVal: an Extensible Attribute-Oriented Programming Validator for Java*. In: Journal of Software Maintenance and Evolution: Research and Practice, Volume 19 , Issue 4, John Wiley & Sons, 2007, p. 253 - 275.

[91] NOVITZKÁ, V.: *Sémantika programov*. Košice : Elfa, 2001. 145 p. ISBN 80-88964-59-8.

[92] PARNAS, D. L.: *Software aging*. In Proc. International Conference on Software Engineering, IEEE Computer Society Press, p. 279–287. 1994.

[93] PARREIRAS, F. S. – STAAB, S. – WINTER, A.: *On Marrying Ontological and Metamodeling Technical Spaces*. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM, 2007, p. 439-448. ISBN 978-1-59593-811-4.

[94] PARR, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007, 361 p. ISBN 0978739256.

[95] PFLEEGER, S. L.: *Software Engineering: Theory and Practice*, 2nd Edition. Prentice Hall, 2001, 659 p. ISBN 0130290491.

[96] PLASMEIJER, R. – EEKELEN, M.: *Clean Version 2.0 Language Report*. University of Nijmegen 2001, http://clean.cs.ru.nl, prístupné 15.5.2008.

[97] PLOTKIN, G. D.: *A Structural Approach to Operational Semantics*. In J. Log. Algebr. Program. 60-61, p. 17-139, 2004 (reprint).
[98] *Programming Language Popularity*, http://www.langpop.com, prístupné 15.5.2008.
[99] RUMBAUGH, J. – JACOBSON, I. – BOOCH, G.*: The Unified Modeling Language Reference Manual*. 2nd Edition. Addison-Wesley Professional, 2004. 752 p. ISBN 78-0321245625.
[100] ROUVOY, R. – MERLE, P.: *Leveraging Component-Oriented Programming with Attribute-Oriented Programming*. In: Proc. of the 11th ECOOP International Workshop on Component-Oriented Programming, 2006.
[101] SABRY, A.: *What is a Purely Functional Language?* Journal of Functional Programming 8:1, p. 1-22, Jan. 1998.
[102] SAMMET, J. E.: *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969.
[103] SEBESTA, R.W.: *Concepts of Programming Languages*. 8th Edition, Addison Wesley, 2007. 752 p. ISBN 978-0321493620.
[104] STAHL, T. – VOELTER, M.: *Model-Driven Software Development: Technology, Engineering*, Management. Wiley, 2006. 444 p. ISBN 0470025700.
[105] STEIMANN, F.: *The paradoxical success of aspect-oriented programming*. In: ACM SIGPLAN Notices, Vol. 41, Issue 10, 2006, p. 481-497, ISSN 0362-1340.
[106] STOY, J. E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1981. 414 p. ISBN 0262690764.
[107] SYME, D. – GRANICZ, A. – CISTERNINO, A.: *Expert F#*. Apress, 2007. 609 p. ISBN 978-1590598504.
[108] Sun Microsystems, Inc.: *Java Platform*. 2008, http://java.sun.com/javase/6/docs/, prístupné 15.5.2008.
[109] Sun Microsystems, Inc.: *Java Platform - Annotation Type PostConstruct*. 2008, http://java.sun.com/javase/6/docs/api/javax/annotation/PostConstruct.html, prístupné 15.5.2008.
[110] SZYPERSKI, C.: *Component Software: Beyond Object-Oriented Programming*. 2nd Edition, Addison-Wesley Professional, 2002. 624 p. ISBN 0201745720.
[111] *The Catalog of Compiler Construction Tools*. Fraunhofer Institute For Computer Architecture and Software Technology, 2006, http://catalog.compilertools.net/, prístupné 15.5.2008.
[112] THOMPSON, S.: *Haskell: The Craft of Functional Programmin.*, 2nd Edition. Addison Wesley, 1999. 528 p. ISBN 0201342758.
[113] *TIOBE Programming Community Index*, http://www.tiobe.com/tiobe_index/index.htm, prístupné 15.5.2008.
[114] TOLVANEN, J. P. – POHJONEN, R. – KELLY, S.: *Advanced Tooling for Domain-Specific Modeling: MetaEdit+*. In: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling, Montreal, Canada, 2007.
[115] TOMITA, M.: *LR parsers for natural languages*. In: 10th International Conference on Computational Linguistics, 1984, p. 354-357.
[116] TOMITA, M.: *An efficient context-free parsing algorithm for natural languages*. In International Joint Conference on Artificial Intelligence, 1985. p. 756-764.
[117] TURNER, D.: *An Overview of Miranda*. Research Topics in Functional Programming, Addison Wesley, 1990.
[118] WADA, H. – SUZUKI, J.: *An Introduction to Attribute-Oriented Programming*. University of Massachusetts, Boston, 2005,

http://dssg.cs.umb.edu/resources/attribute-oriented-programming.html, prístupné 15.5.2008.

[119] WADA, H. – SUZUKI, J.: *Modeling Turnpike Frontend System: a Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming*. In: Proc. of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005), 2005.

[120] WADLER, P.: *Monads for functional programming*. Lecture Notes In Computer Science 925, 1995. p. 24-52, ISBN 3-540-59451-5.

[121] WADLER, P.: *Lazy vs. strict*. ACM Computing Surveys 28 (2), 1996, p. 318-320. ISSN 0360-0300.

[122] WADLER, P. – THIEMANN, P.: *The marriage of effects and monads*. ACM Transactions on Computational Logic 4 (1), 2003, p. 1-32.

[123] WARD, M. P.: *Language Oriented Programming*. In. Software - Concepts and Tools, Vol. 15, No.4, 1994, p. 147-161.

[124] WEXELBLAT, R. L. ed.: *History of Programming Languages*. Academic Press, 1981.

[125] *XML Path Language (XPath) 2.0*. W3C Recommendation, 2007. http://www.w3.org/TR/xpath20/, prístupné 15.5.2008.

[126] YANG, H. – WARD, M.: *Successful Evolution of Software Systems*. Artech House Publishers, 2003, 300 p. ISBN 1580533493.

Relevantné práce autora

[P1] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: From eager PFL to lazy Haskell. In: Computing and Informatics. Vol. 25, No. 1 (2006), pp. 61-80. ISSN 1335-9150.

[P2] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Separating concerns in programming: data, control and actions. In: Computing and informatics. Vol. 24, No. 5 (2005), pp. 441-462. ISSN 1335-9150.

[P3] PORUBÄN, Jaroslav - SAMUELIS, Ladislav - VÁCLAVÍK, Peter: Object-form bridge: A framework for supporting seamles software development. In: Journal of Information, Control and Management Systems. Vol. 5, No. 2/2 (2007), pp. 323-330. ISSN 1336-1716.

[P4] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter – BANDÁKOVÁ, Jana – FORGÁČ, Michal: Software Evolution from a Meta-Level Compiler Perspective. In: SCIENCE & MILITARY, Vol. 2, No. 2 (2007), pp. 29-32, ISSN 1336-8885.

[P5] KOLLÁR, Ján - FORGÁČ, Michal - PORUBÄN, Jaroslav: Adaptiveness of software systems using reflection. In: Acta Electrotechnica et Informatica. Vol. 7, No. 3 (2007), pp. 53-57. ISSN 1335-8243.

[P6] VÁCLAVÍK, Peter - PORUBÄN, Jaroslav: Type environments in object oriented process functional language. In: Acta Electrotechnica et Informatica. Vol. 6, No. 4 (2006), pp. 65-72. ISSN 1335-8243.

[P7] KOLLÁR, Ján - HAVLICE, Zdeněk - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Evaluation of resources by abstract interpretation. In: Journal of information, control and management systems. Vol. 3, No. 1 (2005), pp. 27-34. ISSN 1336-1716.

[P8] KOLLÁR, Ján - VÁCLAVÍK, Peter - PORUBÄN, Jaroslav: The classification of programming environments. In: Acta Universitatis Matthiae Belii. No. 10 (2003), pp. 51-64. ISBN 80-8055-662-8.

[P9] KOLLÁR, Ján - PORUBÄN, Jaroslav: Building Adaptive Language Systems. INFOCOMP - Journal of Computer Science, Vol. 7, No. 1 (2008), pp. 1-10, ISSN 1807-4545.

[P10] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter – BANDÁKOVÁ, Jana – FORGÁČ, Michal: Functional Approach to the Adaptation of Languages instead of Software Systems. In: Computer Science and Information Systems, Vol. 4, No. 2 (2007), pp. 115-129, ISSN 1820-0214.

[P11] KOLLÁR, Ján - NOVITZKÁ, Valerie - VÁCLAVÍK, Peter - PORUBÄN, Jaroslav: Process functional form of imperative programs. In: Romanian Journal of Information Science and Technology. vol. 8, no. 2 (2005), p. 99-114. ISSN 1453-8245.

[P12] BAČA, Ján - KOREČKO, Štefan - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Didactic version of program system for synthesis and diagnostics of logic circuits. In: Problemy programmirovanija. No. 3 (2003), p. 53-58. ISSN 1727-4907.

[P13] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Time and memory profile of a process functional program. In: Acta Polytechnica Hungarica. Vol. 3, No. 2 (2006), pp. 27-40. ISSN 1785-8860.

[P14] PORUBÄN, Jaroslav: Time and Space Execution Profiling for Process Functional Language. Analele Universitatii din Oradea, University of Oradea Romania -

Faculty of Electrotechnics and Informatics, Romania, Oradea May 2003, Vol. 1, pp. 167-172, ISSN 1223-2106.

[P15] PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Generating Software Language Parser from Domain Classes, Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering, The High Tatras - Stará Lesná, Slovakia, September 24-26, 2008, Košice, Department of Computers and Informatics FEEI TU Košice, 2008, 1, 1, pp. 133-140, ISBN 978-80-8086-092-9.

[P16] FORGÁČ, Michal - KOLLÁR, Ján - PORUBÄN, Jaroslav: Reflection as a Tool for Adaptability of Software Systems, SAMI 2008 Proceedings, 6th International Symposium on Applied Machine Intelligence and Informatics, Herľany, Slovakia, January 21-22, 2008, 6, pp. 179-182, ISBN 978-1-4244-2106-0.

[P17] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter - BANDÁKOVÁ, Jana, FORGÁČ, Michal: Adaptive Compiler Infrastructure, Komunikačné a informačné technológie, Tatranské Zruby, 3-5. okt., 2007, pp. 4-5, ISBN 978-80-8040-324-9.

[P18] PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Inheritance Profiles of Process Functional Programs, Proceedings of the 7-th International Scientific Conference Electronic Computers and Informatics ECI 2006, Košice - Herľany, September 20-22, 2006, Košice, 2006, 7, pp. 192-197, ISBN 80-8073-598-0.

[P19] VÁCLAVÍK, Peter - PORUBÄN, Jaroslav: Objects in Functional Languages, Proceedings of the 7-th International Scientific Conference Electronic Computers and Informatics ECI 2006, Košice - Herľany, September 20-22, 2006, Košice, 2006, 7., pp. 198-203, ISBN 80-8073-598-0.

[P20] VÁCLAVÍK, Peter - KOLLÁR, Ján - PORUBÄN, Jaroslav – VIDIŠČAK, Miroslav: Compiling the Process Functional Programs, Proceedings of the 3rd Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence, Herľany, Slovakia, January 21-22, 2005, 2005, pp. 283-296, ISBN 963 7154 35 3.

[P21] PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Process functional program profiling, Proceedings of ECI'2004, 6-th International Scientific Conf. on Electronic Computers and Informatics, Košice-Herľany, Slovakia, September 22-24, 2004, Košice, VIENALA Press, 2004, pp. 139-144, 80-8073-150-0.

[P22] VÁCLAVÍK, Peter - PORUBÄN, Jaroslav: Process functional language compiler architecture, Proceedings of ECI'2004, 6-th International Scientific Conf. on Electronic Computers and Informatics, Košice-Herľany, September 22. - 24., 2004, Košice, VIENALA Press Košice, 2004, pp. 145-150, ISBN 80-8073-150-0.

[P23] VÁCLAVÍK, Peter - PORUBÄN, Jaroslav: Object Oriented Approach in Process Functional Language, Proceedings of the Fifth International Scientific Conference „Electronics Computers and Informatics´2002", Košice-Herľany, October 10-11, 2002, Košice - Herľany, 2002, pp. 92-96, ISBN 80-7099-879-2.

[P24] PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Simulácia používania zdrojov pri vykonávaní procesného funkcionálneho programu, Proceeding of the International Workshop Modeling and Simulation in Management, Inofrmatics and Control MOSMIC 2001, Október 9-10, 2001, Žilina, University of Žilina, 2001, pp. 133-140, ISBN 80-7100-883-4.

[P25] PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Extensible Language Independent Source Code Refactoring, AEI'2008 International Conference on Applied Electrical Engineering and Informatics, Athens, Greece, September 8-11, 2008, in printing.

[P26] VÁCLAVÍK, Peter - PORUBÄN, Jaroslav: Template-Based Content Management System, AEI'2008 International Conference on Applied Electrical Engineering and Informatics, Athens, Greece, September 8-11, 2008, in printing.

[P27] PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Separating User Interface and Domain Logic, Analele Universitatii din Oradea, Proc. 8th International Conference on Engineering of Modern Electric Systems, Oradea, May 24 - 26, University of Oradea, Romania, 2007, pp. 90-95, ISSN 1223-2106.

[P28] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter: Evolutionary Nature of Crosscutting Modularity, Proc. 8th International Conference on Engineering of Modern Electric Systems, Felix Spa-Oradea, May 24 - 26, Oradea, Romania, University of Oradea, 2007, pp. 43-48, ISSN 1223-2106.

[P29] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter - BANDÁKOVÁ, Jana - FORGÁČ, Michal: Adaptive Language Approach to Software Systems Evolution, International Multiconference on Computer Science and Information Technology: 1st Workshop on Advances in Programming Languages (WAPL'07), Wisla, Poland, October 15-17, Polish Information Processing Society, 2007, 2, pp. 1081-1091, ISSN 1896-7094.

[P30] PORUBÄN, Jaroslav - VÁCLAVÍK, Peter - KOLLÁR, Ján: The Essence of Process Functional Language, Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages IFL'06, Budapest, September 4–6, 2006, Budapest, Hungary, Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers, 2006, pp. 132-144, ISBN: 963 463 876 7.

[P31] PORUBÄN, Jaroslav - KOLLÁR, Ján - Vidiščak Miroslav: Aspect-oriented Program Profiling, Analele Universitatii din Oradea, Proc. 8th International Conference on Engineering of Modern Electric Systems, Felix Spa-Oradea, May 26 - 28, University of Oradea, Romania, 2005, pp. 112-117, ISSN 1223-2106.

[P32] VÁCLAVÍK, Peter - KOLLÁR, Ján - PORUBÄN, Jaroslav: Object-oriented Programming with Functional Language, 8'th International Conference ISIM'05, Hradec nad Moravicí, Czech Republic, April 19 - 20, 2005, pp. 167-174, ISBN 80-86840-09-3.

[P33] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter – VIDIŠČAK, Miroslav: Lazy State Evaluation of Process Functional Programs, Proceedings of the Conference "Information Systems Modelling" ISM'02, Rožnov pod Radhoštěm, April 22-24, Ostrava, MARQ, 2002, ACTA MOSIS, 87, pp. 165-172, ISBN 80-85988-70-4.

[P34] KOLLÁR, Ján - PORUBÄN, Jaroslav: Static evaluation of process functional programs, Proceeding of the 6-th International Conference: Engineering of Modern Electric Systems, Oradea, Romania, May 24-26, Oradea, University of Oradea, 2001.

[P35] PORUBÄN, Jaroslav: Process functional program profiling, Košice, Department of Computers and Informatics, TU FEI Košice, PhD thesis, 2004, 87 pp.

[P36] KOLLÁR, Ján, PORUBÄN, Jaroslav, VÁCLAVÍK, Peter, Tóth Marcel, BANDÁKOVÁ, Jana, FORGÁČ, Michal: Multi-paradigm Approaches to Systems Evolution, Computer Science and Technology Research Survey, Košice, Elfa s.r.o., 2007, 1, pp. 6-10, ISBN 978-80-8086-046-2

[P37] KOLLÁR, Ján - PORUBÄN, Jaroslav - VÁCLAVÍK, Peter - BANDÁKOVÁ, Jana, FORGÁČ, Michal: How to Adapt Programming Languages instead of Software

Systems, Computer Science and Technology Research Survey, Košice, Elfa, 2007, 2, pp. 69-79, ISBN 978-80-8086-071-4.

Zoznam obrázkov

Zoznam tabuliek

Názov: Vývoj doménovo-špecifických jazykov
Autor: Jaroslav Porubän
Vydavateľ: Technická univerzita v Košiciach
Rok: 2020
Vydanie: prvé
Náklad: 50 ks
Rozsah: 114
ISBN 978-80-553-3505-6