UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO
2000 Maribor, Smetanova ul. 17

Diplomsko delo

IMPLEMENTACIJA DOMENSKO SPECIFIČNIH JEZIKOV Z RAZŠIRJANJEM ODPRTOKODNIH PREVAJALNIKOV

David Krmpotić

Maribor, september 2005

UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO, RAČUNALNIŠTVO IN INFORMATIKO
2000 Maribor, Smetanova ul. 17

Diplomsko delo univerzitetnega študijskega programa

IMPLEMENTACIJA DOMENSKO SPECIFIČNIH JEZIKOV Z RAZŠIRJANJEM ODPRTOKODNIH PREVAJALNIKOV

Študent: David Krmpotić
Študijski program: univerzitetni, Računalništvo in informatika
Smer: Programska oprema

Mentor: izred. prof. dr. Marjan Mernik

Maribor, september 2005

SKLEP O DIPLOMSKI NALOGI
(fotokopija)

ZAHVALA

Zahvaljujem se mentorju izred. prof. dr. Marjanu Merniku za pomoč in vodenje pri opravljanju diplomskega dela in za koristne nasvete med študijem. Prav tako se zahvaljujem asistentu g. Tomažu Kosarju za njegovo pomoč.

Za nasvete v zadnjem trenutku sem hvaležen g. Mateju Črepinšku. Za zanimive pogovore med študijem, ki niso bili niti malo povezani z računalništvom pa hvala predvsem g. Tadeju Gregorčiču in g. Boštjanu Kukovcu.

IMPLEMENTACIJA DOMENSKO SPECIFIČNIH JEZIKOV Z RAZŠIRJANJEM ODPRTOKODNIH PREVAJALNIKOV

Ključne besede: domensko specifični jeziki, prevajalniki, odprta koda

UDK: 004.43 (043.2)

Povzetek

Implementacija domensko specifičnih jezikov ni enostavno opravilo. To je eden glavnih razlogov, da do sedaj niso bili deležni večje pozornosti. Dostopnost izvorne kode prevajalnikov, v kolikor je dobro organizirana in jasna, predstavlja za načrtovalce jezika eno izmed možnosti integracije lastnih domensko specifičnih jezikov v splošno namenski jezik. Namen tega dela je predstaviti implementacijo domensko specifičnih jezikov s pomočjo dostopnih odprtokodnih prevajalnikov, oceniti potreben trud ter ga primerjati z ostalimi znanimi implementacijskimi pristopi. Razširitev je predstavljena na konkretnem primeru Mono C# prevajalnika in domensko specifičnega jezika Feature Definition Language.

IMPLEMENTATION OF DOMAIN SPECIFIC LANGUAGES WITH EXTENSION OF OPEN-SOURCE COMPILERS

Key words: domain specific languages, compilers, open-source

UDK: 004.43 (043.2)

Abstract

Domain specific language implementation is in general not an easy task. This is the main reason that kept it from getting more attention and reaching the expectations of domain specific language researchers. The implementation of a domain specific language can be demanding, but also very rewarding. Once the hard work has been done, we can profit a lot from the effort invested. The purpose of this work is to introduce the construction of domain specific languages through the extension of open compilers. Their source code - available, well organized and clear - gives language designers a possibility to incorporate domain specific constructs into general-purpose languages by extending their compilers. In this work, the extension of the Mono C# compiler with the domain specific Feature Definition Language is presented.

VSEBINA

UPORABLJENE KRATICE

FDL – Feature Definition Language
BNF – Backus-Naur Form
DSL – Domain Specific language (domensko specifičen jezik)
IT - Informacijska tehnologija
LOC - Lines Of Code
eLOC - Effective Lines Of Code

1 Uvod

Razvoj programskih jezikov je obsežno in zahtevno področje računalništva. Za različna področja uporabe so se razvili različni programski jeziki. Domensko specifični jeziki so ponavadi manjši, ozko specializirani jeziki z dobro definiranim področjem uporabe. Namen raziskave je oceniti trud, ki je potreben za implementacijo domensko specifičnih jezikov z razširitvijo odprtokodnih prevajalnikov, in ga primerjati z ostalimi znanimi implementacijskimi pristopi.

Začnemo z informativnim opisom dveh zapletenih sistemov – človeških možganov in elektronskega računalnika in pogledamo povezavo med njima. Za prikaz osnov delovanja računalnika je podan primer enostavnega programa in opisano delovanja prevajalnikov. Ta del je predstavljen na zelo enostaven in neformalen način z namenom, da bi ga lahko razumel tudi bralec, ki nima o delovanju prevajalnikov nobenega predznanja.

Nadalje je predstavljena osnovna teorija v ozadju delovanje prevajalnikov ter različni tipi programskih jezikov. Domensko specifični jeziki so podrobneje predstavljeni v naslednjem poglavju.

Sledi opis problema in predstavitev osnovnih entitet, ki pri tem nastopajo. Nadalje so prikazani koraki, potrebni za implementacijo izbranega domensko specifičnega jezika v izbran odprtokodni prevajalnik.

Izsledki raziskave in sklep so predstavljeni v zaključku dela.

2 Računalnik in možgani

The body is an instrument, the mind its function, the witness and reward of its operation.

-- George Santayana

2.1 Podobnosti in razlike

Človeške možgane je ustvarjala evolucija skozi milijone let, računalnik pa je ustvaril ravno ta čudoviti biološki *računalnik* – možgani. Človek je ena izmed redkih živali, ki uporablja orodja. Vsa orodja, ki jih je kdaj izumil, so morala služiti vsaj enemu namenu. Računalnik je posebne vrste orodje, ki služi *neskončno možnim* namenom. Računalnik je kompleksna naprava, katere ena uporabnost je razvijati še kompleksnejša orodja in s pomočjo njih človeku pomagati razumeti stvari, do katerih samo s svojimi možgani ne bi mogel priti – pa čeprav so ti sami po sebi najkompleksnejši sistem na Zemlji. Človek je s pomočjo računalnika dekodiral človeški genom, zaznal planete v oddaljenih osončjih, poslal nekaj satelitov v vesolje in pogledal v drobovje materije s pomočjo trkanja osnovnih delcev.

Računalnik ne razume popolnoma ničesar, vendar je sposoben izvajati računske procese. Možgani delujejo na višji ravni zavedanja in procesirajo veliko število informacij paralelno. Vprašanje pa je, če so tudi samo teoretično sposobni *razumeti* nekatera dejstva vesolja, ki mejijo na, oziroma *so* iracionalna. S štiri-dimenzionalnim prostorom [13] in Einsteinovo splošno relativnostjo se še nekako lahko sprijaznimo, s kvantno mehaniko pa se več ne moremo nikakor, čeprav je *najbolje preizkušena teorija v moderni fiziki*. Kaj pa s kombinacijo obeh, ki je potrebna, da bi razumeli, kaj se je dogajalo med začetkom vesolja in takrat, ko je bilo le-to staro »že« 10^{-43} sekunde (torej še manj kot milijardinko milijardinke milijardinke milijardinke sekunde)? 10^{-43} se imenuje *Planckov čas* in je najmanjši čas, ki ima sploh kakšen pomen v vesolju [18].

Interakcija človeških možganov z računalnikom ne bi bila možna brez dobro definiranega vmesnika. Funkcijo vmesnika med sistemoma opravljajo programski jeziki. Programski jeziki so abstraktni koncepti, ki definirajo veljavno sintakso in pomen programov. Programi so sami po sebi tudi abstrakten koncept, čeprav je to malo manj očitno.

Računalnik potrebuje za svoje delovanje programe in tako kot mu ti na zanimiv način omogočijo delovanje, podobno (ampak v bistvu samo zelo približno tako) tudi zavest »usmerja« možgane. Programi so neotipljivi, istočasno pa shranjeni na nekem mediju. Predstavljajmo si, da ta medij vzamemo iz računalnika – lahko rečemo, da je to še vedno program? Lahko, ampak samo, če se strinjamo, da samo mi tako mislimo. Program je torej tudi abstrakten koncept, ki v resnici obstaja samo v domišljiji programerja in računalnika. Pri ljudeh in živalih na podoben način ni ločnice med zavestjo in možgani, saj je ta v celoti povzročena z nevrobiološkimi procesi in realizirana v strukturi možganov [1].

Kot kaže računalniki z današnjo arhitekturo ne bodo nikoli mogli *resnično razumeti* ničesar. John Searlov miselni eksperiment »kitajske sobe« [3] nas poskuša prepričati o tem. V sobi z dvema luknjama je oseba, ki skozi eno luknjo prejme kitajske pismenke, opravi *rutinsko* proceduro in skozi drugo luknjo pošlje angleški prevod. Je oseba razumela kitajsko? Očitno je odgovor »*ne*«. Računalniki opravljajo samo rutinske operacije in zato ne morejo razumeti kitajščine pa tudi česa drugega ne.

Računalnik je samo pomočnik človeka, ki se ne zaveda lastnega obstoja. Računalnike smo povezali v mreže in ustvarili distribuirane sisteme za procesiranje podatkov. Možgani so povezani v distribuiran sistem že s pomočjo naravnih zakonov z namenom ustvarjanja realnosti. Čemu? Tega nihče ne ve in zelo malo verjetno je, da bi kdajkoli resnično izvedeli. Za zdaj kaže, da smo izgubljeni v skrivnostnem vesolju brez da bi imeli kakršen koli namen. Edini pomen je, da se na Zemlji odvijajo zraven striktno fizikalnih tudi izračunljivi in *neizračunljivi* procesi.

2.2 Človeški jezik

V naravi se živali ponavadi sporazumevajo z gibi teles, pri nekaterih sesalcih pa se je razvila tudi komunikacija s pomočjo glasu. Kiti grbavci in delfini uporabljajo zvok za lociranje objektov (t.i. *eholokacija*) in za medsebojno komuniciranje. Skoraj zagotovo najrazvitejša oblika komunikacije na Zemlji pa je človeški jezik.

Dinamika razvoja človeških jezikov je zelo zanimivo in obsežno področje. Človeški jezik se je razvijal vzporedno z možgani. Ne moremo reči, da smo razvili sposobnost razumevanja jezika zato, ker imamo tako razvite možgane. Po drugi strani pa tudi ne moremo trditi, da bi imeli tako

razvite možgane, če ne bi uporabljali in razvijali jezika skozi tisočletja. Jezik se prepleta s sposobnostjo ustvarjanja in razumevanja misli. Živali nimajo tako razvite zavesti, ker niso razvile sposobnosti govora.

Človek se kot otrok najprej nauči samostalnikov, saj le-te enostavno poveže z dejanskimi objekti iz okolja. V možganih se za vsako táko konkretno besedo ustvari neka miselna upodobitev – recimo slika. V zelo kratkem času se nauči veliko besed – vzpostavi veliko povezav med strukturami v možganih, ki preslikajo zvočne vzorce (besede) v miselne predstave.
Pridevnikov se naučimo tako, da nam nekdo pokaže *veliko* vazo in za tem še *majhno* vazo. Pod besedico »veliko« si tako predstavljamo nekaj, kar zasede več prostora od majhnega. Človeški možgani potem brez truda posplošijo uporabo nekega pridevnika na ostale samostalnike.

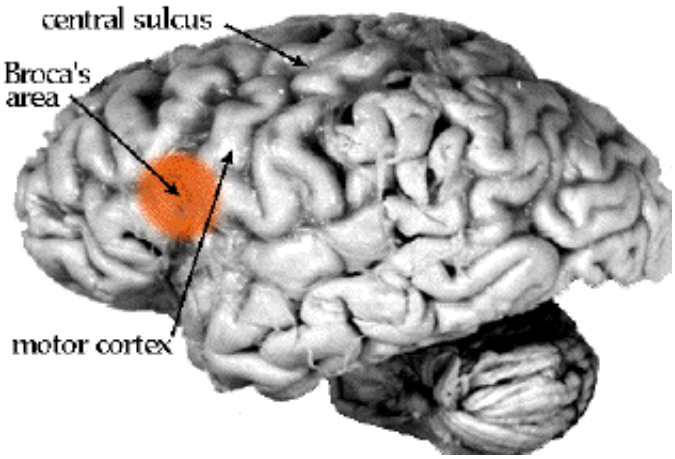
Živo bitje, ki si je sposobno predstavljati stvari iz narave, ustvarjati zapletene relacije med njimi in vplivati na okolje zaradi lastnih interesov ima *zavest*. Lastno zavest si sami težko predstavljamo, ker smo ujeti v njo, tuja zavest pa za nas ne obstaja. Iz naše perspektive je tuja zavest »samo« električni tok po sinapsah in nevronih, ki gradijo abstraktne strukture [17]. Zavest je samo *lastnost* možganov. Določen aspekt človeške zavesti se rodi iz vzajemnega procesa, ko se eni možgani trudijo razumeti druge, se jim prilagajajo in se skupaj razvijajo.

Pomembna razlika v dojemanju jezika človeških možganov in računalnika je tudi, da človeški možgani lahko besedo z več pomeni večinoma brez težav umestijo v širši kontekst in izberejo eno izmed možnih razlag (tudi glede na *emocionalno* stanje). Računalnik pa je deterministčen, kar zagotavlja ponovljivost in eliminira napake, kjer bi za iste podatke lahko dobili različen rezultat.

Zdi se, da so z jezikom povezani procesi skoncentrirani predvsem v levi hemisferi, čeprav pri celotnem procesiranju dejansko sodelujejo razni višje kognitivni podsistemi, katerih delovanje je porazdeljeno po raznih predelih možganov - nekako tako, kot pri ustvarjanju zavesti.

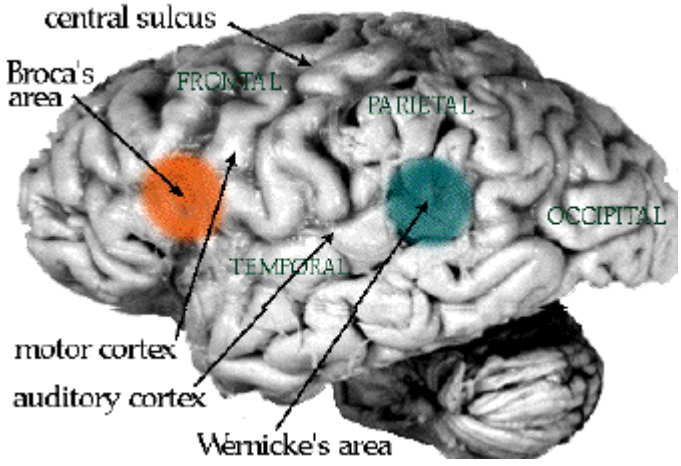
Za samo mehansko ustvarjanju govora je zadolžen poseben majhen predel možganov, ki je, kot bi lahko pričakovali, lociran v bližini področja, ki nadzoruje mišice ust in jezika. Predel se imenuje *Brocovo področje* po zdravniku, ki ga je odkril leta 1861 [14]. Broca je imel pacienta, ki

je po možganski kapi skoraj v celoti izgubil sposobnost govora. Pacient je razumel jezik, vendar je lahko izgovoril samo en zlog. Po njegovi smrti je Broca opravil avtopsijo in ugotovil mesto kapi. Brocovo področje je prikazano spodaj.



Slika 1: Brocovo področje v človeških možganih

Kaj pa razumevanje jezika? Kje bi bilo najbolj logično področje za ta predel? Najprej moramo razmisliti, če je jezik prvenstveno vizualen ali slušen. Ali najprej vidimo besede, ali jih slišimo? Kaj je bilo prej? Pisan ali govorjen jezik? Verjetno se strinjamo, da je jezik v prvi vrsti slušni fenomen. Po pričakovanjih je predel za razumevanje lociran takoj zraven slušnega podsistema. Imenuje se *Wernickovo področje* (Slika 2) po zdravniku, ki ga je odkril leta 1874 [14], tako da je preučeval paciente s težavami pri razumevanju govora. Pacienti niso razumeli jezika (pisanega ali govorjenega). Govor so sicer lahko producirali, vendar ni imel nobenega smisla. Samostalniki in glagoli so bili sicer povezani v nekem gramatičnem smislu, vendar stavki niti približno niso ustrezali temu, kar je pacient hotel izraziti.



Slika 2: Wernickovo področje v človeških možganih

Človeški jezik pa je le preveč kompleksen, da bi lahko bil razdeljen v samo dve diskretni področji. Očitno smo pozabili upoštevati še vizualne in manualne komponente za branje in pisanje jezika in še veliko drugih manj očitnih elementov. Velika uganka je tudi, kako lahko zbirka zlogov (ali bolje rečeno *vibracije molekul zraka*!) – na primer ime osebe – sproži predstavo obraza, osebnosti, datuma rojstva ali glasu te osebe v našem spominu. Jezik je verjetno lociran v predelih po celotnih možganih z obsežno komunikacijo med njimi. Diskretni področji – Brocovo in Wernickovo sta potrebni za jezik, nista pa zadostni.

2.3 Uvod v programske jezike in prevajalnike

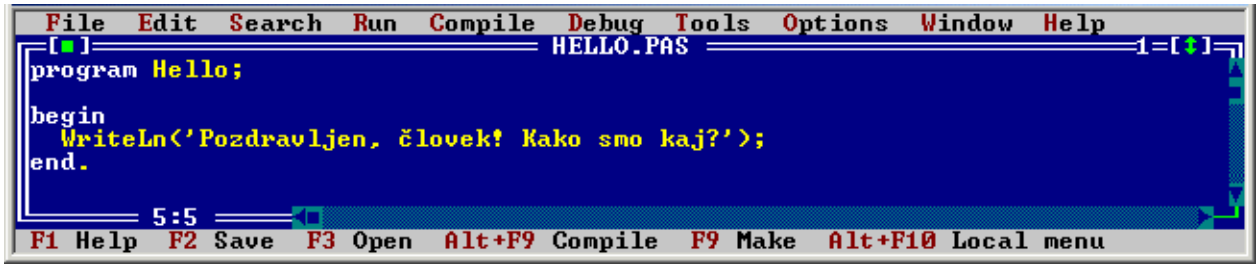
Da računalnik opravi delo namesto nas, mu moramo povedati *kako*. Računalniki so veliko enostavnejši od možganov in zato jim zaenkrat še ne moremo razložiti kaj želimo od njih z našim običajnim jezikom.

Možgani in računalnik se morata srečati na neki skupni točki, da se lahko razumeta. Razumeti se morata v dveh pogledih, in sicer:

- računalnik mora razumeti, kaj človek želi
- človek mora nato razumeti računalnikove rezultate

Nekdo bi lahko rekel, da je to stičišče grafični oziroma tekstovni vmesnik. To sicer drži, vendar imamo tukaj v mislih nižji nivo. Na nivoju vmesnika - oken in vnosnih polj - računalniku daje ukaze končni uporabnik. Na programskem nivoju pa *programerji*.

Najenostavnejši program, ki ga lahko zapišemo v programskem jeziku Pascal, je naslednji:



Slika 3: Enostaven program v Pascalu

Program izpiše na zaslonu stavek »Pozdravljen, človek...« (slika spodaj).

```
c:\>hello.exe
Pozdravljen, človek! Kako smo kaj?

c:\>_
```

Slika 4: Rezultat programa

Kratek komentar programa:

```
program Hello;                  ime programa - ne vpliva na izvajanje

begin                           začetek programa
  WriteLn('Pozdravljen...');    ukaz za izpis vrstice
end.                            konec programa
```

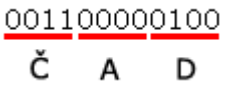
Tukaj se za nas srečata človeški in računalniški »razum«. Sicer se že prej – na nivoju elektronike, ampak o teh nivojih računalniške arhitekture programer zaradi abstrakcije ne rabi vedeti skoraj ničesar.

Na najnižjem nivoju je vse, kar računalnik lahko neposredno zazna, samo logična enica in logična ničla. Taka logična informacija je v splošnem lahko implementirana v *katerem koli* fizičnem mediju, pri računalniku pa je definirana s prisotnostjo električnega toka. Turingov stroj, ki je teoretični model, uporablja papirnat trak, na katerega zapisuje dva različna simbola. Turingov stroj je podlaga za teorijo izračunljivosti in v veliki meri za moderne računske stroje. Prvi mehanski računski stroj je delno razvil Charles Babbage v 19. stoletju. Celoten projekt se ni uspešno končal zaradi težav s financiranjem [30].

Če definiramo, kaj nam predstavljajo skupki štirih bitov, lahko predstavimo črke od A do O (16 črk):

0000 – A
0001 – B
0010 – C
0011 – Č
0100 – D
0101 – E
...
1111 - O

Besedica »ČAD« se v tem sistemu zapiše z naslednjim binarnim nizom:



Program je shranjen v spominu na enak način, vendar z ASCII kodo, ki je 8-bitna. Vendar pa to, da imamo v spominu shranjen kos besedila še ne pomeni ničesar. Za človeka je pomen sicer jasen, za računalnik pa ga moramo šele definirati.

Računalnik moramo naučiti, kako pretvoriti zgornji program v naslednje:

```
mov ax,cs
mov ds,ax
mov ah,9
mov dx, offset Hello
int 21h
xor ax,ax
int 21h

Hello:
  db "Pozdravljen...","$"
```

To je *zbirni jezik (assembly language*) in je končni jezik, ki ga procesor (čip) neposredno razume.

Programi, ki opravljajo nalogo pretvorbe izvorne kode v inštrukcije zbirnega jezika, se imenujejo prevajalniki in bodo podrobneje opisani v 3. poglavju.

Program na nivoju zbirnega jezika je sestavljen iz inštrukcij. Inštrukcije opravljajo razna elementarna opravila, kot so pisanje v spomin in branje iz njega, premikanje vrednosti med registri in podobno. Potrebno je razumeti, da so tudi inštrukcije shranjene v spominu v binarni obliki, vendar tako, da je vsaki prirejena točno določena vrednost (prikaz spodaj).

```
00000000 = mov
00000001 = int
00000010 = xor
00000011 = db
```

Med inštrukcijami in njihovimi binarnimi kodami torej obstaja linearna preslikava. To pomeni, da med tema dvema entitetama ni prav nobene razlike. Odvisno je samo, kako na njih

pogledamo. Človek vidi prejšnji program v zbirnen jeziku, tako kot je tukaj prikazan, računalnik pa ga »*vidi*« kot nekaj takega:

```
0000000000100100
1001100011110001
0010001000100011
1000011000110001
0001100001000000
0100010000111000
0010000010001000
```

Korake, ki so potrebni da pridemo iz izvorne kode progama do njegove binarne predstavitve v zbirnem jeziku, bomo pogledali v naslednjem poglavju.

3 Prevajalniki

// this is a comment, whole line will be ignored

3.1 Leksikalna analiza

Leksikalno analizo opravi leksikalni analizator, imenovan tudi pregledovalnik *(scanner)*, ki je vmesnik med izvornim programom in sintaktičnim analizatorjem. Glavna naloga leksikalnega analizatorja je posredovati sintaktičnemu analizatorju osnovne leksikalne simbole. Osnovni leksikalni simboli so: rezervirane besede (ključne besede jezika), ločila, identifikatori (npr. imena procedur), literali (besedilo med narekovaji) ipd. Osnovni leksikalni simboli prejšnjega programa so:

Leksikalni simbol	Tip
Hello	identifikator
;	ločilo
begin	rezervirana beseda
WriteLn	identifikator
(	ločilo
'Pozdravljen...'	literal
)	ločilo
;	ločilo
end	rezervirana beseda
.	ločilo

Tabela 1: Osnovni leksikalni simboli enostavnega programa

Pogosto uporabljan formalizem za opis osnovnih leksikalnih simbolov so deterministični končni avtomati. Deterministični končni avtomat je peterka $M =< Q, \Sigma, \delta, q_0, F >$, kjer je:

Q: končna množica stanj,

Σ: abeceda avtomata,

δ: parcialna funkcija, ki preslika stanje in vhodni simbol abecede Σ v novo stanje,

$(\delta : (Q \times \Sigma) \rightarrow Q)$,

q_0: začetno stanje avtomata $(q_0 \in Q)$

F: neprazna množica končnih stanj avtomata $(F \subset Q)$

Program, ki opravlja leksikalno analizo, imenujemo leksikalni analizator ali pregledovalnik. Najpreprosteje ga sestavimo tako, da implementiramo deterministični končni avtomat. Funkcijo prehodov δ lahko implementiramo s tabelo ali programsko kodo, kjer so stanja oznake, prehode pa implementiramo s skoki. Če leksikalni analizator implementiramo s tabelo, potrebujemo še algoritem, ki jo interpretira. Avtomat prične vedno z razpoznavanjem v začetnem stanju. Trenutno stanje in prebrani znak na vhodu določata naslednje stanje. Če ob trenutnem stanju in prebranem znaku na vhodu prehod ni določen, preverimo, ali je avtomat v končnem stanju. Če je v končnem stanju, leksikalni analizator vrne osnovni leksikalni simbol, ki je pridružen temu stanju, v nasprotnem primeru pride do leksikalne napake. Najpogostejši implementaciji determinističnega končnega avtomata sta s programsko kodo in s tabelo prehoda stanj. Prednosti implementacije funkcije prehodov s programsko kodo pred implementacijo s tabelo so predvsem manjša prostorska in deloma tudi časovna zahtevnost. Prednost implementacije s tabelo je preprostost, saj se leksikalni analizatorji med seboj razlikujejo le v tabeli. Algoritem, ki implementira tabelo pa ostaja nespremenjen.

3.2 Sintaktična analiza

V tej fazi prevajalnik preveri veljavnost programov. Program se mora skladati z *gramatiko* jezika (o tem, kako jo opišemo več v poglavju 6.2).

Primer neveljavnega programa bi bil:

```
begin begin
  WriteLn('Pozdravljen...;
end.
```

Ta program ni veljaven Pascalski program, ker je besedica »begin« podvojena in manjka zaklepaj.

Naloga sintaktične analize je ugotavljanje strukture stavkov jezika. Jezik L, generiran iz gramatike G, je definiran na naslednji način: $L(G) = \{x \mid Z \Rightarrow^* x \wedge x \in T^*\}$. Jezik je množica vseh tistih zaporedij terminalnih simbolov, ki jih lahko izpeljemo iz začetnega simbola gramatike. Takšna zaporedja simbolov imenujemo stavek jezika [5].

Uporaben jezik mora imeti določeno strukturo, zapisano s končnim številom pravil, ki povedo, kako naj povezujemo posamezne besede. Množico pravil, s katerimi opišemo strukturo stavkov jezika, imenujemo sintaksa ali gramatika jezika. Gramatika je četverka: G = <T, N, Z, P>, kjer so:

- T: končna množica terminalnih ali končnih simbolov (terminalni simbol je sinonim za osnovni leksikalni simbol iz leksikalne analize)
- N: končna množica neterminalnih ali nekončnih simbolov, $T \cap N = Ø$
- Z: začetni simbol, $Z \in N$
- P: množica produkcij oz. končna neprazna podmnožica relacije $(T \cup N)^* N (T \cup N)^* \rightarrow (T \cup N)^*$

Dogovorjeno je, da pišemo terminalne simbole (terminale) z malimi in neterminalne simbole (terminale) z velikimi črkami. Elemente $(T \cup N)^*$ označujemo z malimi grškimi črkami. Zaporedje terminalov zapišemo z T^*, zaporedje neterminalov pa z N^* [5]

Terminalni simboli so tisti, ki se neposredno pojavljajo v stavkih jezika (ključne besede, ločila, dobeseden tekst - literali), neterminali pa so skupine enega ali več terminalov zaradi lažjega zapisa pravil gramatike. Lahko bi definirali neterminal »LOČILA«, ki bi predstavljal piko, vejico ali podpičje. To bi zapisali takole:

LOČILA = . | , | ; *(znak »|« pomeni »ali«)*

Izpeljevanje pomeni, da zamenjamo neterminal z desno stranjo ene izmed produkcij, kjer nastopa ta neterminal na levi strani.

Za podajanje formalne sintakse, je v uporabi jezik BNF *(Backus-Naur Form)*. S pomočjo BNF definiramo produkcijska pravila, ki tvorijo gramatiko jezika.

Poglejmo najprej, kako deluje BNF na zelo preprostem jeziku aritmetičnih izrazov, katerega gramatika je definirana takole:

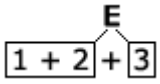
G = <{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {E}, E, P> z naslednjo množico produkcij:
E = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
E = E + E

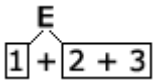
Izraz (E) je torej lahko

- ena izmed števk med 0 in 9,
- vsota dveh izrazov

Nekaj primerov izrazov v tako definiranem jeziku:
6
1 + 2
1 + 2 + 3

To so torej trije veljavni *stavki* jezika. Da so res veljavni, vidimo tako, da jih izpeljemo iz začetnega simbola gramatike (E).
Za število **6** takoj vidimo, da ustreza prvi produkciji
1 + 2 je enak E + E, kjer je prvi E enak številu 1, drugi pa številu 2.
1 + 2 + 3 si lahko razlagamo na dva načina, in sicer:





Take gramatike se imenujejo *dvoumne*. Gramatika je dvoumna, če za vsaj en stavek, ki ga generira ta gramatika, obstaja več kot eno drevo izpeljav. Treba je poudariti, da govorimo o dvoumni gramatiki in ne o dvoumnem jeziku. Za določen jezik obstaja več različnih gramatik, med katerimi so lahko določene dvoumne, druge pa ne. Obstajajo pa dvoumni jeziki, za katere ne najdemo nedvoumnih gramatik [5].

3.2.1 *Sintaktični analizator*

Sintaktični analizator ali razpoznavalnik *(parser)*, je program, ki opravlja sintaktično analizo. Na splošno obstajata dva pristopa. Pri prvem izhajamo iz začetnega simbola in poskušamo zgraditi drevo izpeljav, ki bi imelo za liste terminalne simbole. Ta pristop imenujemo razpoznavanje od zgoraj navzdol *(top-down parsing)*. V drugem pristopu izhajamo iz listov drevesa (terminalnih simbolov) in poskušamo pridi do korena drevesa (začetnega simbola). Ta pristop imenujemo razpoznavanje od spodaj navzgor *(bottom-up parsing)*. Pri razpoznavanju od zgoraj navzdol ločimo več pristopov in tehnik: tehniko vračanja *(backtracking, »brute-force«)* in razpoznavalnike LL(1), kot so: rekurzivni navzdolnji razpoznavalnik *(recursive-descent parser)*, skladovni razpoznavalnik in tabelarično vodeni razpoznavalnik *(table-driven parser)*. Pri razpoznavanju od spodaj navzgor prav tako poznamo več vrst razpoznavalnikov: razpoznavalnike na osnovi prioritete operatorjev *(operator precedence parser)*, razpoznavalnike na osnovi preproste prioritete *(simple precedence parser)*, razpoznavalnike na osnovi razširjene prioritete *(extended precedence parser)*, razpoznavalnike LR(k) *(LR(k) parser)*, razpoznavalnike SLR(1), razpoznavalnike LALR(1) itd. Razpoznavalniki LR so zelo učinkoviti in jih v praksi veliko uporabljamo, vendar pa je njihova zgradba zahtevnejša od razpoznavalnikov od zgoraj navzdol.

Če je gramatika LL(1) *(Left-to-right scanning, Left-most derivation)*, se lahko na osnovi terminalnega simbola na vhodu vedno odločimo za pravilno produkcijo in je vračanje nepotrebno. S tem smo izboljšali algoritem razpoznavanja, vendar smo hkrati omejili razred gramatik. Če je gramatika LL(1), lahko zapišemo zelo učinkovit sintaktični analizator ali razpoznavalnik. Razpoznavalnike LL(1) imenujejo nekateri avtorji tudi brezpovratna analiza z enosimbolnim oknom na vhodu *(one symbol look-ahead without backtracking)*.

3.3 Semantična analiza

Semantiko programskih jezikov opišemo običajno v naravnem jeziku. Takšen opis je razumljiv in dostopen širokemu krogu uporabnikov, vendar ima tudi veliko pomanjkljivosti, npr. nenatančnost in dvoumnost, ki dopuščata možnost različnih interpretacij. Zato potrebujemo formalno metodo, ki opiše pomen programskih stavkov na razumljiv in nedvoumen način. Danes še nimamo za semantiko podobnega standarda, kot je BNF za sintakso. Ena izmed formalnih

metod za opis semantike v uporabi je atributna gramatika *(atribute grammar)*, ki je posplošitev kontekstno prostih gramatik. Neterminalnim in terminalnim simbolom kontekstno proste gramatike pridružimo atribute, ki predstavljajo semantično informacijo. Vsaki produkciji dodamo množico semantičnih pravil za ovrednotenje atributov. Glede na zahtevnost ovrednotenja atributov delimo atributne gramatike na več razredov: atributne gramatike vrste S, atributne gramatike vrste L, neciklične in absolutno neciklične gramatike.
S pomočjo formalne metode lahko generiranje prevajalnikov in interpreterjev povsem avtomatiziramo.

3.3.1 *Atributna gramatika*

Naj bo dana kontekstno prosta gramatika G = <T, N, Z, P> z množico terminalov T, množico neterminalov N, začetnim neterminalom Z in produkcijami P, ki definirajo jezik L(G). Definirajmo množico $V = T \cup N$. Produkcijo $p \in P$ lahko pišemo kot $X_0 \to X_1, X_2 ... X_{n_p}$, kjer so $n_p \geq 0, X_0 \in N$ in $X_i \in V$ za $1 \leq i \leq n_p$. Privzemimo, da je kontekstno prosta gramatika G standardizirana *(standardized context free grammar)*, kar pomeni, da se začetni neterminal Z ne pojavlja na desni strani nobene produkcije. Če naša gramatika ni standardizirana, jo standardiziramo tako, da uvedemo novi začetni neterminal Z' in gramatiki dodamo produkcijo $Z' \to Z$.

Atributna gramatika je nadgradnja kontekstno proste gramatike G, tako da vsaki produkciji $p \in P$ priredimo množico semantičnih funkcij. Vsakemu neterminalu $X \in N$ priredimo dve ločeni končni množici atributov $S(X)$ in $I(X)$, kjer velja $S(X) \cap I(X) = \emptyset$. Elemente množice $S(X)$ imenujemo pridobljeni atributi *(synthesized attributes)* in elemente množice I(X) podedovani atributi *(inherited attributes)* neterminala X. Velja naj še $I(Z) = \emptyset$, kar pomeni, da začetni neterminal gramatike G nima podedovanih atributov, temveč samo pridobljene. Definirajmo še naslednje izraze:

- množico vseh atributov neterminala X: $A(X) = S(X) \cup I(X)$
- množico vseh pridobljenih atributov: $S = U_{x \in N} S(X)$,
- množico vseh podedovanih atributov: $I = U_{x \in N} S(X)$
- množico vseh atributov atributne gramatike: $A = I \cup S$.

Atribute si lahko predstavljamo kot spremenljivke nekega podatkovnega tipa, ki zavzamejo določene vrednosti. Vrednosti atributov so na začetku nedefinirane; določimo jih s semantičnimi funkcijami. Atributi predstavljajo poljubne objekte, ki jih želimo ovrednotiti po sintaktičnem razpoznavanju vhodnega stavka in vsebujejo informacije o neterminalih. Z njimi predstavimo poljubne vrednosti: niz, število, strukture, objekte itd.
Poleg neterminalov nastopajo v kontekstno prosti gramatiki tudi terminali. Tem simbolom ne prirejamo atributov, saj imajo že definiran pridobljen atribut, v katerem je shranjena njihova leksikalna vrednost *(lexem)*.

Postopek izračunavanja vrednosti atributov navedemo s semantičnimi funkcijami. Vsaki produkciji p: $X_0 \to X_1, X_2 ... X_{n_p}$ priredimo množico semantičnih funkcij, ki določajo način izražanja vrednosti atributov v množicah $S(X_0)$ in $I(X_i)$, $1 \le i \le n_p$. Za vsak pridobljeni atribut vsakega neterminala X_i, $1 \le i \le n_p$. Pri izvrševanju semantičnih funkcij velja omejitev, da se vsaka semantična funkcija izvrši samo enkrat, in tako se tudi atributom določi njihova vrednost samo enkrat.

3.4 Generatorji prevajalnikov

Generatorji prevajalnikov so orodja, ki popolnoma avtomatizirajo del ali celoten postopek prevajanja. Začetna komponenta je generator pregledovalnikov *(scanner generator),* ki dobi na vhodu formalni opis osnovnih leksikalnih simbolov in generira pregledovalnik, ki sprejema želene osnovne leksikalne simbole. Najbolj znan generator pregledovalnikov je Lex (1975), ki je standardno orodje okolja UNIX. Naslednja komponenta, generator razpoznavalnikov *(parser generator)* dobi na vhodu BNF in generira razpoznavalnik. Zgodnji generatorji razpoznavalnikov so generirali rekurzivne navzdolnje razpoznavalnike. Danes se pogosteje uporabljajo generatorji razpoznavalnikov, ki temeljijo na gramatikah LR, ker sprejemajo širši razred gramatik kot generatorji na podlagi gramatik LL. Najbolj znan generator razpoznavalnikov je YACC, ki je bil razvit leta 1975 na Berkeleyu. YACC uporablja PDA *(Push Down Automation)*. PDA razpoznavanje pomeni, da je naslednja akcija funkcija trenutnega stanja, terminalnega simbola na vhodu in stanja na skladu.

Struktura datoteke pri YACC-u je naslednja:

```
deklaracije
%%
Produkcijska pravila s semantičnimi akcijami
%%
Pomožna koda
```

V prvi sekciji (`deklaracije`) so deklarirani osnovni leksikalni simboli.
Primer:

```
%token FEATURE
%token ALL
%token ONE_OF
%token MORE_OF
%token OPT
...
%token NUMBER
```

Kaj natanko pomenijo ti simboli, moramo navesti še drugje v kodi (več v poglavju 7.2.1).

V drugi sekciji so podana produkcijska pravila, ki zelo močno spominjajo na notacijo v BNF, imajo pa dodane še semantične akcije. Semantične akcije so koda, ki se pokliče preden ali po tem, ko se razpozna katero od produkcijskih pravil. Za naš preprost jezik aritmetičnih izrazov sta definirani dve produkcijski pravili in dve semantični akciji, ki se izvršita *po* razpoznavanju ustreznih pravil:

```
E       : NUMBER
        {
                $$ = $1
        }

        | E '+' E
        {
                $$ = $1 + $3;
        }
```

Razlaga znakov $$ in $n, pri čemer je *n* naravno število:
»$$« - s tem simbolom vrnemo pomen neterminala, ki smo ga pravkar razpoznali,
»$n« - predstavlja pomen n-tega vhodnega parametra (terminala ali neterminala).

Pomen vsakega izraza je njegova numerična vrednost. V kolikor je izraz že število (prva produkcija), ga lahko enostavno vrnemo. »$1« predstavlja pomen terminala NUMBER, ki je že implicitno numerična vrednost. Pri drugi produkciji je izraz sestavljen iz prvega izraza, simbola '+' med njima in še drugega izraza. Pomen takega izraza je vsota števil za prvim in drugim izrazom. »$1« nam vrne pomen prvega izraza, »$3« pa pomen drugega. $2 bi nam vrnil pomen znaka '+' (torej sam znak '+' – njegov *lexem*), ki je drugi simbol produkcije.

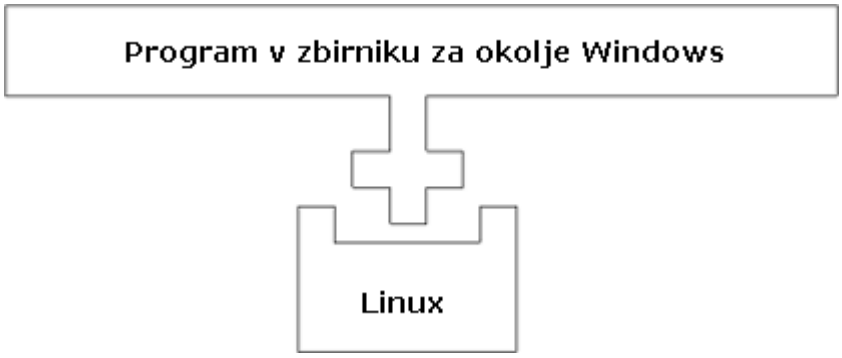
3.5 Prevajanje z vmesno kodo

Namen tega podpoglavja je, da se bralec okvirno seznani s pristopom vmesnega prevajanja - predvsem zato, ker Mono C# prevajalnik deluje znotraj ogrodja .NET, ki temelji na enakih principih kot java.

Tako kot obstaja veliko programskih jezikov, v katerih lahko pišemo programe, obstaja tudi veliko različnih operacijskih sistemov in procesorjev - *platform*, na katerih tečejo ti programi.

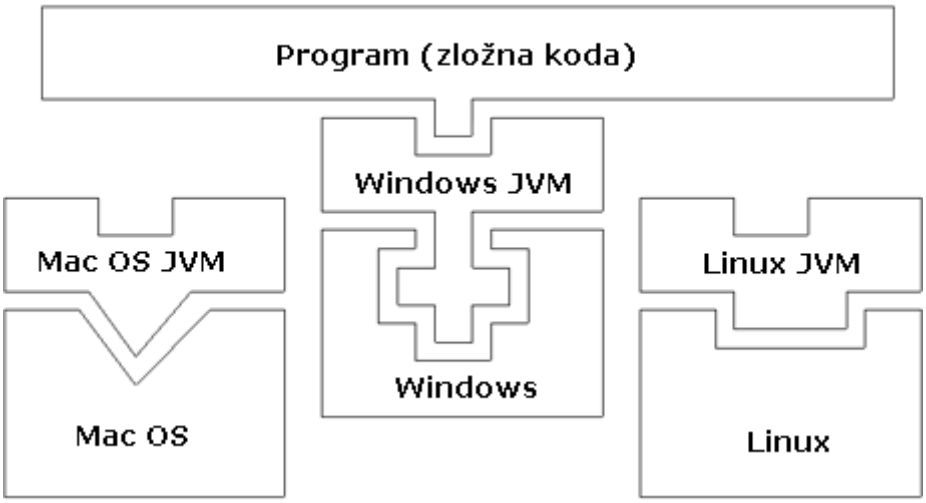
Različne platforme razumejo različne množice inštrukcij na najnižjem nivoju in to predstavlja problem, če imamo že preveden program. Običajno takega programa ne moremo neposredno zagnati na drugem sistemu. Tako programa, ki je bil preveden pod okoljem Windows (Slika 5) ne moremo zagnati pod Linuxom (Slika 6).

Slika 5: Program, preveden za okolje Windows



Slika 6: Isti program, ki pa ne more teči pod Linuxom

java ta problem elegantno reši tako, da prevede izvorno kodo v vmesni jezik, ki se imenuje zložna koda *(bytecode)*. Zložna koda lahko teče na javinem virtualnem stroju *(Java Virtual Machine – JVM)*. JVM je programsko implementiran procesor, ki mora biti naložen na sistem, na katerem želimo poganjati javine programe.



Slika 7: Funkcija javinega virtualnega stroja (JVM)

Isti preveden program lahko tako teče na vseh platformah, za katere je bil prirejen oz. narejen virtualni stroj (Slika 7). Aplikacij nam sedaj več ni potrebno prevajati v zbirni jezik za različne platforme, temveč le implementiramo virtualni stroj za vsako od njih. V moderni IT industriji so odličen kandidat za implementacijo navideznega stroja različne mobilne naprave, ki vsebujejo procesorje in operacijske sisteme veliko različnih proizvajalcev.

4 Vrste programskih jezikov

"Programs must be written for people to read, and only incidentally for machines to execute."

- *Abelson & Sussman, SICP, preface to the first edition*

Skozi kratko zgodovino računalništva se je razvilo več kot 3000 programskih jezikov. Razlike med nekaterimi so tako velike, da mora programer popolnoma spremeniti svoj način razmišljanja in pogledati problem iz nove perspektive.

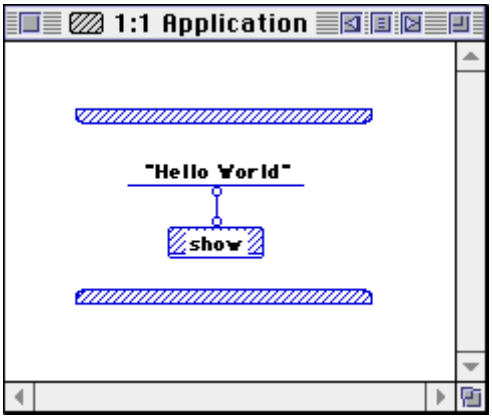
Poglejmo različne delitve jezikov [5]. Delitev po paradigmi (vzorcu) bomo obdelali v posebnem poglavju, ker je najbolj zanimiva in zahteva največ razlage.

Delitev programskih jezikov *glede na področje uporabe:*

- znanstvene aplikacije
- poslovne aplikacije
- umetno inteligenco
- sistemsko programiranje itd.

glede na zapis programov:

- linearne ali tekstovne (program, zapisan v obliki teksta)
- vizualne (program v obliki vizualne ponazoritve):



Slika 8: Primer v jeziku Prograph [17]

glede na nivo (število inštrukcij, ki so potrebne, da se izvede posamezen stavek jezika):

- nizke oz. zbirne,
- visoke,
- zelo visoke

glede na način implementacije:

- prevedene (strojna koda se ustvari iz višje nivojskega programa samo ob prevajanju)
- interpretirane (strojna koda se generira sproti ob vsakem zagonu)

glede na način opisa problema:

- imperativni (podamo podrobne korake rešitve)
- deklarativni (ni nam potrebno natančno opisati postopka rešitve)

glede na namen uporabe:

- splošno namenske (jeziki, ki se trudijo zadostiti širokemu spektru problemov)
- domensko specifične (manjši jeziki, ki rešujejo specifične probleme iz določenih področij)

Različni jeziki imajo po enega ali več atributov iz vsake skupine. Nek jezik je tako lahko imperativen, linearen, visokonivojski, interpretiran, objektno orientiran (delitev po paradigmi) in uporabljan pretežno v poslovnih aplikacijah.

4.1 Delitev po paradigmi

4.1.1 *Proceduralni jeziki*

Proceduralna abstrakcija *(procedure abstraction)* vključuje ukaze, ki se izvedejo pri njenem klicu. Kot rezultat dobimo ažurirane spremenljivke. Uporabnik proceduralne abstrakcije opazi le ta ažuriranja, ne pa tudi vmesnih korakov.
Procedure, ki vračajo neko vrednost se velikokrat imenujejo *funkcije*.

Primer preproste procedure v jeziku C:

```
int add(int a, int b)
{
        return a + b;
}
```

4.1.2 *Objektno orientirani jeziki*

Objektno orientirani jeziki so naravna evolucija proceduralnih jezikov. Omogočajo večjo produktivnost, ponovno uporabo kode in jasno izražanje logike. Programer mora pri prehodu na objekten način razmišljanja na novo prilagoditi svoj sistem dojemanja algoritmov. Pri objektnem pristopu smiselne logične enote združimo v objekte. Procedure, definirane nad objekti, se imenujejo metode. Koncept objektov in metod se nadalje razvije v kompleksnejše mehanizme objektnega pristopa:

- **kapsuliranje** – algoritmi so abstrahirani in skriti znotraj objektov. Koda, ki kliče metode nad objekti ne rabi (oz. skoraj *ne sme*) vedeti natančne implementacije teh metod,
- **dedovanje** – skupne lastnosti podobnih razredov se definirajo v enem nadrazredu, od katerega vsi ostali dedujejo,
- **polimorfizem** – klic metode nad vmesnikom nadrazreda povzroči klic ustrezne metode dejanskega razreda. Polimorfizem je mogoč zaradi dedovanja in je najmočnejše orodje objektne paradigme.

4.1.3 *Funkcijski jeziki*

Funkcijsko programiranje kot primarne programske konstrukte uporablja funkcijske klice. Osnove izhajajo iz teorije računalništva, zato funkcijski jeziki tvorijo povezavo med formalnimi metodami in njihovo praktično uporabnostjo. Teoretična podlaga za funkcijske jezike je *lambda račun*. Lambda račun je matematično orodje, s pomočjo katerega se da izračunati vse, kar je izračunljivo. Turingov stroj in lambda račun sta enako močna koncepta.

4.1.3.1 Imena spremenljivk in vrednosti

Tradicionalni programski jeziki so osnovani na konceptu *spremenljivke*. Spremenljivke so povezave med imenom *(identifikatorjem)* in vrednostjo. Takim jezikom pravimo imperativni, zato ker so sestavljeni iz niza ukazov:

```
<ukaz1>;
<ukaz2>;
<ukaz2>;
...
```

Ponavadi vsak ukaz zamenja vrednost neke spremenljivke, kar vključuje ovrednotenje izraza in prireditev rezultata identifikatorju:

```
<identifikator> := <izraz>
```

Izraz vsakega izmed ukazov se nanaša na spremenljivke, katerih vrednosti so lahko bile spremenjene v prejšnjih ukazih. Na ta način je omogočeno prenašanje vrednosti med ukazi.

Funkcijski jeziki so osnovani na strukturiranih funkcijskih klicih. Funkcijski program je izraz, sestavljen iz klica funkcije, ki kliče drugo funkcijo.

```
<funkcija1>(<funkcija2>(<funkcija3> ... ))
```

Vsaka izmed funkcij prejme vrednosti od klicoče funkcije in ji po končanem izvajanju pošlje novo vrednost - rezultat. To se imenuje *funkcijska kompozicija* ali *vgnezdenje*.

V imperativnih jezikih lahko ukazi spremenijo vrednosti spremenljivk, torej ima lahko *nek identifikator med izvajanjem programa prirejene različne vrednosti***.** V funkcijskih jezikih se identifikatorji navedejo le pri specifikaciji formalnih parametrov funkcij. Formalni parametri se povežejo z vrednostmi le ob klicu funkcij. Ko je formalni parameter povezan z dejanskim parametrom, ni več nobene možnosti, da bi se vrednost v njem spremenila. Koncept ukaza, kjer se spremeni vrednost, povezana z identifikatorjem, tukaj ne obstaja. To pomeni, da tudi koncepta zaporedja ali ponavljanja ukazov ni. *V funkcionalnih jezikih se identifikator poveže z vrednostjo le enkrat.*

4.1.3.2 Vrstni red izvajanja

V imperativnih jezikih je vrstni red ovrednotenja ukazov ponavadi bistven. Vrednosti se prenašajo med ukazi preko referenc do skupnih spremenljivk. Poljuben ukaz lahko spremeni vrednost neke spremenljivke preden je ta uporabljena v naslednjem ukazu. Če se vrstni red izvajanja ukazov spremeni, se lahko spremeni tudi obnašanje in pomen celotnega programa.

Imperativni jeziki imajo določen vrstni red izvajanja.

V funkcijskih jezikih funkcijski klici ne morejo spremeniti vrednosti, povezanih z identifikatorji. Vrstni red, po katerem se ovrednotijo vgnezdene funkcije ni pomemben, ker funkcijski klici ne morejo vplivati eden na drugega.

V nekem funkcijskem jeziku imamo lahko naslednji klic:

```
F(A(D), B(D), C(D))
```

vrstni red, po katerem se ovrednotijo A(D), B(D) in C(D) ni pomemben, ker funkcije A, B in C ne morejo spremeniti dejanskega argumenta D.

V funkcijskih jezikih ni pomemben vrstni red ovrednotenja. Ta neodvisnost je ena izmed prednosti funkcijskih jezikov, ki omogoča uporabo teh jezikov na različnih formalnih in praktičnih področjih.

4.1.3.3 Podatkovne strukture v funkcijskih jezikih

V imperativnih jezikih lahko spremenimo vrednosti elementov polj in sestavljenih tipov z zaporednimi prirejanji. V funkcijskih jezikih, ker ni prireditve, se podstrukture v podatkovnih strukturah ne morejo spremeniti, ampak je potrebno znova zapisati celotno strukturo z eksplicitnimi spremembami v podstrukturah.

Funkcijski jeziki tako ne podpirajo polj, ker bi morali zapisati celotno polje vsakič znova, če bi hoteli prirediti vrednost nekemu indeksu. Namesto tega vsebujejo funkcijski jeziki podatkovne strukture, ki so definirane kot seznami. Seznami so osnovani na rekurzivni notaciji, kjer so operacije na celoti opisane preko rekurzivnih operacij na podstrukturah. V programskem jeziku lisp je uporabljena enaka predstavitev za funkcije in podatkovne strukture.

4.1.3.4 Funkcije kot vrednosti

V veliko imperativnih jezikih lahko podamo podprograme kot dejanske parametre, le redki jeziki pa omogočijo vračanje podprogramov. V funkcijskih jezikih funkcije lahko ustvarijo nove funkcije in jih vrnejo kot rezultat. *Funkcijski jeziki omogočijo funkcijam, da jih obravnavamo kot vrednosti.* Koncept funkcij kot vrednosti izvira iz teoretičnih osnov funkcijskih jezikov – lambda računa.

4.1.4 *Logični jeziki*

Najbolj znan in uporabljan predstavnik te skupine je prolog *(PROgramming in LOGic)*. Osnovni entiteti programov v prologu so baza dejstev in logične relacije med njimi. Programer in morda končni uporabnik dobivata odgovore s pomočjo postavlja vprašanj v obliki predikatov. Sistem oceni predikat ali definira manjkajoče spremenljivke v njem, tako da preiskuje bazo in poskuša najti odgovor z logičnim sklepanjem oz. dedukcijo. Prolog je izrazito deklarativen jezik.

Primer baze dejstev in relacij:

```
% dejstva
male(albert).
male(peter).
male(james).
female(lisa).
female(sandra).
```

```
% relacije
parent(albert, lisa).
parent(albert, peter).
parent(peter, sandra).
parent(peter, james).

% siblings(A,B)
% pomeni, da sta A in B v relaciji brat-brat, brat-sestra ali sestra-sestra
siblings(A,B) :-
    parent(X,A),       % A in B sta potomca istega starša
    parent(X,B),       %
    A \== B.           % A ni ista oseba kot B
```

Predikati nad bazo:

```
?- male(sandra).

no


?- female(X).

X = sandra
X = lisa

yes


?- sibling(lisa, X).

X = peter

yes
```

Prolog pride do rešitev s pomočjo algoritma unifikacije. Unifikacija je implementirana z vračanjem *(backtracking)* nad drevesom izpeljav.

5 Domensko specifični jeziki

When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.

Alan Perlis

Domensko specifični jeziki so jeziki, prirejeni za določeno področje uporabe. V primerjavi s splošno namenskimi jeziki ponujajo znatne prednosti v izrazni moči in enostavnosti uporabe v svoji domeni. Razvoj DSL-jev je težek, ker zahteva poznavanje domene in izkušnje pri razvoju programskih jezikov. Ljudi, ki bi znali oboje, pa ni veliko. Posledično ne preseneča dejstvo, da se razvoj, če do razmišljanja o njem sploh pride, pogosto ustavi že pri ideji [4].

Veliko računalniških jezikov je domensko specifičnih. Domensko specifičnim jezikom pravimo tudi aplikacijsko-orientirani *(application-oriented)* [8], posebno-namenski *(special-purpose)* [9, p. xix], specifični po opravilu *(task-specific)* [10], specializirani *(specialized)* [11] in aplikacijski *(application)* [12].

DSL	Opis
BNF	Specifikacija sintakse
VBA	Makroji v Wordu in Excelu
HTML	Web strani
Latex	Urejanje besedila
Make	Avtomatizacija prevajanja programov
MATLAB	Inženirsko računalništvo
SQL	Poizvedovanje po podatkovnih bazah
VHDL	Načrtovanje logičnih vezij

Tabela 2: Primeri domensko specifičnih jezikov [4]

Velikokrat se splošno namenski jeziki pokažejo kot neustrezno orodje za reševanje problemov. Vzroki za to so:

- otežena jasnost razumevanja problema zaradi dodatnih konstruktov,
- večja možnost napak,
- strma krivulja učenja,
- velikokrat moramo računalniku natančno povedati, kako naj reši problem, s tem pa izgubimo prednost abstrakcije.

DSL-ji so ponavadi deklarativni, kar pomeni, da skrijejo veliko implementacijskih podrobnosti. Programerju je potrebno samo pravilno definirati problem, ni mu pa treba vedno znova podati korakov, ki pripeljejo do rešitve - to postane del abstrakcije.

5.1 Faze razvoja

5.1.1 *Odločitev*

Tukaj se ukvarjamo predvsem s vprašanjem »*kdaj*« razvijati DSL, medtem ko se v ostalih fazah sprašujemo »*kako*« jih razvijati [4]. V praksi pa razvoj DSL-jev ni linearen proces in zato lahko prehajamo med fazami. Na fazo odločitve lahko vpliva pred-analiza, ki sama po sebi mora včasih odgovoriti na vprašanja, ki se bodo šele *pojavila* v fazi načrtovanja. Na fazo načrtovanja pa pogosto vplivajo implementacijske omejitve.
Odločitev za razvoj novega DSL-ja ponavadi ni enostavna. Za pomoč pri odločanju o razvoju novega jezika so nam na voljo nekateri vzorci [4]:

Notacija *(notation)*: odločitveni faktor je razpoložljivost ustrezne (nove ali obstoječe) domensko-specifične notacije. Dva pomembna podvzorca sta:

- **transformacija vizualne predstavitve v besedilo**: pri tem se pojavljajo mnoge prednosti, kot so lažja kompozicija specifikacij večjih programov in omogočitev AVOPT načrtovalskega vzorca obravnavanega kasneje,
- **dodajanje uporabniško prijazne notacije v obstoječo knjižnico**: s tem načinom lahko »naredimo« DSL iz že narejene knjižnice.

AVOPT: domensko specifična analiza *(analysis)*, preverba *(verification)*, optimizacija *(optimization)*, paralelizacija *(parallelization)* in transformacija *(transformation)* ponavadi niso možne, ker so vzorci izvorne kode preveč kompleksni in preslabo definirani. Uporaba ustreznega domensko specifičnega jezika omogoči te operacije. Z nadaljnjim napredkom pri multiprocesiranju na nivoju čipa *(chip-level multiprocessing)*, bo ta vzorec postal vedno bolj pomemben [22]. Vzorec AVOPT se prekriva z večino ostalih.

Avtomatizacija opravil *(task automation):* programerji pogosto izgubijo veliko časa z dolgotrajnimi in ponavljajočimi se opravili v splošno namenskih jezikih. V takih primerih se lahko potrebna programska koda generira avtomatično z generatorjem aplikacij (prevajalnikov) za ustrezen DSL.

Produktna linija *(product line)**:*** člani iste skupine izdelkov [21] imajo enako arhitekturo in so razviti iz skupne množice osnovnih elementov. Uporaba DSL-jev lahko pogosto olajša njihovo specifikacijo. Ta vzorec ima veliko skupnega z vzorcem avtomatizacije opravil in sistemskim *front-end* vzorcem.

Predstavitev podatkovnih struktur *(data structure representation):* progamska koda temelji na inicializiranih podatkovnih strukturah, katerih kompleksnost lahko oteži pisanje in vzdrževanje kode. Takšne strukture so pogosto lažje izražene z uporabo domensko specifičnega jezika.

Prehod čez podatkovne strukture *(data structure traversal):* prehod čez kompleksne podatkovne strukture je lahko izražen bolje in bolj zanesljivo v ustreznem domensko specifičnem jeziku.

Sistemski »front-end« *(system front-end): front-end*, osnovan na DSL-ju je lahko pogosto uporabljen za obravnavanje sistemske konfiguracije in prilagoditve.

Vplivanje *(interaction):* vplivanje na aplikacijo z uporabo teksta in/ali ukazov iz uporabniškega vmesnika mora pogosto biti obogateno z ustreznim DSL-jem. V njem je naveden kompleksen ali ponavljajoč se vhodni tok podatkov. Za primer: Excelov interaktiven način je razširjen z makro jezikom, ki naredi aplikacijo »programabilno«.

Konstrukcija uporabniškega vmesnika *(GUI construction):* ta naloga se pogosto opravi s pomočjo opisa v domensko specifičnem jeziku.

5.1.2 *Analiza*

V fazi analize se identificira problemsko področje (domena) in pridobi znanje o njej. Znanje dobimo preko različnih tehničnih dokumentov, od strokovnjakov s področja, obstoječe izvorne kode in anket uporabnikov. Rezultati analize so zelo različni, ampak v osnovi sestavljeni iz domensko specifične terminologije in semantike.

Vzorci analize, definirani v [4] so:

- **neformalni**: domena je analizirana na neformalen način,
- **formalni**: uporabljena je metodologija domenske analize,
- **»extract from code«**: uporabljeno je »rudarjenje« (*»data mining«)*, ki izlušči domensko znanje iz zapuščinske *(legacy)* kode s pregledom, z uporabo orodij ali kombinacijo obojega.

Večinoma je domenska analiza narejena neformalno, včasih pa so uporabljene domensko analizne metodologije. Primeri takih metodologij so: DARE *(Domain Analysis and Reuse Environmen)* [23], DSSA *(Domain Specific Software Architectures)* [24], FAST *(Family-Oriented Abstractions, Specification and Translation)* [25], FODA *(Feature-Oriented Domain Analysis)* [26], ODE *(Ontology-based Domain Engineering)* [27], ODM (Organization Domain Modeling) [28].

5.1.3 *Načrtovanje*

Pri načrtovanju se moramo ukvarjati z dvema ločenima aspektoma:

- razmerje DSL-ja z obstoječim jezikom
- formalnim opisom načrtovanja

Najlažji način za razvoj domensko specifičnega jezika je, da ga baziramo na že obstoječem jeziku. Pri tem imamo vsaj dve možnosti; lahko uporabimo sintakso in konstrukte obstoječega jezika, lahko pa obstoječi jezik razširimo z novimi konstrukti. Različni načini so opisani v poglavju 5.1.4.

Ko je relacija z gostiteljskim jezikom definirana, se lahko lotimo načrtovanja domensko specifičnega jezika. Ločiti moramo med *neformalnim* in *formalnim* načrtovanjem. Prvo je velikokrat realizirano s pomočjo naravnega jezika in nekaj vzorčnimi DSL programi. Formalno načrtovanje pomeni uporabo specifikacij, napisanih s pomočjo katere od obstoječih metod za definicijo semantike (več o tem v poglavju 6.2).

5.1.4 *Implementacija*

5.1.4.1 Preprocesor

Konstrukti domensko specifičnega jezika se prevedejo v konstrukte gostiteljskega jezika s spreminjanjem izvorne kode. Statična analiza je omejena na tisto, ki jo naredi prevajalnik gostiteljskega jezika. Najpomembnejši pristopi so:

- **makro procesiranje:** pomen novih konstruktov se definira preko konstruktov gostiteljskega jezika preko makro definicij,
- ***source-to-source*** transformacija: DSL izvorna koda se transformira v izvorno kodo gostiteljskega jezika,
- **cevovod *(pipeline)*:** procesor zaporedno obdeluje podprograme DSL-ja in jih pošilja kot vhodne podatke v naslednji stadij,
- **leksikalno procesiranje:** tukaj je potrebna samo enostavna leksikalna analiza brez kompliciranе analize sintaktične drevesne strukture.

Prednost takega pristopa je zelo enostavna implementacija. Namreč skoraj vsa, če že ne vsa semantična analiza je opravljena preko procesorja gostiteljskega jezika. Vendar pa je ravno ta odsotnost semantične analize tudi glavna slabost. Javljanje napak je problematično, ker se sporočila o napakah nanašajo na koncepte gostiteljskega jezika.

5.1.4.2 Vgrajeni pristop

Pri vgrajenem pristopu se obstoječi mehanizmi gostiteljskega jezika uporabijo za gradnjo knjižnic, kjer se definirajo novi abstraktni tipi in operacije. Knjižnice - API so najosnovnejša oblika tega pristopa. Sintaktični mehanizmi gostiteljskega jezika se uporabijo za izražanje idioma

domene. Vgrajen DSL »podeduje« konstrukte splošno namenskega jezika in doda svoje domensko specifične konstrukte, ki so bližje domeni.

Prednost takega pristopa je uporaba že obstoječega prevajalnika oz. interpreterja za nek jezik. Glavna omejitev je manjša izraznost sintaktičnih mehanizmov v gostiteljskem jeziku. Velikokrat optimalna notacija ni dosežena zaradi omejitev gostiteljskega jezika. Javljanje napak ni tako učinkovito zaradi istih razlogov kot pri prejšnjem pristopu.

5.1.4.3 Prevajalnik / Interpreter

Pri tem pristopu so za implementacijo DSL-jev uporabljene standardne tehnike prevajalnikov / interpreterjev. Pri prevajalniku se DSL konstrukti prevedejo v konstrukte osnovnega jezika (jezika, v katerem je bil zapisan prevajalnik oz. interpreter) in klice knjižnic. Celotna statična analiza je narejena na specifikacijah DSL programa.

Pri interpreterju se DSL konstrukti razpoznajo in interpretirajo z uporabo standardnega cikla za dekodiranje inštrukcij *(fetch-decode-execute cycle)*. Ta način je primeren za jezike, ki so po naravi dinamični ali če hitrost izvajanja ni zelo pomembna. Prednosti interpretiranja pred prevajanjem so enostavnost, večja kontrola izvajalnega okolja in lažja razširitev.

Pomembna pomanjkljivost pristopa je, da je potrebno zgraditi prevajalnik oz. interpreter od začetka, kar pa je zelo zahtevno.

5.1.4.4 Generator prevajalnikov *(Compiler Generation)*

Ta pristop je zelo podoben prejšnjemu, s tem da se nekatere faze implementirajo s pomočjo sistemov za avtomatsko generiranje prevajalnikov. Ti sistemi se imenujejo prevajalniki prevajalnikov *(compiler compilers)*. Na ta način je trud, potreben za implementacijo, zelo zmanjšan in tako so slabosti prejšnjega pristopa minimizirane.

5.1.4.5 Razširjen prevajalnik / interpreter

To je pristop, ki je podrobneje opisan v tem diplomskem delu. V prevajalnik ali interpreter za določen splošno namenski jezik se vgradi sposobnost razumevanja novega domensko specifičnega jezika. Razširjanje interpreterjev je ponavadi razmeroma enostavno, medtem ko je

prevajalnike težko razširiti, razen če jasno podpirajo to možnost. Potrebna je velika mera previdnosti, da dodana domensko specifična notacija ne vpliva na obstoječo.

5.1.4.6 Commercial Off-The-Shelf (COTS)

Obstoječa orodja in/ali notacije so uporabljene na specifični domeni. Primer so domensko specifični jeziki, osnovani na standardu XML. Tak pristop predstavlja možno alternativo pri reševanju nekaterih domensko specifičnih problemov (npr. XML obeta veliko pri procesiranju in poizvedovanju dokumentov). V splošnem je XML neprijazen za človeka. Če je uporabljen skupaj z drugimi orodji *(*npr. *source-to-source* implementacijski pristop*)*, pa lahko generira zadovoljive rešitve.

6 Predstavitev izbranega domensko specifičnega jezika in definicija problema

You can never solve a problem on the level on which it was created.

Albert Einstein

Jezik, s katerim bomo razširili prevajalnik je **Feature Definition Language** (v nadaljevanju FDL). FDL je manjši domensko specifičen jezik, ki omogoča opis sistema s pomočjo definiranja relacij med *lastnostmi (features)*, ki ga sestavljajo. Vsak sistem je sestavljen iz ene ali več lastnosti, vsaka od njih pa iz drugih lastnosti - razen *atomičnih lastnosti*, ki so končni elementi.

Pomen podanega opisa so *vse možne konfiguracije sistema*.

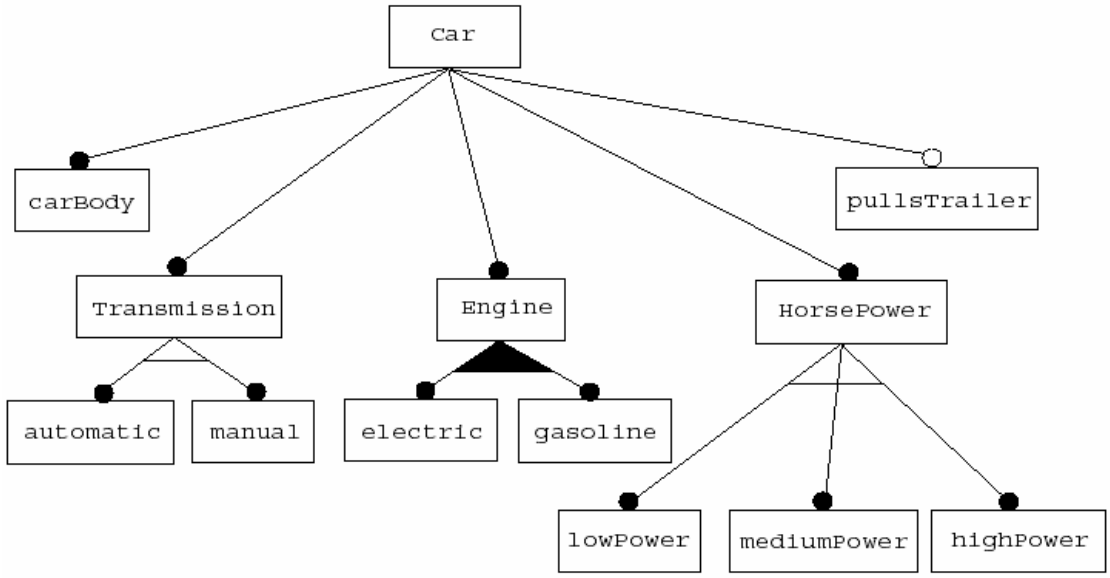
Veljavne lastnosti v jeziku FDL so naslednje:

- **AtomicFeature**: lastnost na najnižjem nivoju, ki ne vsebuje nobene druge,
- **OptFeature**: lastnost, ki izbirno vsebuje drugo lastnost,
- **OneOfFeature**: vsebuje več lastnosti, od katerih je lahko hkrati prisotna samo ena,
- **MoreOfFeature**: vsebuje več lastnosti, katerih poljubna kombinacija je lahko prisotna,
- **AllFeature:** vsebuje več lastnosti, ki se morajo pojaviti vse.

Opis sistema lahko podamo na dva načina, in sicer z *diagramom lastnosti* ali pa s *programom v jeziku FDL*.

6.1 Diagram lastnosti

Slika je za človeka ponavadi jasnejša od teksta oz. programa. Zato najprej poglejmo, kako bi podali opis nekega sistema z diagramom lastnosti - za primer vzemimo *avtomobil* (slika spodaj).



Slika 9:Diagram lastnosti

Avtomobil je sestavljen iz karoserije *(carBody)*, menjalnika (*Transmission),* motorja *(Engine)*, ima določeno konjsko moč *(HorsePower)* in izbirno še priključek za prikolico *(pullsTrailer)*. Izbirno prisotnost nakazuje bel krog. Pri lastnostih, ki morajo biti vedno prisotne je to prikazano s črnim krogom. Izmed pravkar naštetih lastnosti sta atomarni *(atomic)* samo *carBody* in *pullsTrailer*.

Če *Transmission* ni atomarna lastnost, mora biti sestavljena – torej *OneOf*, *MoreOf* ali *All*. *OneOf* je po dogovoru prikazana z ravno črto med vejami. Menjalnik je tako ali avtomatičen *(automatic)*, ali pa ročen *(manual)*, ne pa oboje hkrati.

Engine je označen s črnim trikotnikom med vejami, kar predstavlja lastnost tipa *MoreOf*. Motor je lahko električen *(electric)*, bencinski *(gasoline)* ali oboje hkrati. To so vse možne kombinacije.

Konjska moč avtomobila je lahko ali nizka *(lowPower)*, ali srednja *(mediumPower)*, ali visoka *(highPower)*, nikoli pa več hkrati.

Pomen diagrama lastnosti je definiran kot vse možne kombinacije, ki jih dobimo, če zadovoljimo vsem podanim kriterijem. Vse rešitve opisanega sistema so prikazane naslednje:

```
one-of(
all(carBody, automatic, electric, lowPower, pullsTrailer)
all(carBody, automatic, electric, lowPower)
all(carBody, automatic, electric, mediumPower, pullsTrailer)
all(carBody, automatic, electric, mediumPower)
all(carBody, automatic, electric, highPower, pullsTrailer)
all(carBody, automatic, electric, highPower)
all(carBody, automatic, electric, gasoline, lowPower, pullsTrailer)
all(carBody, automatic, electric, gasoline, lowPower)
all(carBody, automatic, electric, gasoline, mediumPower, pullsTrailer)
all(carBody, automatic, electric, gasoline, mediumPower)
all(carBody, automatic, electric, gasoline, highPower, pullsTrailer)
all(carBody, automatic, electric, gasoline, highPower)
all(carBody, automatic, gasoline, lowPower, pullsTrailer)
all(carBody, automatic, gasoline, lowPower)
all(carBody, automatic, gasoline, mediumPower, pullsTrailer)
all(carBody, automatic, gasoline, mediumPower)
all(carBody, automatic, gasoline, highPower, pullsTrailer)
all(carBody, automatic, gasoline, highPower)
all(carBody, manual, electric, lowPower, pullsTrailer)
all(carBody, manual, electric, lowPower)
all(carBody, manual, electric, mediumPower, pullsTrailer)
all(carBody, manual, electric, mediumPower)
all(carBody, manual, electric, highPower, pullsTrailer)
all(carBody, manual, electric, highPower)
all(carBody, manual, electric, gasoline, lowPower, pullsTrailer)
all(carBody, manual, electric, gasoline, lowPower)
all(carBody, manual, electric, gasoline, mediumPower, pullsTrailer)
all(carBody, manual, electric, gasoline, mediumPower)
all(carBody, manual, electric, gasoline, highPower, pullsTrailer)
all(carBody, manual, electric, gasoline, highPower)
all(carBody, manual, gasoline, lowPower, pullsTrailer)
all(carBody, manual, gasoline, lowPower)
all(carBody, manual, gasoline, mediumPower, pullsTrailer)
all(carBody, manual, gasoline, mediumPower)
all(carBody, manual, gasoline, highPower, pullsTrailer)
all(carBody, manual, gasoline, highPower)
)
```

Diagrami lastnosti so pomembno orodje pri modeliranju in domenski analizi. Prikažejo nam razne odvisnosti med entitetami domensko specifičnega jezika, ki ga želimo implementirati. Veljajo za pomembno tehniko pri *FODA (Feature-Oriented Domain Analysis)*.

Po fazi *analize*, ki smo jo pravkar zaključili, lahko začnemo z *načrtovanjem*. Pomen programa (*z normalizacijo in razširitvijo*) bomo iskali kasneje (podpoglavje 7.3).

FDL je bil uspešno uporabljen na kar nekaj praktičnih področjih. Dve izmed njih sta:

- Spletna trgovina, kjer se s pomočjo FDL-ja stranki najdejo vsi ustrezni tipi nekega izdelka glede na želene kriterije,
- pri *Feature Oriented Programming (FOP)* za kombiniranje programskih konstruktov. FOP temelji na podobnih principih kot *aspektno usmerjeno programiranje* [16].

6.2 Feature Definition Language (FDL)

Jezik FDL je pomensko ekvivalenten zapis diagrama lastnosti. Enak sistem, ki je bil predstavljen v diagramu lastnosti, opišemo z osnovnimi konstrukti jezika FDL tako:

```
feature transmission = one_of ("automatic", "manual")
feature engine = more_of ("electric", "gasoline")
feature horsePower = one_of ("lowPower", "mediumPower", "highPower")
feature car = all ("carBody", transmission, engine, horsePower,
                        opt("pullsTrailer"))
```

Komaj po formalni definiciji (podpoglavje 6.3) bomo lahko vgradili sposobnost razumevanja našega jezika v prevajalnik za jezik C#.

Pomen programa se poišče s pomočjo regularizacije *(regularization)*, normalizacije *(normalization)*, razširitve *(expansion)* in uporabe omejitev *(constraints)*.

6.2.1 *Regularizacija*

Z regularizacijo se reference na lastnosti zamenjajo z dejanskimi definicijami lastnosti.

Zgornji program postane:

```
feature car = all (
                        "carBody",
                        one_of ("automatic", "manual"),
                        more_of ("electric", "gasoline"),
                        one_of ("lowPower", "mediumPower", "highPower"),
                        opt ("pullsTrailer")
                    )
```

Tako pripravljena struktura je primerna za naslednji korak - *normalizacijo*.

6.2.2 *Normalizacija*

Diagram lastnosti se normalizira tako, da se uporabijo pravila, ki poenostavijo vse lastnosti, tako da se rešimo dvojnih pojavitev in »degeneriranih« primerov. Normaliziran diagram ima isti pomen kot original – mu je torej ekvivalenten. Normalizacija se torej reši redundance v zapisu. Podobno kot bi recimo v matematiki »normalizirali« enačbo *5 + 3 + 0 = 8* v *5 + 3 = 8*.

Normalizacija poteka po naslednjih pravilih:

```
[N1] Fs, F, Fs', F?, Fs'' = Fs, F, Fs', Fs''
[N2] Fs, F, Fs', F, Fs'' = Fs, F, Fs', Fs''
[N3] opt(opt(F)) = opt(F)
[N4] all(F) = F
[N5] all(Fs, all(Ft), Fs') = all(Fs, Ft, Fs')
[N6] one-of( F ) = F
[N7] one-of(Fs, one-of(Ft), Fs') = one-of(Fs, Ft, Fs')
[N8] one-of(Fs, F?, Fs') = one-of(Fs, F, Fs')?
[N9] more-of( F ) = F
[N10] more-of(Fs, more-of(Ft), Fs') = more-of(Fs, Ft, Fs')
[N11] more-of(Fs, F?, Fs') = more-of(Fs, F, Fs')?
```

Neformalna razlaga pravil za normalizacijo:

N1 združi obvezne in izbirne lastnosti seznama.

N2 izbriše podvojitve lastnosti v seznamu.

N3 združi vgnezdeno izbirno lastnost.

N4-N5 normalizira posebne primere *all*. Vgnezdene pojavitve se *sploščijo (flatten)*.

N6-N7 normalizira posebne primere *one-of*. Vgnezdene pojavitve se sploščijo.

N8 pretvori *one-of,* ki vsebuje eno izbirno lastnost v izbirni *one-of.*

N9-N10 normalizira posebne primere *more-of*. Vgnezdeni *more-of* se sploščijo.

N1 pretvori *more-of,* ki vsebuje eno izbirno lastnost v izbirni *more-of.*

Nekaj enostavnih primerov:

1. `all(all("automatic", "manual")) = all("automatic", "manual")`
   Pravilo [N5]

2. `all("automatic", "manual", "automatic") = all("automatic", "manual")`
   Pravilo [N5]

6.2.3 *Razširitev*

Pravila za razširitev so naslednja:

```
[E1] all(Fs, opt(F), Ft)
      = one-of(all(Fs, F, Ft), all(Fs, Ft))

[E2] all(Ft, opt(F), Fs)
      = one-of(all(Ft, F, F), all(Ft, Fs))

[E3] all(Fs, one-of(F, Ft), Fs')
      = one-of(all(Fs, F, Fs'), all(Fs, one-of(Ft), Fs'))

[E4] all(Fs, more-of(F, Ft), Fs')
      = one-of( all(Fs, F, Fs'),
         all(Fs, F, more-of(Ft),Fs'),
         all(Fs, more-of(Ft), Fs')
      )
```

Ft - seznam z eno ali več lastnostmi

Fs - seznam z nič ali več lastnostmi.

Neformalna razlaga pravil za razširitev:

E1, E2 pretvori *all,* ki vsebuje izbirno lastnost v dve lastnosti: eno z in drugo brez te lastnosti

E3 pretvori *all,* ki vsebuje *one-of* v dve lastnosti: eno s prvo alternativo in eno z *one-of* s prvo alternativo odstranjeno

E3 pretvori *all*, ki vsebuje *more-of*, v tri lastnosti: eno s prvo alternativo, eno s prvo alternativo in preostalim *more-of* in eno samo s preostalim *more-of*

6.2.4 *Omejitve*

Po dobljenem rezultatu moramo preveriti, če se vse lastnosti v rešitvi skladajo z omejitvami. Omejitve so lahko naslednje:

`A1 requires A2`: če je prisotna lastnost A1, potem mora biti prisotna tudi lastnost A2

`A1 excludes A2`: če je prisotna lastnost A1, potem lastnost A2 ne sme biti prisotna

`include A`: lastnost mora biti prisotna

`exclude A`: lastnost A ne sme biti prisotna

6.3 Definicija problema

V prevajalnik za jezik C# želimo vgraditi sposobnost razpoznavanja in iskanja pomena vzorčnega domensko specifičnega jezika – Feature Definition Language (FDL).

Razširjen program je prikazan tukaj:

```
class Car
{

  //začetek FDL specifikacij
  begin_spec(car, strResults)

    begin_features()
      feature transmission = one_of ("automatic", "manual")
      feature engine = more_of ("electric", "gasoline")
      feature horsePower = one_of ("lowPower", "mediumPower", "highPower")
      feature car = all ("carBody", transmission, engine, horsePower,
                         opt("pullsTrailer" ) )
    end_features()

    begin_constraints()
      constraint "pullsTrailer" requires "highPower"
      constraint include "pullsTrailer"
    end_constraints()

  end_spec()
  //konec FDL specifikacij

  static void Main()
  {
      Console.WriteLine(strResults);
  }
}
```

Rezultat, ki bi ga dobili v tem primeru je:

```
all(carBody, automatic, electric, highPower, pullsTrailer)
all(carBody, automatic, electric, gasoline, highPower, pullsTrailer)
all(carBody, automatic, gasoline, highPower, pullsTrailer)
all(carBody, manual, electric, highPower, pullsTrailer)
all(carBody, manual, electric, gasoline, highPower, pullsTrailer)
all(carBody, manual, gasoline, highPower, pullsTrailer)
```

V oklepajih v *begin_spec* sta podani ime glavne lastnosti *(car)* in ime spremenljivke tipa *string*, ki se naj ustvari implicitno in kamor se naj zapiše rezultat.

Dodatna možnost jezika FDL je, da lahko v njem opišemo tudi omejitve *(constraints)*. Omejitve so dodatni filtri nad končnim rezultatom, ki nam ohranijo samo rešitve, ki se skladajo z njimi.

Uporabili smo naslednji omejitvi:

```
constraint "pullsTrailer" requires "highPower"
constraint include "pullsTrailer"
```

Prva pomeni, da nujno potrebujemo visoko moč motorja, če želimo imeti možnost vlečenja prikolice. Druga omejitev pa določi, da morajo imeti vsi avtomobili nujno vgrajen priključek. Iz rešitev zgoraj vidimo, da sta bili podani omejitvi upoštevani. Skupno so v popolni definiciji jezika štiri omejitve (podpoglavje 6.2.4). Zraven omenjenih še *excludes* in *exclude*. *Excludes* prepove dano lastnost, če se pojavi določena druga lastnost. *Exclude* pa brezpogojno prepove določeno lastnost.

BNF notacija za FDL, ki ga bomo vgraditi v C# prevajalnik je naslednja:

```
FDL_SPECIFICATION ::= begin_spec(rootFeature, strResultsVariable)
                      FEATURE_SPEC
                      CONSTRAINT_SPEC?
                      end_spec

FEATURE_SPEC      ::= begin_features()
                      FEATURES_DECL*
                      end_features()

FEATURES_DECL     ::= feature FEATURE_NAME = FEATURE

FEATURE           ::= ALL_FEATURE | ONE_OF_FEATURE | MORE_OF_FEATURE |
                      OPT_FEATURE | ATOMIC_FEATURE

ALL_FEATURE       ::=  all(FEATURE_LIST)
ONE_OF_FEATURE    ::=  one-of(FEATURE_LIST)
MORE_OF_FEATURE   ::=  more-of(FEATURE_LIST)
OPT_FEATURE       ::=  opt(FEATURE_LIST_ELEMENT)
ATOMIC_FEATURE    ::=  LITERAL_STRING

FEATURE_LIST_ELEMENT ::= FEATURE_NAME | FEATURE
FEATURE_LIST         ::= FEATURE_LIST_ELEMENT*

FEATURE_NAME         ::= LITERAL_STRING

CONSTRAINT_SPEC      ::= begin_constraints()
                         CONSTRAINTS_DECL*
                         end_features()

CONSTRAINTS_DECL     ::= constraint CONSTRAINT

CONSTRAINT           ::= REQUIRES_CONSTRAINT | EXCLUDES_CONSTRAINT |
                         INCLUDE_CONSTRAINT | EXCLUDE_CONSTRAINT

REQUIRES_CONSTRAINT  ::= FEATURE_NAME requires FEATURE_NAME
EXCLUDES_CONSTRAINT  ::= FEATURE_NAME excludes FEATURE_NAME
INCLUDE_CONSTRAINT   ::= include FEATURE_NAME
EXCLUDE_CONSTRAINT   ::= exclude FEATURE_NAME
```

6.4 Mono C# prevajalnik

Mono je ambiciozen odprtokodni projekt, ki ga je začel Miguel Icaza, sedaj pa se naprej razvija pod sponzorstvom Novella. Ogrodje vsebuje vse potrebne komponente za razvoj in poganjanje .NET aplikacij na naslednjih operacijskih sistemih: Linux, Solaris, Mac OS, Windows in Unix. Projekt implementira različne tehnologije, ki so bile poslane na ECMA *(European Computer Manufacturers Association)* za standardizacijo s strani Microsofta.

.NET je Microsoftovo ogrodje, ki deluje na podobnih principih kot java (tukaj je mišljen celoten sistem z javinim virtualnim strojem in ne samo programski jezik java). .NET vsebuje prevajalnike za različne programske jezike, ki pretvorijo izvorno kodo v vmesni jezik IL *(Intermediate Language)*. Sistem, ki zna izvajati ta vmesni jezik in ga sproti prevajati v inštrukcije procesorja se imenuje CLR *(Common Language Runtime)*. Microsoft je implementiral CLR na svojem sistemu Windows, Mono projekt je tako zanimiv zato, ker je poskrbel za implementacijo še na ostalih razširjenih sistemih.

Mono vsebuje:

- prevajalnike za veliko različnih jezikov, izmed katerih je najbolj pomemben prevajalnik za C#
- virtualni stroj CLR
- zbirko knjižnic različnih tipov, ki poskrbijo za komunikacijo s sistemom *(core system routines)*

Prednosti:

- teče na več platformah
- zasnovan na ECMA/ISO standardih
- podpira C#, Javo, Visual Basic, Phyton, javaScript, Oberon, PHP, Object Pascal, DotLisp, #SmallTalk, ...
- odprtokoden, zastonj
- komercialno podprt

Mono se je že razvil do te mere, da razen .NET komponent implementira še naslednje:

- Remoting.CORBA: implementacija Corbe za Mono
- Ginzu: implementacija na najvišjem sloju Remotinga za ICE (Internet Communications Engine)
- Gtk#: povezava s popularnim Gtk+ uporabniškim vmesnikom za Linux/Windows
- #ZipLib: knjižnica za manipuliranje različnih stisnjenih datotek in arhivov (zip, tar...)
- GlGen: podpora za OpenGL
- Mono.LDAP: LDAP dostop za .NET aplikacije
- Mono.Data: podpora za Postgress, MySql, Sybase, DB2, SqlLite, Tds (SQL server protocol) in Oracle
- Mono.Cairo: povezava za Cairo prikazovalni modul (Mono podpora za standardne knjižnice za izris je implementirana s pomočjo tega modula)
- Mono.Posix: podpora za gradnjo POSIX aplikacij z uporabo C#
- Mono.Http: podpora za izdelavo lastnih, vgrajenih (embedded) HTTP serverov in HTTP upravljalcev

Nekatere od teh komponent so bile razvite v okviru Mono razvojne skupine, druge pa v *»open-source«* skupnosti.

Mono C# prevajalnik sem izbral zato, ker je njegova izvorna koda lahko dostopna [19] in ker je uspešen projekt, za katerega se bo podpora zanesljivo nadaljevala. Velika prednost je tudi, da je prevajalnik napisan kar v C# samem. Po prevajanju dobimo .NET program – prevajalnik, ki se zna izvajati na .NET ogrodju na vseh sistemih, kjer je implementiran. Posledica tega je, da lahko programer, ki želi razširiti prevajalnik z novim jezikom in ima že pripravljene knjižnice (API) za delo z njim, enostavno uporabi obstoječi API znotraj prevajalnika. Vse, kar mora storiti je, da vstavi produkcijska pravila v razpoznavalnik in od tam kliče funkcije API-ja.

7 Implementacija rešitve

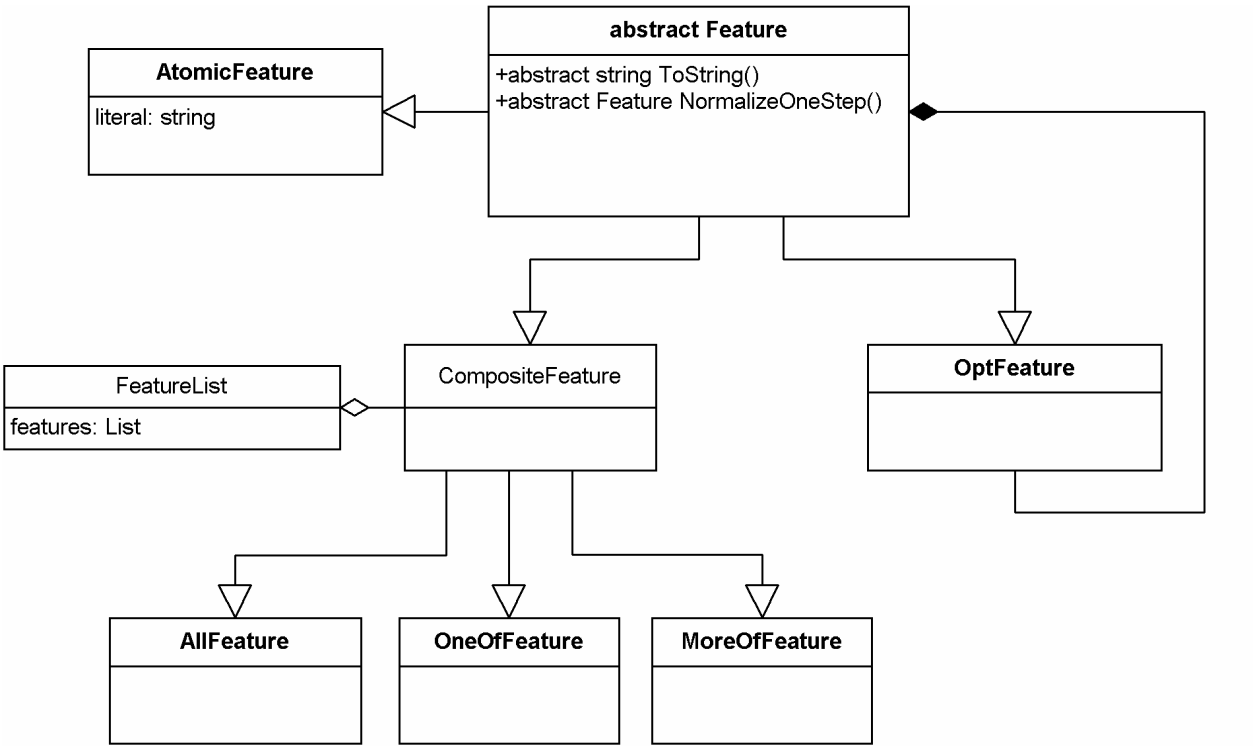
If you have a procedure with 10 parameters, you probably missed some.

Alan Perlis

7.1 Knjižnica (API)

Prvi korak pri implementaciji domensko specifičnih jezikov in včasih edini je, da razvijemo knjižnico za določen splošno namensko programski jezik. Uporabijo se konstrukti in sintaksa gostiteljskega jezika *(host language)*. Tak pristop se uporablja zato, ker je najenostavnejši in ker prevajalniki velikokrat nimajo lahko dostopne izvorne kode ali definiranih vstopnih točk za razširjanje. Še pomembnejši razlog pa je, da imajo redki programerji dovolj znanja iz analize, načrtovanja in implementacije domensko specifičnih jezikov.

Razredi v naši knjižnici, ki skrbijo za pripravo strukture lastnosti so naslednji:



Slika 10: UML diagram lastnosti

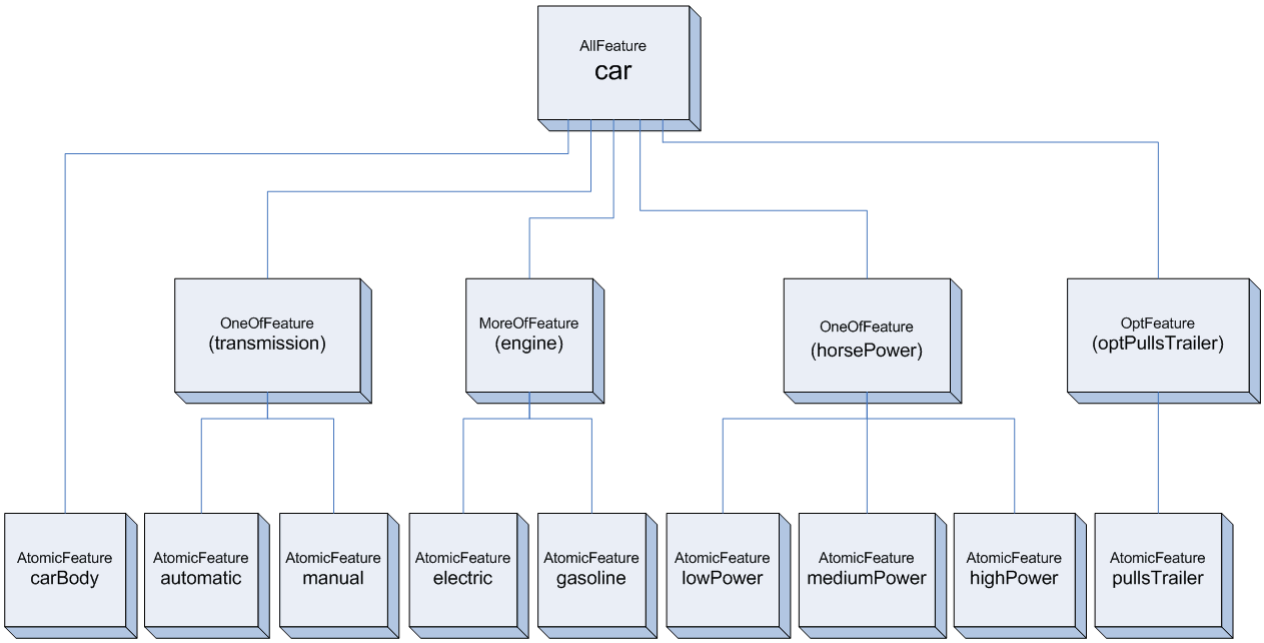
Tukaj so prikazane le osnovne relacije med razredi. Razvidno pa je tudi, da se znajo vse lastnosti:

- izpisati,
- normalizirati (o tem več v poglavju 6.2.2).

Z uporabo klicev iz pripravljene knjižnice naš program izgleda tako:

```
class Car
{

  static void Main()
  {
      AtomicFeature carBody = new AtomicFeature("carBody");

      OneOfFeature transmission = new OneOfFeature();
      transmission.AddChild(new AtomicFeature("automatic"));
      transmission.AddChild(new AtomicFeature("manual"));

      MoreOfFeature engine = new MoreOfFeature();
      engine.AddChild(new AtomicFeature("electric"));
      engine.AddChild(new AtomicFeature("gasoline"));

      OneOfFeature horsePower = new OneOfFeature();
      horsePower.AddChild(new AtomicFeature("lowPower"));
      horsePower.AddChild(new AtomicFeature("mediumPower"));
      horsePower.AddChild(new AtomicFeature("highPower"));

      OptFeature optPullsTrailer = new OptFeature(
            new AtomicFeature("pullsTrailer")
      );

      AllFeature car = new AllFeature(
            carBody, transmission, engine, horsePower, optPullsTrailer
      );

      ConstraintManager constraints = new ConstraintManager();

      constraints.AddConstraint(
            new RequiresConstraint("pullsTrailer", "highPower")
      );

      constraints.AddConstraint(
            new IncludeConstraint("pullsTrailer")
      );


      string strResults = ResultsManager.GetResults(car, constraints);
      Console.WriteLine(strResults);
  }
}
```

Struktura objektov, ki se vzpostavi v spominu je naslednja:



Slika 11: Diagram lastnosti v spominu

Kasneje bomo drevo zgradili z uporabo istih razredov ob samem razpoznavanju programa. Koda za to bo podana v semantičnih akcijah pred in po specifikaciji ustreznih pravil za razpoznavanje. Razredi za iskanje pomena programa so opisani v poglavju 7.3. Vstopna točka je metoda, ki prejme zgoraj prikazano drevo in nad njim opravi normalizacijo, razširitev, pregleda omejitve ter najde rešitev.

7.2 Mono razpoznavalnik

Mono razpoznavalnik se generira iz specifikacij, ki so zapisane v datoteki s končnico *.jay*. Jay je poseben format zapisa leksikalnih simbolov, produkcijskih pravil in semantičnih akcij in je verzija YACC-a *(yet another compiler compiler),* prirejena za C#.

7.2.1 *Implementacija produkcijskih pravil (sintakse)*

Pravila za razpoznavanje razredov so zapisana v datoteki *cs-parser.jay*. Iz te datoteke se avtomatično generira C# koda razpoznavalnika za Mono C# prevajalnik. Pogledali bomo samo nekaj osnovnih konstruktov.

Definicija razreda je:

```
class_declaration
  : opt_attributes
    opt_modifiers
    CLASS IDENTIFIER
    opt_class_base
    class_body
    opt_semicolon
;
```

Ta definicija je širše umeščena v definicijo imenskega prostora *(namespace)*, ki tukaj ni prikazana. Semantične akcije so izpuščene zaradi enostavnosti.

Vsak razred ima pred ključno besedico *class* izbirno lahko *atribute* in *določila dostopa* (*modifiers*). Atributi v C# so podatki, ki jih lahko priredimo različnim tipom (razredom, strukturam itd.).

Kode v *jay* datoteki za razpoznavanje atributov tukaj ne bo navedena, kdor je zainteresiran, pa si lahko pogleda izvorno kodo na domači strani projekta Mono [19].

Polna definicija določil dostopa s podanimi akcijami je naslednja:

```
opt_modifiers
      : /* empty */           { $$ = (int) 0; }
      | modifiers
      ;

modifiers
      : modifier
      | modifiers modifier
        {
            int m1 = (int) $1;
            int m2 = (int) $2;

            if ((m1 & m2) != 0) {
                  Location l = lexer.Location;
                  Report.Error (1004, l, "Duplicate modifier: `" +
                                          Modifiers.Name (m2) + "'");
            }
            $$ = (int) (m1 | m2);
        }
        ;

modifier
      : NEW             { $$ = Modifiers.NEW; }
      | PUBLIC          { $$ = Modifiers.PUBLIC; }
```

```
      | PROTECTED       { $$ = Modifiers.PROTECTED; }
      | INTERNAL        { $$ = Modifiers.INTERNAL; }
      | PRIVATE         { $$ = Modifiers.PRIVATE; }
      | ABSTRACT        { $$ = Modifiers.ABSTRACT; }
      | SEALED          { $$ = Modifiers.SEALED; }
      | STATIC          { $$ = Modifiers.STATIC; }
      | READONLY        { $$ = Modifiers.READONLY; }
      | VIRTUAL         { $$ = Modifiers.VIRTUAL; }
      | OVERRIDE        { $$ = Modifiers.OVERRIDE; }
      | EXTERN          { $$ = Modifiers.EXTERN; }
      | VOLATILE        { $$ = Modifiers.VOLATILE; }
      | UNSAFE          { $$ = Modifiers.UNSAFE; }
      ;
```

V edinem semantičnem pravilu se preverja samo, če je bilo trenutno določilo dostopa že uporabljeno. V primeru, da je res prišlo do dvojne navedbe nekega določila (recimo *public public*), se vrne napaka pri prevajanju s pomočjo funkcije *Report.Error*.

Glavni del, *class_body*, je sestavljen iz deklaracij članov *(class members)*:

```
class_body
      :  OPEN_BRACE opt_class_member_declarations CLOSE_BRACE
      ;

opt_class_member_declarations
      : /* empty */
      | class_member_declarations
      ;

class_member_declarations
      : class_member_declaration
      | class_member_declarations
        class_member_declaration
      ;

class_member_declaration
      : constant_declaration
      | field_declaration
      | method_declaration
      | property_declaration
      | event_declaration
      | indexer_declaration
      | operator_declaration
      | constructor_declaration
      | destructor_declaration
      | type_declaration
      ;
```

Našo definicijo želimo umestiti sem, kot še enega možnega člana – enako kot je na primer definicija metod. To bo izgledalo takole:

```
class_member_declaration
      : constant_declaration
      | field_declaration
      | method_declaration
      | property_declaration
      | event_declaration
      | indexer_declaration
      | operator_declaration
      | constructor_declaration
      | destructor_declaration
      | type_declaration
      | FDL_specification
      ;
```

Sedaj lahko zapišemo pravila, ki bodo prepoznavala FDL specifikacije na podlagi navedene BNF notacije. Kodo za semantično razpoznavanje bomo dodali naknadno. Zaenkrat bi radi dosegli samo, da se prevedejo le pravilni FDL podprogrami, trenutno pa se pomen programa še ne razpozna.

Na začetku moramo definirati rezervirane besede. Prvi korak je, da jih navedemo v *.jay* datoteki, tako da jih dodamo k že obstoječim leksikalnim simbolom.

```
// nekaj že obstoječih definicij
%token ABSTRACT
%token AS
%token ADD
%token ASSEMBLY
%token BASE
%token BOOL
...

// naše definicije
%token BEGIN_SPEC
%token END_SPEC
%token FEATURE
%token ALL
%token ONE_OF
%token MORE_OF
%token OPT
%token BEGIN_FEATURES
%token END_FEATURES
%token BEGIN_CONSTRAINTS
%token END_CONSTRAINTS
%token CONSTRAINT
%token REQUIRES
%token EXCLUDES
%token INCLUDE
%token EXCLUDE
```

Taka navedba osnovnih leksikalnih simbolov povzroči samo, da se v *cs-parser.cs* (ki nastane iz *cs-parser.jay* po prevodu z Jay prevajalnikom) ustvari razred *Token*:

```
// %token constants
 class Token {

  public const int ABSTRACT = 261;
  public const int AS = 262;
  public const int ADD = 263;
  public const int ASSEMBLY = 264;
  public const int BASE = 265;
  public const int BOOL = 266;
  public const int BREAK = 267;

  ...

  public const int BEGIN_SPEC = 342;
  public const int END_SPEC = 343;
  public const int FEATURE = 344;
  public const int ALL = 345;
  public const int ONE_OF = 346;
  public const int MORE_OF = 347;
  public const int OPT = 348;
  public const int BEGIN_FEATURES = 349;
  public const int END_FEATURES = 350;
  public const int BEGIN_CONSTRAINTS = 351;
  public const int END_CONSTRAINTS = 352;
  public const int CONSTRAINT = 353;
  public const int REQUIRES = 354;
  public const int EXCLUDES = 355;
  public const int INCLUDE = 356;
  public const int EXCLUDE = 357;
}
```

Prevajalnik iz tega ne more dobiti literalov, ki jih potrebuje za prepoznavanje. To so samo konstante, ki jih uporabljamo v izvorni kodi. Znanje o dejanskih rezerviranih besedah, ki ustrezajo posamezni konstanti, podamo v metodi *InitTokens()* v *cs-tokenizer.cs* takole:

```
static void InitTokens ()
{
      keywords = new CharArrayHashtable [64];

      AddKeyword ("abstract", Token.ABSTRACT);
      AddKeyword ("as", Token.AS);
      AddKeyword ("add", Token.ADD);
      AddKeyword ("assembly", Token.ASSEMBLY);
      AddKeyword ("base", Token.BASE);
      AddKeyword ("bool", Token.BOOL);

      ...

      AddKeyword ("begin_spec", Token.BEGIN_SPEC);
      AddKeyword ("end_spec", Token.END_SPEC);
      AddKeyword ("feature", Token.FEATURE);
```

```
      AddKeyword ("all", Token.ALL);
      AddKeyword ("one_of", Token.ONE_OF);
      AddKeyword ("more_of", Token.MORE_OF);
      AddKeyword ("opt", Token.OPT);
      AddKeyword ("begin_features", Token.BEGIN_FEATURES);
      AddKeyword ("end_features", Token.END_FEATURES);
      AddKeyword ("begin_constraints", Token.BEGIN_CONSTRAINTS);
      AddKeyword ("end_constraints", Token.END_CONSTRAINTS);
      AddKeyword ("constraint", Token.CONSTRAINT);
      AddKeyword ("requires", Token.REQUIRES);
      AddKeyword ("excludes", Token.EXCLUDES);
      AddKeyword ("include", Token.INCLUDE);
      AddKeyword ("exclude", Token.EXCLUDE);
}
```

Specifikacija »okostja« sledi direktno iz prikazane BNF notacije za FDL in je:

```
FDL_specification
      : BEGIN_SPEC OPEN_PARENS IDENTIFIER opt_spec_variable CLOSE_PARENS
        opt_features_decl
        opt_constraints_decl
        END_SPEC OPEN_PARENS CLOSE_PARENS opt_semicolon
      ;

//spremenljivka, v katero se bo zapisal pomen programa po prevajanju
opt_spec_variable
      : /* empty */    { $$ = null; }
      | COMMA IDENTIFIER
    ;

opt_features_decl
      : /* empty */
      | BEGIN_FEATURES OPEN_PARENS CLOSE_PARENS features_decl END_FEATURES
OPEN_PARENS CLOSE_PARENS opt_semicolon
      ;

features_decl
      : feature_decl
      | features_decl feature_decl
    ;

feature_decl:
  FEATURE IDENTIFIER ASSIGN feature
;

feature : all_feature
        | one_of_feature
        | more_of_feature
        | opt_feature
        | atomic_feature
;

all_feature:
  ALL OPEN_PARENS feature_list_elements CLOSE_PARENS
;
```

```
one_of_feature:
  ONE_OF OPEN_PARENS feature_list_elements CLOSE_PARENS
;

more_of_feature:
  MORE_OF OPEN_PARENS feature_list_elements CLOSE_PARENS
;

opt_feature:
  OPT OPEN_PARENS feature_list_element CLOSE_PARENS
;

atomic_feature:
  LITERAL_STRING
;

feature_list_elements:
  feature_list_element
  | feature_list_elements COMMA feature_list_element
;

feature_list_element:
  IDENTIFIER
  | feature
;

opt_constraints_decl
      : /* empty */
      | BEGIN_CONSTRAINTS OPEN_PARENS CLOSE_PARENS
         opt_constraints_decl
         END_CONSTRAINTS OPEN_PARENS CLOSE_PARENS opt_semicolon
      ;

opt_constraints_decl
   : /*empty*/
   | constraints_decl
   ;

constraints_decl
      : constraint_decl
      | constraints_decl constraint_decl
    ;

constraint_decl:
  CONSTRAINT LITERAL_STRING REQUIRES LITERAL_STRING
  | CONSTRAINT LITERAL_STRING EXCLUDES LITERAL_STRING
  | CONSTRAINT INCLUDE LITERAL_STRING
  | CONSTRAINT EXCLUDE LITERAL_STRING
;
```

Naslednji korak, ki bo potreben je, da vgradimo semantične akcije, ki se bodo izvedle pred oz. po razpoznavanju posameznih pravil. Te semantične akcije podamo v obliki C# kode. Osnovni cilj bo zgraditi strukturo lastnosti s pomočjo C# konstruktov, ki smo jih že definirali v API (podpoglavje 7.1). Dobiti moramo isto drevo v spominu kot na Slika 11.

7.2.2 *Implementacija semantičnih akcij z uporabo sklada*

Pred implementacijo semantičnih akcij moramo poskrbeti še za regularizacijo (podpoglavje 6.2.1) – torej na problem sklicevanja na lastnosti. Sklicevanje pomeni, da je neka lastnost definirana samostojno v kodi in se na njo sklicujemo od drugod. Primer, ko je lastnost *engine* definirana posebej, lastnost *car* pa se nanjo sklicuje:

```
feature car = all( "carBody", transmission, engine, horsePower,
                   opt("pullsTrailer" ) )
```

Lastnosti *Transmission*, *engine* in *horsePower* se bodo v prvi fazi predstavili kot objekti tipa *FeatureReference*. FeatureReference vsebuje samo ime lastnosti, na katero se sklicujemo ("transmission", "engine", "horsePower"), medtem ko so te lastnosti resnično definirane drugje v kodi. Na primer:

```
feature transmission = one_of ("automatic", "manual")
```

Na tem mestu se iz definicij na desni strani enačaja ustvari objekt tipa *OneOfFeature*, za katerega pa ni takoj jasno, kateri lastnosti pripada. Par *literal "transmission" – objekt OneOfFeature* zato shranimo s pomočjo objekta tipa *ReferenceResolver*. Ta razred vsebuje samo omenjeno preslikavo, ki nam pretvori ime lastnosti v ustrezen objekt, zgrajen med razpoznavanjem izvorne kode.

Vsak razred, ki deduje od »Feature«, vsebuje posebno metodo *ResolveReferences*, ki sprejme ReferenceResolver objekt in vrne nov objekt tipa Feature, ki je enak staremu, s tem da več ne vsebuje nobenih sklicev na lastnosti, temveč lastnosti same.

Po klicu ResolveReferences nad lastnostjo *car*, dobimo torej naslednje:

```
feature car = all( "carBody",
                   one_of ("automatic", "manual"),
                   more_of ("electric", "gasoline"),
                   one_of ("lowPower", "mediumPower", "highPower"),
                   opt    ("pullsTrailer") )
```

Sedaj lahko začnemo z implementacijo semantičnih akcij. Osredotočimo se na posamezne deklaracije lastnosti, kot je naslednja:

```
feature transmission = one_of ("automatic", "manual")
```

Pravilo za njihovo razpoznavanje je:

```
feature_decl:
  FEATURE IDENTIFIER ASSIGN feature
;
```

Pri tem nastopata dva neterminala, in sicer ime lastnosti in njena deklaracija. Pri razpoznavanju pa se bo iz nje ustvaril objekt v spominu, ki bo predstavljal ustrezno lastnost (Slika 11).

Torej je »IDENTIFIER« v bistvu referenca na lastnost in zato bomo ta par *ime-lastnost* dodali v naš referenceResolver tako:

```
feature_decl:
  FEATURE IDENTIFIER ASSIGN feature
  {
    referenceResolver.AddPair((string)$2, (Feature)$4);
  }
;
```

»Feature« je, kot že vemo, lahko enega izmed naslednjih tipov:

```
feature : all_feature
        | one_of_feature
        | more_of_feature
        | opt_feature
        | atomic_feature
;
```

Če hočemo, da to pravilo vrne zgrajen objekt, ga bomo morali zgraditi v posameznih pravilih, ki razpoznajo vsakega od tipov lastnosti. Začnimo z »atomic feature«:

```
atomic_feature:
  LITERAL_STRING
  {
    $$ = new AtomicFeature((string)$1);
  }
;
```

Neterminal *AtomicFeature* je v definiciji FDL samo literal, zato ga lahko zgradimo zelo enostavno in kar takoj vrnemo. V pravilu, ki je klicalo to pravilu pa moramo rezultat podati naprej, in sicer tako:

```
feature : all_feature
        | one_of_feature
        | more_of_feature
        | opt_feature
        | atomic_feature
          {
            $$ = $1
          }
;
```

Sedaj se lahko lotimo pravila za sestavljene lastnosti. Poglejmo za primer sestavljeno lastnost *AllFeature*:

```
all_feature:
  ALL OPEN_PARENS feature_list_elements CLOSE_PARENS
;
```

Lastnost *AllFeature* je sestavljena lastnost, zato moramo ustrezno ovrednotiti seznam.
Pravila za seznam so naslednja:

```
feature_list_elements:
  feature_list_element
  | feature_list_elements COMMA feature_list_element
;

feature_list_element:
  IDENTIFIER
  {
    //tu bomo morali ustvariti FeatureReference
  }
  | feature
  {
    // ***
  }
;
```

Na mestu, označenem s tremi zvezdicami, se nam pojavi problem. Tukaj bomo imeli ustrezen objekt (dedovan od razreda Feature) za trenutno lastnost v seznamu, ki smo jo razpoznali. Očitno je, da jo moramo dodati kot otroka lastnosti, kateri dejansko pripada, nimamo pa dostopa do reference na ta objekt, ker Jay ne pozna podedovanih atributov. Zato uporabimo sklad.

V pseudokodu bi to naredili tako:

```
feature :
{
  //potisni (Push) na sklad nov AllFeature
  featureStack.PushNewAllFeature(); //enkapsuliramo dejansko ustvarjanje
}
all_feature // ko se tukaj razpozna posamezen element seznama, ga dodaj kot
otroka lastnosti, ki je na vrhu sklada
{
  //vzami lastnost s sklada (Pop) in ga vrni klicočemu pravilu razpoznavanja
  $$ = featureStack.PopFeature();
}

        | one_of_feature
        | more_of_feature
        | opt_feature
        | atomic_feature
;
```

Pri razpoznavanju elementa seznama, kot že rečeno, postopamo takole:

```
feature_list_element:
  IDENTIFIER
  {
    //ustvarimo poseben, začasen tip Feature (FeatureReference), ki bo
    //izginil iz drevesne strukture, ko se pokliče metoda Resolve (glej
    //opis ReferenceResolver)

    FeatureReference fe = new FeatureReference((string)$1);
    featureStack.AddChildToTopmostFeature(fe);
  }
  | feature
  {
    featureStack.AddChildToTopmostFeature((Feature)$1);
  }
;
```

Razpoznavanje omejitev je zelo enostavno, saj je njihova definicija v BNF sestavljena iz samih terminalov:

```
constraint_decl:
  CONSTRAINT LITERAL_STRING REQUIRES LITERAL_STRING
  {
     constraintManager.AddConstraint(new RequiresConstraint((string)$2, (string)$4));
  }
  | CONSTRAINT LITERAL_STRING EXCLUDES LITERAL_STRING
  {
     constraintManager.AddConstraint(new ExcludesConstraint((string)$2, (string)$4));
  }
  | CONSTRAINT INCLUDE LITERAL_STRING
  {
    constraintManager.AddConstraint(new IncludeConstraint((string)$3));
  }
  | CONSTRAINT EXCLUDE LITERAL_STRING
  {
    constraintManager.AddConstraint(new ExcludeConstraint((string)$3));
  }
;
```

7.3 Iskanje pomena programa

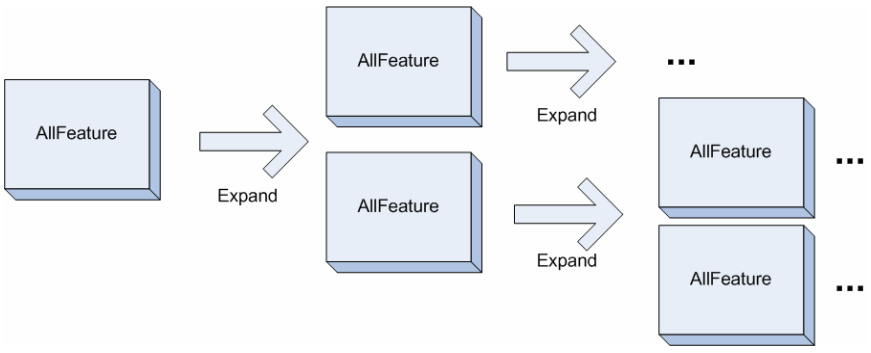
Prvi korak pri iskanju pomena programa je normalizacija (formalna definicija je bila opisana v poglavju 6.2.2). Metoda *NormalizeOneStep*, ki jo morajo implementirati vse lastnosti, naredi največ eno normalizacijo. Klicati jo moramo tako dolgo, dokler lastnost ni normalizirana. Za ta namen imamo naslednjo metodo v razredu *ResultsManager* (Slika 15):

```
public static Feature Normalize(Feature feature)
{
	while(!feature.IsNormalized)
	{
		feature = feature.NormalizeOneStep();
	}

	return feature;
}
```

Glavno orodje za iskanje pomena programa je razširitev (podpoglavje 6.2.3). Razširitev vedno deluje nad lastnostmi tipa *AllFeature*, in sicer tako, da jih razcepi v eno ali več lastnosti istega tipa.

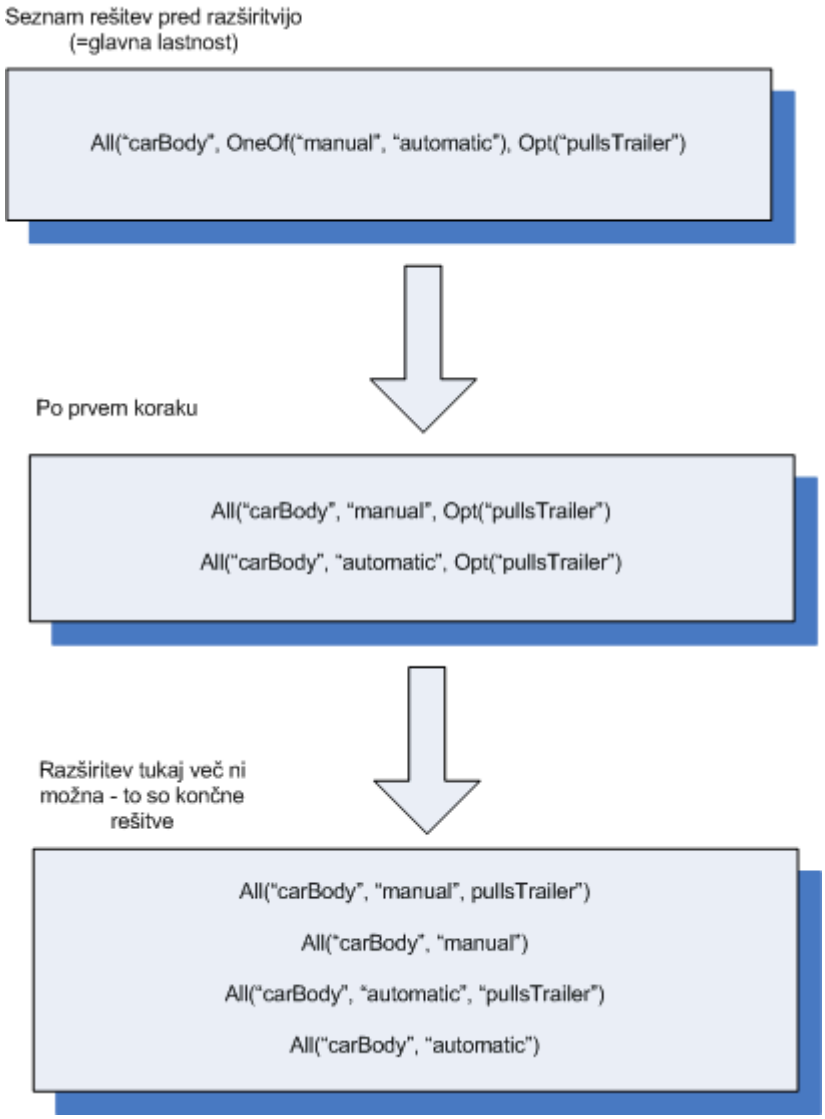
V začetku imamo samo en rezultat - osnovno lastnost. V kolikor ta sama ni končna (ne vsebuje samih atomarnih lastnosti), se razširi v dve ali več lastnosti. Vsaka od teh pa še nadalje, dokler niso vsi rezultati lastnosti *AllFeature*, ki vsebujejo samo atomarnih lastnosti.

Diagram, ki lepše prikaže pravkar povedano:



Slika 12: Shematski prikaz razširitve

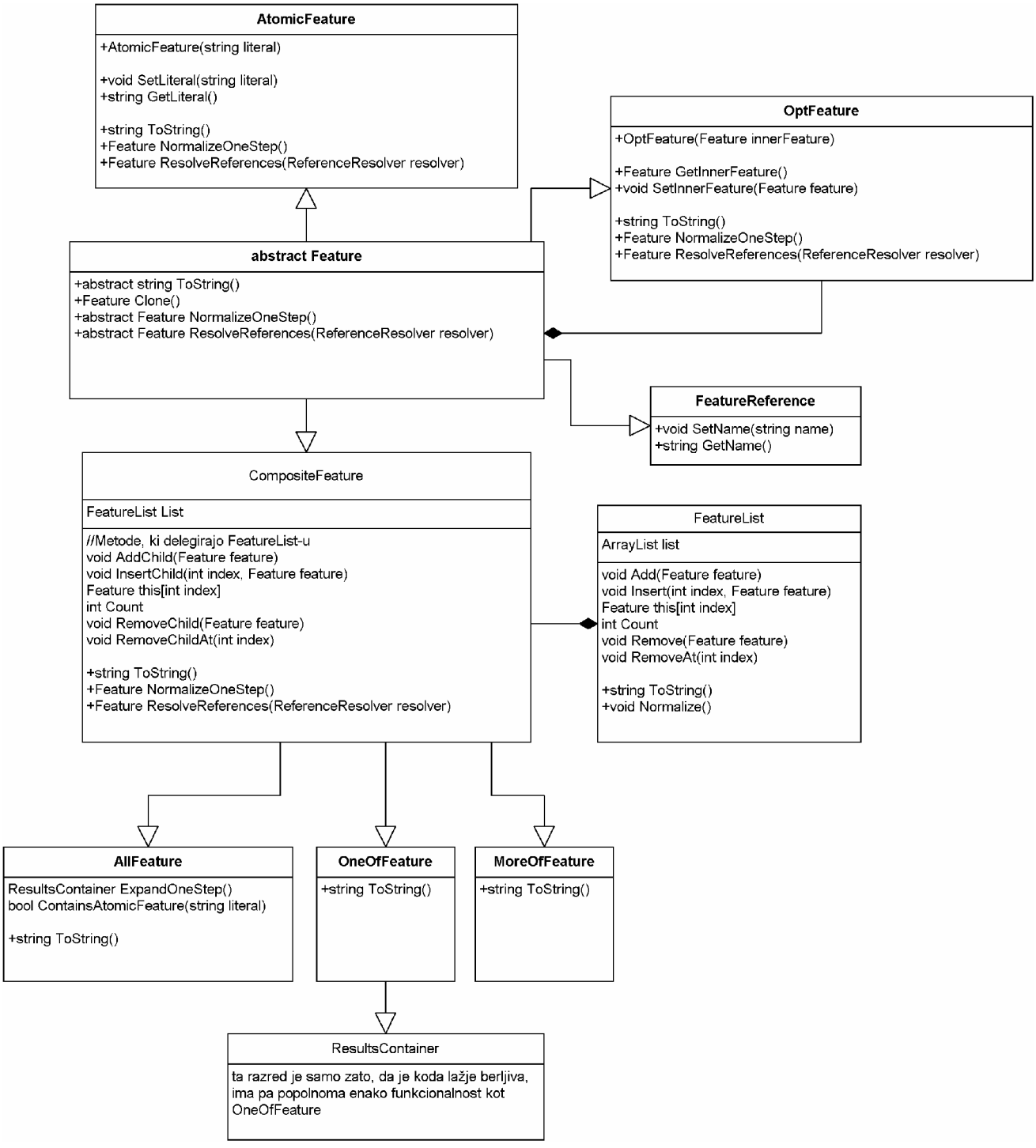
Zaradi jasnosti poglejmo sedaj še enostaven praktičen primer. Poenostavimo sistem *(car)* tako, da vsebuje samo karoserijo, menjalnik in priključek za prikolico.



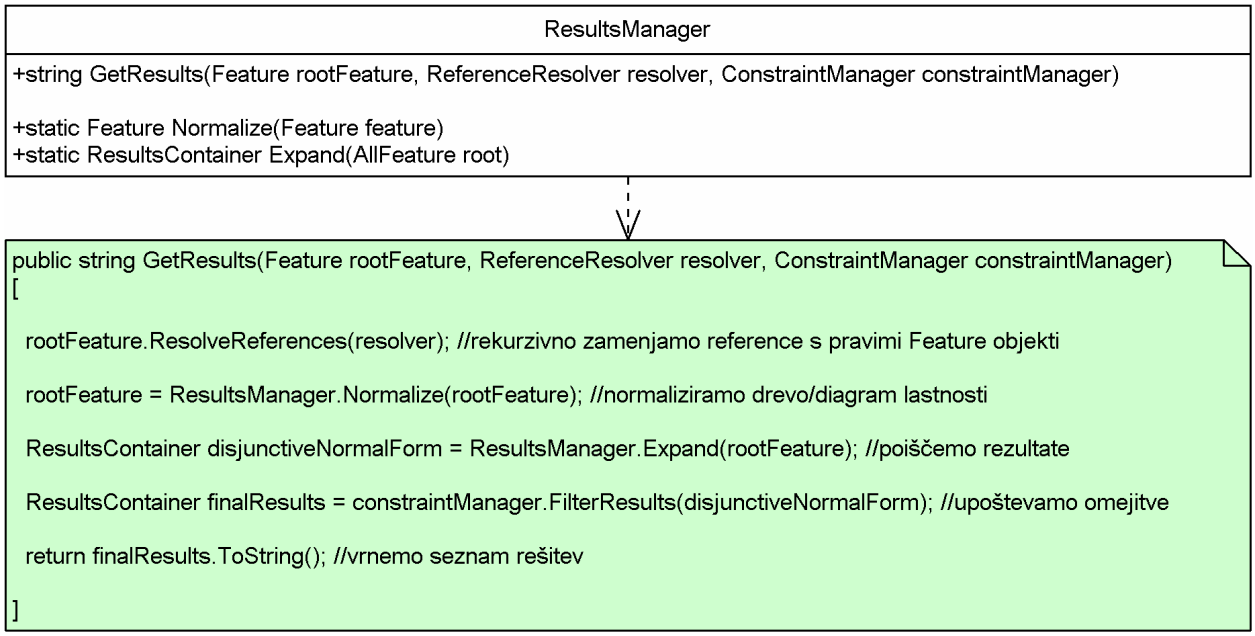
Slika 13: Razširitev po korakih na preprostem primeru

Potek je poenostavljen in v nekaterih pogledih ne sledi očitno pravilom (podpoglavje 6.2.3). Pravila so podana rekurzivno. Lastnost *AllFeature*, ki vsebuje lastnost *OneOf,* se razcepi v dve. V eno, ki vsebuje glavo lastnosti *OneOf* in drugo, ki vsebuje lastnost *OneOf* z ostalimi elementi seznama. Tako dobljena lastnost *AllFeature* se bo dalje cepila preko tega člana *OneOf*, dokler ne bo sama vsebovala samo enega elementa. Lastnost *OneOf*, ki vsebuje samo eno lastnost pa je ekvivalentna tej notranji lastnosti (po pravilu normalizacije [N5] – podpoglavje 6.2.2). Primer: OneOf(»manual«) = »manual«.

7.3.1 *UML diagrami glavnih razredov knjižnice (API)*



Slika 14: Podroben UML diagram strukture osnovnih razredov



Slika 15: Glavna metoda za iskanje pomena programa

8 Rezultati

The roots of education are bitter, but the fruit is sweet.
- Aristotle

Implementirali smo domensko specifičen jezik Feature Definition Language v odprtokodni prevajalnik Mono C# Compiler. Po tem, ko smo imeli pripravljeno knjižnico (API) za delo s FDL, s samo razširitvijo prevajalnika nismo imeli pretirano veliko dela. Semantična pravila smo lahko razmeroma enostavno vgradili v razpoznavalnik, tako da smo preslikali BNF domensko specifičnega jezika v JAY razpoznavalnik. Pozorni smo morali biti le na to, da ne ogrozimo obstoječe strukture prevajalnika.

Nekatere prednosti pristopa z razširitvijo so:

- možno je definirati nove konstrukte, ki imajo za posledico jasnejšo kodo,
- sporočanje napak in njihove natančne lokacije med prevajanjem je učinkovito, ker se doseže na nivoju domensko specifičnega jezika,
- pri nekaterih urejevalnikih lahko dodamo nove konstrukte (ključne besede, operatorje itd.) v seznam, tako da se med urejanjem ustrezno obarvajo.

Slabosti pristopa so naslednje:

- najti je potrebno prevajalnik, katerega izvorna koda je lahko dostopna, oziroma so pripravljene vstopne točke,
- uporabnik se mora naučiti novih konstruktov.

Formalna primerjava z ostalimi implementacijskimi pristopi je bila narejena znotraj raziskave *Experiencing diverse implementation approaches for Domain Specific Languages* [6] v okviru sodelovanja med Fakulteto za elektrotehniko, računalništvo in informatiko v Mariboru ter argentinsko Fakulteto za informatiko LIFIA. Članek obravnava najbolj znane pristope implementacije domensko specifičnih jezikov. Eden izmed opisanih pristopov je tudi razširjanje prevajalnika, implementirano v tem diplomskem delu. Primerjava med pristopi je povzeta v nadaljevanju.

Če želimo primerjati uporabljen pristop z ostalimi, moramo primerjavo razdeliti na:

- trud, potreben za implementacijo
- trud končnega uporabnika pri uporabi implementiranega jezika

8.1 Primerjava truda pri implementaciji z različnimi pristopi

Ocenitev vloženega truda pri implementaciji v primerjavi z ostalimi pristopi je težavna, saj so bili uporabljeni zelo različni programski jeziki (objektno orientirani, funkcijski, imperativni in hibridni). Različni pristopi vsebujejo različne sekcije kode za sintaktično analizo (npr. vgrajeni pristop sploh ne implementira te kode). Zaradi tega ne moremo primerjati samo števila porabljenih vrstic kode, temveč moramo uporabiti različne metrike.

8.1.1 *Effective Lines Of Code*

Osnovno primerjavo izvedemo tako, da pogledamo, koliko vrstic kode *(Effective Lines Of Code - eLOC)* je bilo potrebnih za implementacijo (tabela spodaj). Vrednost *eLOC* je porazdeljena med sintakso in semantiko. V sintaksi je zajeto število vrstic kode za implementacijo sintaktičnega analizatorja, v semantiki pa število vrstic za implementacijo semantičnih pravil v razpoznavalnik. V primerih, ko koda za sintakso ni bila potrebna, je polje v tabeli prazno.

Pristop	Jezik	Sintaksa	Semantika	eLOC	Rang
source-to-source	java	512	1283	1795	10
	Lisa	44	1272	1316	6
	haskell		578	578	2
makro procesiranje	C++		1482	1482	8
vgrajeni pristop	haskell		294	294	1
interpreter/prevajalnik	java	512	1003	1515	9
generator prevajalnikov	Lisa	48	911	950	5
	SmaCC	29	794	829	4
razširjen prevajalnik	**C#**	**103**	**515**	**618**	**3**
COTS	XML		1383	1383	7

Tabela 3: Primerjava efektivnega števila vrstic (eLOC), potrebnih za implementacijo pri različnih pristopih

Če primerjamo samo po metriki *eLOC*, je pristop z razširitvijo prevajalnika na 3. mestu (nižje število pomeni manj truda po tej metriki).

8.1.2 *Analiza funkcijskih točk*

Metrika *Function Point Analysis* (FPA) je močno orodje, ki omogoča ocenitev stroškov in truda pri projektih, zajemanje zahtev in merjenje kvalitete. FPA je neodvisna od uporabljene tehnologije in jo zato lahko lepo uporabimo pri ocenitvi eLOC, potrebnih za različne vrste implementacij. S pomočjo te metrike normaliziramo različne pristope, da jih lahko pravično primerjamo. Opis in primeri izračunov te metrike so navedeni v [6].

8.2 Primerjava truda za končnega uporabnika

8.2.1 *Primerjava sintakse DSL programov*

Vsi pristopi niso sposobni izraziti čiste DSL notacije, kar je posledica narave uporabljenega jezika ali uporabe določene obstoječe infrastrukture. Vseeno pa so bili vsi, razen vgrajenega pristopa in COTS, dovolj blizu originalni notaciji. Za grobo oceno razlik med različnimi notacijami je bila uporabljena LOC metrika (Tabela 4).

Pristop	Jezik	IOdevice	Car	Menu	BigSample	Deviacija	Rang
čista FDL notacija		6	6	13	20		
API	java	58	53	99	163	8.56	10
source-to-source	java	6	6	13	20	1	1
	Lisa	6	6	13	20	1	1
	haskell	6	6	19	20	1	1
makro procesiranje	C++	13	13	45	25	1.76	7
vgrajeni pristop	haskell	39	37	13	49	4.64	9
interpreter/prevajalnik	java	6	6	13	20	1	1
generator prevajalnikov	Lisa	6	6	13	20	1	1
	SmaCC	6	6	13	20	1	1
razširjen prevajalnik	**C#**	**19**	**16**	**23**	**30**	**2.27**	**8**
COTS	XML	103	95	196	332	16.17	11

Tabela 4: Primerjava velikosti DSL programov (eLOC)

Ta metrika ne more povedati dovolj o razliki. Pri makro procesiranju je za ločitev elementov seznama uporabljen znak '|' (namesto ',' pri originalni notaciji). Torej se lahko programi razlikujejo dosti bolj, kot lahko to izrazi metrika LOC.

Bolj realistična primerjava DSL programov se doseže s sistemi za računanje faktorja podobnosti med kodo DSL programov in originalno notacijo. Uporabljen je sistem odkrivanja plagiatorstva, razvit in uporabljan na FERI [33]. Ta sistem uporablja mehko logiko *(fuzzy logic)*. Rezultati testa podobnosti so prikazani v Tabela 5

Pristop	Jezik	Podobnost	Rang
API	java	23.7%	11
source-to-source	java	100%	1
	Lisa	100%	1
	haskell	100%	1
makro procesiranje	C++	53.28%	7
vgrajeni pristop	haskell	38.35%	9
interpreter/prevajalnik	java	100%	1
generator prevajalnikov	Lisa	100%	1
	SmaCC	100%	1
razširjen prevajalnik	**C#**	**52.69%**	**8**
COTS	XML	25.69%	10

Tabela 5: Testiranje podobnosti med programom in originalno FDL notacijo

Pri pristopih *source-to-source,* interpreter / prevajalnik in generator prevajalnikov je bila dosežena 100% podobnost. Pri vseh ostalih pristopih je podobnost manjša od pričakovane – približno 50% ali manj.

8.2.2 *Učinkovitost (performance)*

Učinkovitost prevajanja je faktor, ki ga ne smemo zanemariti pri nekaterih problemskih področjih. Za ta namen je bil ustvarjen obsežen FDL program - *BigSample*. Program ima 20 eLOC in generira rešitev z 11880 možnostmi. Po merjenju časa prevajanja vseh implementacij na istem istemu se je za najhitrejšega izkazal pristop z razširjanjem prevajalnika, medtem ko se progam v COTS tudi po nekaj urah ni prevedel (Tabela 6).

Pristop	Jezik	Čas	Rang
source-to-source	java	7min 41s 021ms	8
	Lisa	5min 23s 075ms	5
	haskell	8s 980ms	3
makro procesiranje	C++	57s 887ms	4
vgrajeni pristop	haskell	5s 756ms	2
interpreter/prevajalnik	java	7min 30s 951ms	7
generator prevajalnikov	Lisa	5min 49s 902ms	6
	SmaCC	7min 59s 257ms	9
razširjen prevajalnik	**C#**	**119ms**	**1**
COTS	XML	N/A	

Tabela 6: Primerjava hitrosti prevajanja DSL programa

8.3 Odločitev med pristopi

Primerjava različnih pristopov implementacij domensko specifičnih jezikov je nujno večplastna. Da bi le-te lahko pošteno primerjali, se moramo v prvi meri zavedati svojih potreb in pričakovanj. Različni pristopi imajo različna razmerja prednosti in slabosti. Pri odločitvi nam lahko pomagajo nekatere metrike, ki ocenijo pristope glede na specifične aspekte. Da bi prišli do končne odločitve, pa moramo *vzporedno* ovrednotiti najmanj tri stvari: svoje zahteve in pričakovanja, različne metrike in potencialne posledice odločitve. Za ustrezne odločitve so seveda potrebne izkušnje, ki pa jih lahko dobimo samo tako, da se učimo postopoma in na manjših projektih. Postopno učenje pa neizogibno pomeni tudi to, da je potrebno storiti določeno število napak, ki nas pomagajo usmerjati v prihodnje.

9 Sklep

"The imagination of nature is far, far greater than the imagination of man."
- Richard Feynman

Implementacija z razširjanjem odprtokodnega prevajalnika se je izkazala za potencialno zelo zanimivo možnost. V primerjavi z nekaterimi drugimi pristopi je razmeroma enostavna – ob pomembni predpostavki, da je prevajalnik jasno napisan in lepo strukturiran. Velika prednost je tudi, če je prevajalnik za nek jezik napisan v tem jeziku samem. To nam omogoči uporabo že narejenih konstruktov in olajša implementacijo novih. Če je pomembna časovna zahtevnost prevajanja domensko specifičnega programa, se nam še posebej izplača razmišljati v tej smeri. Seveda pa to še zdaleč ne pomeni, da je treba to možnost vzeti v obzir kot prvo možno izbiro – prava pot je celovit pogled na različne možnosti ter končna izbira glede na upoštevanje čim večjega števila parametrov, kar pa brez dvoma ni lahko. Odtehtati je treba prednosti in slabosti različnih pristopov in se odločiti za tistega, ki se kaže kot najbolj primeren za specifičen problem, ki ga rešujemo.

Implementacija domensko specifičnega jezika v prevajalnik lahko sčasoma postane rutinsko opravilo, za načrtovanje novih jezikov pa še vedno potrebujemo velike izkušnje s tega področja. Pred odločitvijo o razvoju novega programskega jezika je smotrno pretehtati vse argumente ter biti popolnoma seznanjen z ekonomskimi in drugimi vidiki odločitve.

Čeprav je potreba po razvoju domensko specifičnih jezikov prisotna, ni na voljo veliko ustrezne literature s tega področja. Raziskave so zato dobrodošle in je treba z njimi nadaljevati. Skozi čas se pojavljajo nove možnosti za razvoj domensko specifičnih jezikov, saj je to področje zelo dinamično in vsaka nova paradigma implementacije je lahko del podlage za naslednjo.

10 Literatura

[1] John R. Searle: *Consciousness*, Scientific American - Feb 1995

[2] John R. Searle: *Is the Brain a Digital Computer?*
http://www.ecs.soton.ac.uk/~harnad/Papers/Py104/searle.comp.html

[3] Scientific American - Jan 1990, *Is the Brain's Mind a Computer Program*, John R. Searle

[4] Marjan Mernik, Jan Heering, Anthony M. Sloane, *When and How to Develop Domain-Specific Languages*, CWI Technical Report 2003

[5] Marjan Mernik, Viljem Žumer, *Principi programskih jezikov*, FERI, Maribor, 2001

[6] Tomaž Kosar, Pablo E. Martínez López, Pablo A. Barrientos, Marjan Mernik: Experiencing diverse implementation approaches for Domain Specific Languages,
članek je v postopku recenzije pri reviji Software Practice & Experience

[7] David Krmpotić, Tomaž Kosar, Marjan Mernik: *Extending open compilers*, Proceedings of the 27th International Conference on Information Technology Interfaces, June 20-23 2005 Dubrovnik, Croatia

[8] J.E. Sammet, *Programming Languages: History and Fundamentals,* Prentice Hall, 1969

[9] R. L. Wexelblat, editor. *History of Programming Languages,* Academic Press, 1981

[10] B. A. Nardi, *A Small Matter of Programming: Perspective on End User Computing,* MIT Press, 1993

[11] T. J. Bergin and R.G. Gibson, editors, *History of Programming Languages II,* ACM Press, 1996

[12] J. Martin, *Fourth-Generation Languages*, Prentice Hall, 1985, Vol. I: Principles, Vol II: Representative 4GLs

[13] Edward B. Burger, Michael Starbird, *The Heart Of Mathematics: An Invitation to Effective Thinking, second edition*, Key College Publishing, 2005

[14] Neuroscience Manual, The Washington University School of Medicine, *http://thalamus.wustl.edu/course/*

[15] The Hello World Collection, *http://www.roesler-ac.de/wolfram/hello.htm*

[16] Don Batory, Jacob Neal Sarvela, Axel Rauschmayer, *Scaling Step-Wise Refinement*, Proceedings of the 25th International Conference on Software Engineering, 2003.

[17] John R. Searle, *Philosophy of mind,* The Teaching Company

[18] Richard Wolfson, *Einstein's Relativity and the Quantum Revolution*, The Teaching Company

[19] Mono Project, *http://www.mono-project.com*

[20] Greg Michaelson: *An Introduction To Functional Programming Through Lambda Calculus,* Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, 1988

[21] D. Weiss and C. T. R. Lay: *Software Product Line Engineering.* Addison Wesley, 1999

[22] D. J. Kuck, Platform 2015 software: Enabling innovation in parallelism for the next decade. *Technology@Intel Magazine,* pages 1-9, April 2005

[23] W. Frakes, R. Prieto-Diaz, and C. Fox. DARE: Domain analysis and reuse environment. *Annals of Software Engineering*, 5:125-141. 1998

[24] R. N. Taylor, W. Tracz, and L. Coglianese, Software development using domain-specific software architectures, *ACM SIGSOFT Software Engineering Notes,* 20(5):27-37. 1995

[25] D. Weiss and C. T. R. Lay, *Software Product Line Engineering*, Addison-Wesley, 1999.

[26] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990

[27] R. A. Falbo, G. Guizzardi, and K. C. Duarte, An ontological approach to domain engineering, In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, pages 351-358, ACM, 2002.

[28] M. Simos and J. Anthony, Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 94-102. IEEE Computer Society, 1998

[29] Arie van Deursen, Paul Klint: *Domain-Specific Language Design Requires Feature Descriptions*, Journal of Computing and Information Technology 10(1):1-17, 2002

[30] Charles Babbage on Wikipedia, *http://en.wikipedia.org/wiki/Charles_Babbage*

[31] A. J. Albrecht and J. E. Gaffney, Jr., *Software function, source lines of code, and development effort prediction: A software science validation,* IEEE Transactions on Software Engineering, 9(6):639-648, November 1983.

[32] S. Schleimer, D. S. Wilkerson, and A. Aiken. *Winnowing: local algorithms for document fingerprinting*. In ACM, editor, Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 76, 85. ACM Press, 2003

[33] M. Lenič, J. Brest, E. Avdičaušević, M. Mernik, V. Žumer. *Information system for laboratory work management. In Proceedings of the 5th Euromedia conference 2000 (Euromedia'2000)*, pages 245, 249. SCS Europe BVBA, cop., 2000.