PD.TVZ.HR
POLYTECHNIC & DESIGN Vol. 11, No. 3, 2023.
Mrežna inačica: ISSN 2459-6302
Tiskana inačica: ISSN 1849-1995
DOI: 10.19279/TVZ.PD.2023-11-3-03

LEGO++: PROGRAMSKI OKVIR ZA IMPLEMENTACIJU DOMENSKIH JEZIKA S UNAPRIJED DEFINIRANOM SINTAKSOM

LEGO++: A FRAMEWORK FOR IMPLEMENTING DOMAIN-SPECIFIC LANGUAGES WITH PREDEFINED SYNTAX

Aleksandar Stojanović[1], Željko Kovačević[1], Silvio Plehati[1], Branimir Barun[2]

[1]Tehničko veleučilište u Zagrebu, Vrbik 8, 10000 Zagreb, Hrvatska
[2]Ministarstvo obrane Republike Hrvatske (MORH), Trg kralja Petra Krešimira IV br. 1, 10000 Zagreb, Hrvatska

SAŽETAK

Implementacija domenskih jezika može biti zahtjevan zadatak jer obuhvaća poznavanje domene, definiranje sintakse i semantike te razvoj interpretera ili prevodioca. Nadalje, proširivanje takvih jezika novim sintaksnim oblicima, kao što je dodavanje novih naredbi, može zahtijevati složene modifikacije u implementaciji. U ovom radu opisan je programski okvir Lego++ koji olakšava implementaciju domenskih jezika upotrebom konzistentne sintakse jezika i implementacijom koja omogućava dodavanje novih naredbi bez modificiranja samog interpretera. Nadalje, korisnik (programer) ne mora poznavati detalje rada interpretera nego samo sučelje koje omogućava pristup elementima izvornog koda, kao što su vrijednosti parametara naredbe ili rezultat izvršenja bloka naredbi, kao i apstraktnom sintaksnom stablu izvornog koda. Ovaj programski okvir implementiran je u jeziku C++ i po performansama je usporediv s drugim sličnim programskim okvirima ili implementacijama.

Ključne riječi: *Domenski jezik, interpreter, proširljiv, konzistentna sintaksa*

ABSTRACT

The implementation of domain languages can be a demanding task because it includes knowledge of the domain, definition of syntax and semantics, and development of the interpreter or compiler. Furthermore, extending such languages with new syntax forms, such as adding new commands, may require complex implementation modifications. This paper describes a programming framework Lego++ that facilitates the implementation of domain languages by using a consistent language syntax and an implementation that enables the addition of new commands without modifying the interpreter itself. Furthermore, the user (programmer) does not have to know the details of the interpreter's inner working, only the interface that provides access to the source code elements, such as command parameter values or the result of a command block execution, as well as the abstract syntax tree of the source code. This framework is implemented in C++ and its performance is comparable to other similar program frameworks or implementations.

Keywords: *Domain language, interpreter, extensible, consistent syntax*

1. UVOD
1. INTRODUCTION

Domenski jezici (engl. *Domain Specific Languages* ili *DSL*) [1] računalni su jezici specijalizirani za određenu domenu primjene, kao što je razvoj web aplikacija, rad s bazama podataka ili jezici za kontrolu uređaja. Tako primjerice, u SQL jeziku ne pišemo algoritam kojim želimo doći do rezultata (kao što je to slučaj s programskim jezicima treće generacije), već tek pišemo što želimo dobiti kao rezultat (jezici

četvrte generacije). DSL-ovi su napravljeni tako da se njima mogu izraziti koncepti i operacije određene domene na jasniji i koncizniji način od jezika opće namjene (engl. *General Purpose Languages* ili *GPL*). DSL-ovi nude nekoliko prednosti za razvoj specifičan za domenu, kao što je povećana produktivnost, poboljšana kvaliteta i komunikacija, te jednostavnije održavanje. Tipičan proces razvoja domenskog jezika uključuje sljedeće korake [2]:

1. Odluka (treba li ići u razvoj novog DSLa?)
2. Analiza (identifikacija domene problema i informacija)
3. Dizajn (karakteristike novog DSLa sa stajališta sintakse i semantike)
4. Implementacija (izrada interpretera ili prevodioca)

S tehničke strane, koraci 3 i 4 su najzahtjevniji jer uključuju razvoj domenskog jezika i njegovu implementaciju.

U ovom radu opisan je programski okvir Lego++ čiji je cilj omogućiti proširivanje jezika novim naredbama (ne samo funkcijama), kao što su one za uvjetno izvršavanje, petlje ili jednostavne naredbe koje samo primaju određeni broj parametara, bez da to zahtijeva modifikaciju gramatike jezika i njegove implementacije. Lego++ sastoji se od sljedećeg:

- Ugrađenog programskog jezika koji ima implementirane osnovne naredbe, ali koje se po potrebi mogu modificirati ili ukloniti;
- Korisničkog sučelja ili API (engl. *Application Programming Interface*) za proširivanje programskog jezika;
- Interpretera za programski jezik u koji se mogu dodati nove naredbe i koji se može kontrolirati preko korisničkog sučelja.

Ovo sve zajedno nazivamo „programskim okvirom“ (engl. *framework*) jer već postoji osnovna funkcionalnost koja se po potrebi može proširivat i prilagođavat specifičnim potrebama, slično kao programski okviri za razvoj web aplikacija. Lego++ razvijen je u svrhu izrade domenskog jezika za uniformni pristup podacima iz heterogenih izvora podataka (integracija podataka). Lego++ je potpuno novi programski okvir u cijelosti razvijen i izrađen od strane autora i, prema saznanju autora, jedinstven u pristupu izradi domenskih jezika.

Neke moguće primjene ovog programskog okvira su sljedeće:

1. Izrada domenskih jezika s unaprijed definiranom sintaksom.
2. Kao komandna linija za softverske sustave u svrhu upravljanja, konfiguriranja ili automatiziranja određenih aktivnosti tog sustava.
3. Kao tekstualno sučelje za razne programske alate.
4. Kao ugradbeni skriptni jezik za programe pisane u jeziku C++ (na primjer, računalne igre) gdje bi se Lego++ koristio za konfiguraciju i proširenje takvih programa.
5. Kao generator HTML sadržaja (zbog za to pogodne sintakse).

Ovaj je rad organiziran na sljedeći način. U dijelu 2 prikazana je sintaksa jezika i njegova gramatika. U dijelu 3 opisan je princip rada njegovog interpretera, a u dijelu 4 pokazani su primjeri definiranja novih naredbi, zajedno s prikazom nekih ključnih funkcionalnosti. U dijelu 5 napravljena je kratka usporedba performansi izvršavanja izvornog kod pisanog u ovom jeziku s tri druga jezika, *PowerShell* [3], *Lua* [4] i *Bash* [5]. U dijelu 6 obrazloženi su rezultati i opisane neke mogućnosti poboljšanja performansi, a u dijelu 7 nalazi se zaključak s opisom ciljeva daljnjeg razvoja ovog programskog okvira.

2. SINTAKSA I SEMANTIKA JEZIKA
2. SYNTAX AND SEMANTICS OF THE LANGUAGE

Ovisno o jeziku i njegovoj implementaciji, promjene u sintaksi jezika (primjerice, dodavanje novog konstrukta) zahtijeva i promjene u njegovoj implementaciji, kako u dijelu za sintaksnu analizu (parsiranje) tako i u dijelu za izvršavanje ili prevođenje [6]. Kao jedan primjer možemo uzeti tipičan oblik naredbe for u jeziku C++ [7]:

```
for (int i = 0; i < n; ++i) …
```

Programer nema mogućnosti sâm definirati ovakvu naredbu kao što može definirati neku

funkciju zato jer ona ima specifičnu sintaksu koja je ugrađena u prevodioc programskog jezika. U ovakvim jezicima mnogi konstrukti imaju zasebnu sintaksu, odnosno sintaksa takvih jezika nije konzistentna. Na primjer, dva broja zbrajamo s `a + b`, dok neku funkciju, recimo *zbroji*, pozivamo sa `zbroji(a, b)`. Dakle, ako bi željeli biti konzistentni u sintaksi trebali bi (u kontekstu ovog primjera) usvojiti jednu od dviju alternativa: operaciju kao što je „+“ pisati kao `+(a, b)` ili funkciju kao što je `zbroji` pozivati s `a zbroji b`.

Jedna od temeljnih osobina Lego++ programskog okvira je konzistentna sintaksa jezika koji je njime implementiran. Da bi se izbjegla potreba za modifikacijom gramatike jezika pri dodavanju novih naredbi, svaki konstrukt ovog jezika ima sintaksu oblika

$$\{naredba\ parametar_1\ parametar_2 \ldots parametar_n\} \quad (2.1)$$

Svaki $parametar_i$ može po potrebi biti zasebna naredba. Oblik (2.1) predstavlja jedan *izraz* (engl. *expression*) odnosno konstrukt koji po izvršenju vraća neku vrijednost. Ovakva sintaksa ima prednost u tome što svaki konstrukt jezika ima isti oblik, t.j., nema razlike između ugrađenih i korisnički definiranih naredbi. Sličan koncept, ali s malo drugačijom sintaksom, primijenjen je i za programske jezike kao što je Lisp [8] i njegovi dijalekti.

Nedostatak ovakve sintakse je u tome što se neki konstrukti, kao što su aritmetičko-logički izrazi, moraju pisati u prefiksnoj notaciji, primjerice `{+ a b}` umjesto `a+b`, jer „+“ je varijabla koja sadrži naredbu zbrajanja, a *a* i *b* su parametri, što odgovara obliku (2.1). Ako želimo da se, recimo, na mjestu *b* nalazi neki podizraz, kao što je `potencija x y`, onda ga također moramo staviti u vitičaste zagrade: `{+ a {potencija x y}}`. Zagrade su neophodne jer one označavaju početak i kraj izraza; bez njih bi sličan izraz mogao izgledati kao `opr a potencija x y`, gdje se postupkom sintaksne analize ne bi moglo samo na osnovu ovakvog zapisa utvrditi šta je varijabla koja sadrži naredbu, a šta varijabla koja sadrži neki podatak. Na primjer, je li *potencija* varijabla koja sadrži naredbu ili je samo parametar naredbe *opr*? Općenito, svaki se podizraz mora nalaziti unutar vitičastih zagrada.

Primjer 1 prikazuje gramatiku jezika Lego++ u EBNF notaciji (u tom primjeru nisu prikazane produkcije za neterminale *symbol, string, int, double* i *bool* jer je njihova sintaksa trivijalna).

Primjer 1 *Glavni dio gramatike jezika Lego++ u EBNF notaciji.*

Example 1 *The main part of the Lego++ grammar in EBNF notation.*

```
block     = statement, {statement};
statement = command, ("|", statement | ";");
command   = parameter, {parameter};
parameter = "{", block, "}" | symbol | string |
double | int | bool;
```

Kao još jedan primjer, izraz *if* sastoji se od naredbe *if* i tri parametra: uvjeta, konzekvence i (opcionalno) alternative:

```
if uvjet konzekvenca else alternativa
```

Jedan primjer ovog izraza je

```
if {> x y} {print x} else {print y}
```

Semantika ove naredbe je kao i u drugim programskim jezicima: ako je uvjet zadovoljen izvršava se konzekvenca, u suprotnom izvršava se alternativa (ako je definirana). Iza *if* nalazi se prvi parametar koji predstavlja uvjet, *{> x y}*. Nakon toga slijedi parametar za konzekvencu *{print x}*, a iza njega parametar alternative *{print y}*. Ovaj izraz ispisuje vrijednost varijable *x* ako je $x > y$, u suprotnom ispisuje vrijednost varijable *y*. Svaki od ovih podizraza u vitičastim zagradama može sadržavati više drugih izraza. Simbol *else* služi samo kao oznaka alternative jer naredba *if* može imati više parova *uvjet/konzekvenca*. Općenito, semantika svake naredbe određena je implementacijom te naredbe, što je detaljnije opisano u idućem poglavlju.

Primjer 2 prikazuje implementaciju Quicksort algoritma [9]. Naredbom *fn* definira se nova funkcija. Ta se naredba sastoji od dva ili više parametara, zavisno od toga koliko parametara ima funkcija koja se njome definira. Prvi parametar je ime funkcije i on je obavezan. Nakon njega mogu uslijediti nula ili više parametara funkcije, a na kraju se obavezno mora nalaziti parametar koji predstavlja tijelo funkcije.

Još jedan nedostatak ovakve sintakse je u tome što zahtijeva zagrade kao oznaku početka i kraja

podizraza. Međutim, taj je zahtjev djelomično prevaziđen na sljedeći način:

1. Početni (vanjski) izraz nema zagrade (što se vidi iz gramatike);
2. Samo se podizrazi moraju nalaziti u zagradama;
3. U mnogim se slučajevima uvjet 2) rješava time što se nakon učitavanja izvornog koda, ali prije sintaksne analize, na mjesto zareza automatski dodaju vitičaste zagrade po određenom principu.

Primjer 2 *Quicksort algoritam u Lego++ jeziku.*

Example 2 *The Quicksort algorithm in Lego++.*

```
fn partition a low high {
    let pivot, at a high
    let i, - low 1
    for j low, - high 1 {
        if {≤ {at a j} pivot} {
            ++ i
            swap a i j
        }
    }

    swap a {+ i 1} high
    return, + i 1
}

fn qsort a low high {
    if, < low high {
        let p, partition a low high
        qsort a low, - p 1
        qsort a {+ p 1} high
    }
}
```

Zarez nije dio sintakse ovog jezika. Nakon učitavanja ovakvog izvornog koda, zarezi će biti zamijenjeni vitičastim zagradama. Na primjer, u drugom redu, izraz

```
let pivot, at a high
```

biti će transformiran u `let pivot {at a high}`. Općenito, na mjesto svakog zareza postavlja se otvorena vitičasta zagrada, a zatim na kraju reda, prije druge otvorene ili zatvorene vitičaste zagrade ili znaka „|“, postavlja se zatvorena vitičasta zagrada. Na primjer, u redu `qsort a {+ p 1} high`, podizraz `{+ p 1}` mora ostati u zagradama jer bi `a, + p 1 high` bio transformiran u `a {+ p 1 high}`, odnosno zagrade bi obuhvatile i parametar *high* čime bi dobili pogrešan podizraz. Isto tako, znak „;“ dodaje se automatski na kraj svakog izraza unutar bloka.

3. IMPLEMENTACIJA
3. IMPLEMENTATION

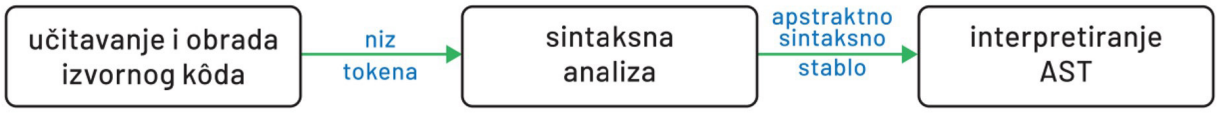
Interpreter za Lego++ implementiran je u programskom jeziku C++. Slika 1 prikazuje postupak izvršavanja izvornog koda pisanog u Lego++ jeziku. S obzirom da ovaj jezik nema složenu gramatiku, parser je implementiran metodom *rekurzivnog spusta* (engl. *recursive descent parsing*) [6] direktno u kodu interpretera, bez upotrebe parser generatora. Interpretacija koda izvodi se direktno nad apstraktnim sintaksnim stablom (engl. *Abstract Syntax Tree* ili *AST*) [6] [10].

Lego++ ima naredbu *ast* kojom se može dobiti apstraktno sintaksno stablo za zadani izvorni kôd. Primjer 3 prikazuje ispis koji bi dobili ako u komandnu liniju unesemo `ast {while {< x 5} {print x; inc x}}`.

Primjer 3 *Primjer apstraktnog sintaksnog stabla.*

Example 3 *An abstract syntax tree example.*

```
*<BLOCK>
  *<STATEMENT>
    *<COMMAND>
        #<SYMBOL>: while (global symbol)
        *<BLOCK>
            *<STATEMENT>
                *<COMMAND>
                    #<SYMBOL>: < (global symbol)
                    #<SYMBOL>: x
                    #<INT>: 5
        *<BLOCK>
            *<STATEMENT>
                *<COMMAND>
```



Slika 1 *Postupak izvršavanja izvornog koda.*

Figure 1 *The source code execution steps.*

```
            #<SYMBOL>: print (global symbol)
            #<SYMBOL>: x
        *<STATEMENT>
            *<COMMAND>
            #<SYMBOL>: ++ (global symbol)
            #<SYMBOL>: x
```

Za naredbu *ast* nije nužno da je naredba *while* definirana. Parser samo provjerava sintaksu, odnosno to da li ulazni kôd po svojoj stukturi odgovara gramatici jezika. Prema tome, on će prvi simbol, *while* (ili bilo koji drugi), uzeti kao naziv naredbe, a sve što slijedi iza njega kao parametre te naredbe.

Ovakav prikaz apstraktnog sintaksnog stabla može pomoći u implementaciji novih naredbi. Promatrajući Primjer 3, ako bi htjeli implementirati naredbu kao što je *ako* (uvjetno izvršavanje kao kod naredbe *if*), onda vidimo da će se kod te naredbe nakon simbola *ako* (naziv naredbe) nalaziti tri *bloka* (dio koda unutar vitičastih zagrada): prvi za uvjet, drugi za konzekvencu i treći za alternativu. To znači da bi prvo morali izvršiti blok koji predstavlja uvjet; ako bi taj blok vratio rezultat *true* onda bi izvršili drugi blok, t.j. konzekvencu, a ako bi vratio *false* izvršili bi treći blok, t.j. alternativu (ako postoji). U sljedećem dijelu opisano je kako se definiraju nove naredbe, s par primjera postojećih naredbi koje su implementirane na taj način.

4. DEFINIRANJE NOVIH NAREDBI
4. DEFINING NEW COMMANDS

U Lego++ programskom okviru svaka je naredba objekt i implementirana je u jeziku C++ kao klasa ili struktura. Da bi se omogućila implementacija naredbi koje trebaju pristup parametrima, čvorovima apstraktnog sintaksnog stabla, tablici simbola i drugim dijelovima interpretera, svaka naredba ima na raspolaganju API koji se nalazi u baznoj klasi *Command*. Primjer 4 prikazuje naredbu koja računa potenciju broja.

Primjer 4 *Naredba za spajanje stringova.*

Example 4 *A command for string concatenation.*

```
struct Potencija : Command {
        Potencija () : Command{
                "potencija",
                DOUBLE,
                Params(this,
                        M(DOUBLE|INT, "baza"),
                        M(DOUBLE|INT, "eksp")
                ),
                "Vraća bazu potenciranu na eksponent.",
                "math"
        } {}

        void exec(Value* r, Params& param) override {
          double v = pow(param["baza"]→double_value,
                         param["eksp"]→double_value);
          set_result(v, r);
        }
};
```

U konstruktoru naredbe *Potencija* inicijaliziran je bazni dio objekta. Prvo je zadan simbol naredbe, „potencija“ (koji će označavati tu naredbu u izvornom kodu), zatim povratni tip, *DOUBLE*, a nakon toga tip obaveznih parametara (tip *M* označava obavezne parametre jer postoje i neki drugi kao što su opcionalni parametri i zastavice). S obzirom da ova naredba prima dva cijela ili realna broja kao parametre, tip oba parametra je *DOUBLE* ili *INT*. Za *ili* dio koristi se *bitni* (engl. *bitwise*) operator „|“. Kao idući parametar inicijalizacije zadan je opis naredbe koji se može dobiti naredbom *info* u komandnoj liniji (kao `info math.potencija`) te na kraju imenski prostor kojem naredba pripada.

Ovdje vidimo da tipove parametara kao i njihov broj nije potrebno provjeravati u samoj naredbi – za to će se pobrinuti interpreter koji će pri njenom izvršenju baciti iznimku ako povratni tip naredbe, tip parametara ili njihov broj ne odgovara specifikaciji zadanoj u konstruktoru.

Virtuelnu funkciju *exec* poziva interpreter kada u apstraktnom sintaksnom stablu naiđe na čvor koji predstavlja naredbu. To je apstraktna funkcija bazne klase koju svaka naredba mora implementirati i u kojoj se nalazi kôd koji sačinjava njenu semantiku.

Jedan primjer korištenja naredbe „potencija“ je sljedeći (prikazano u komandnoj liniji):

```
> math.potencija 2 3
⟹ 8.0 <DOUBLE>
```

Parametri naredbe ne moraju biti literali dotičnog tipa vrijednosti, kao 2 ili 3, već mogu biti i podizrazi. Na primjer,

```
> math.potencija {+ 1 2} 2
⟹ 9.0 <DOUBLE>
```

U gornjem primjeru prvi parametar je rezultat naredbe „+“, a drugi je trivijalan izraz. Prema tome, da bi se dobile dvije vrijednosti za ova dva parametra naredbe „potencija“, interpreter mora evaluirati podizraze kojima su ti parametri specificirani. To se dešava automatski, bez intervencije korisnika (onoga tko definira ovakvu naredbu). Rezultat naredbe na kraju se smješta u prvi parametar funkcije *exec* funkcijom *set_result*.

Općenito, implementacija ovakvih naredbi ne zahtijeva poznavanje detalja rada intepretera; korisnik samo treba poznavati API koji omogućava pristup raznim elementima izvornog koda potrebnim za njenu implementaciju.

Kao posljednji korak implementacije naredbe, potrebno ju je registrirati u interpreter (Primjer 5).

Primjer 5 *Primjer registriranja naredbe.*

Example 5 *Example of command registration.*

```
Interpreter rt;
rt.register_cmd<Potencija>();
```

Ovo interpreteru daje kontrolu nad instanciranjem i životnim vijekom objekata koji predstavljaju naredbe.

Primjer 6 prikazuje implementaciju naredbe za petlju while koja ima oblik `{while` *`uvjet tijelo-petlje`*`}`.

Primjer 6 *Implementacija naredbe while.*

Example 6 *The implementation of command while.*

```
01 struct While : Command {
02 While() : Command{
03     "while",
04     ANY,
05     Params(2),
06     "Izvršava tijelo naredbe dok je uvjet True."
07 } {}

08 void exec(Value* result) override {
09     BlockNode* body = param_node(2);

10     Value body_result;
11     Value cond;
12     for (;;) {
13       param_val(BOOL, 1, &cond);
14       if (cond.bool_value) {
15          execute_block(body, &body_result);
16          if (get_return_flag()) {
17             set_result(*get_return_value(), result);
18             return;
19          }
20       }
21       else {
22          break;
23       }
24     }

25     set_result_nil(result);
26 }
27 };
```

U retku 9 *param_node* daje čvor apstraktnog sintaksnog stabla koji sadrži tijelo petlje (kao što Primjer 3 pokazuje, to je drugo dijete nakon simbola *while* čvora COMMAND). Nakon toga petlja se izvodi unutar *for* petlje. U retku 13 odredi se rezultat uvjeta – ako on vrati *True* izvršava se blok tijela petlje (redak 15), u suprotnom prekida se izvršavanje petlje *while* (redak 22). U retku 16 provjerava se je li unutar *while* petlje izvršena naredba *return*. S obzirom da ta naredba mijenja tijek izvršavanja programa, ona mora biti implementirana u samom interpreteru, odnosno ne može se u potpunosti implementirati kao objekt tipa *Command*. Prema tome, ako je interpreter naišao na naredbu *return*, postavlja unutrašnji indikator za to koji se ovdje provjerava u retku 16. Ako je taj indikator postavljen, u redcima 17 i 18 prekida se izvršavanje petlje, a ujedno se, ako je definiran, vraća rezultat naredbe *return*.

5. REZULTATI
5. REZULTATI

U ovom dijelu prikazane su performanse Lego++ interpretera u odnosu na tri druga programska ili komandna jezika: Lua, Powershell i Git Bash[1]. Iako nismo našli slične programske okvire, ovi su jezici odabrani jer imaju barem neke sličnosti po namjeni i/ili načinu implementacije u odnosu na Lego++. Lua je programski jezik koji se često koristi za izradu skriptova koji se umeću u druge sustave, kao što su računalne igre, pa zbog toga postoje određene sličnosti u namjeni s Lego++. PowerShell je jezik prvenstveno predviđen za administraciju sustava i može se proširivati dodavanjem novih komandi izradom odgovarajućih klasa u .NET okruženju. PowerShell i Lego++ su najsličniji po implementaciji jer oba sustava omogućuju

1 Mjereno na računalu s procesorom Intel i7/2.7 GHz, 32GB radne memorije na 64-bitnom operacijskom sustavu Windows 10 i Microsoftovim C++ prevodiocem v. 19.36.32532 (s uključenom optimizacijom koja preferira brzinu koda).

Algoritam	Lego++	Lua	Powershell	Git Bash
Quicksort (1000 elemenata)	0.013	0.002	0.21	0.48
Fibonacci rekurzivno (12ti Fibonaccijev broj)	0.004	0.000001	0.014	5.7
Fibonacci rekurzivno (30ti Fibonaccijev broj)	4.58	0.12	62.18	Nepoznato
Bubble sort (2000 elemenata)	1.01	0.17	0.25	11.7
Broj prostih brojeva u intervalu 1-100.000	2.33	0.09	2.8	34.7

Tablica 1 *Performanse Lego++ u odnosu na tri druga jezika.*

Table 1 *Lego++ performanse compared to three other languages.*

proširivanje jezika na sličan način, ali je osnovna razlika u tome da PowerShell nije predviđen za direktno ugrađivanje u druge sustave. Bash je relativno stari komandni jezik koji se koristi na Unix/Linux operacijskim sustavima, ali upotrebljava se i za Git sustav (kao Git Bash inačica). S obzirom da se i Lego++ može upotrebljavati kao komandni jezik i ovdje postoje određene sličnosti.

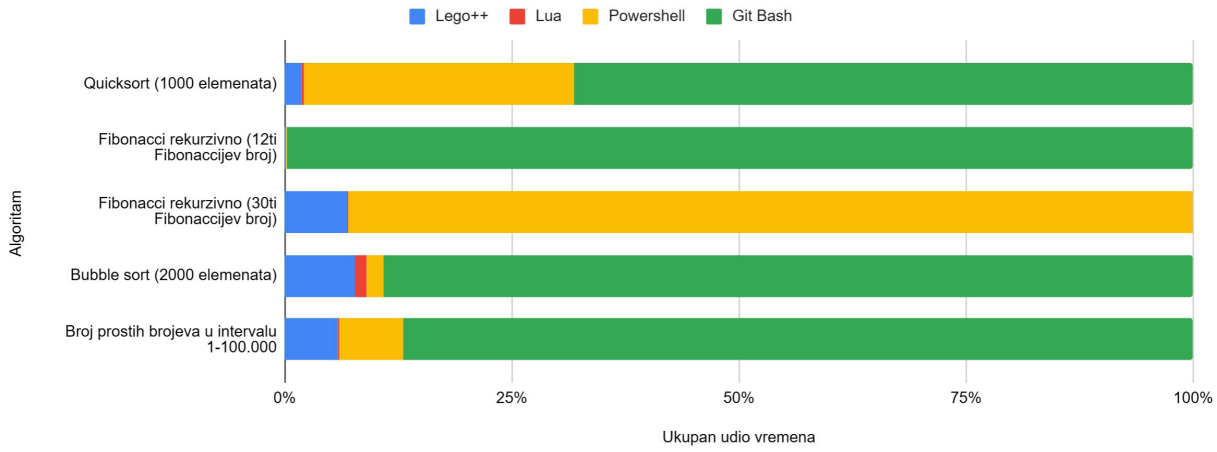
Tablica 1 prikazuje performanse Lego++ interpretera u odnosu na ova tri druga jezika kroz implementaciju četiriju algoritama: Quicksort, bubble sort, fibonaccijevi brojevi i traženje prostih brojeva [9].

Algoritmi su odabrani tako da mjere performanse poziva funkcija kroz rekurziju (Quicksort i Fibonaccijevi brojevi) i iterativno izvršavanje (prosti brojevi i bubble sort). U sva četiri jezika ovi su algoritmi implementirani na isti način da bi se minimizirale razlike u načinu njihovog izvođenja.

Slika 2 prikazuje ove podatke tako da se lakše vidi ukupan udio vremena za pojedinačni algoritam i jezik u odnosu na ukupno vrijeme za taj algoritam kroz sva četiri jezika. Broj elemenata za sve algoritme odabran je tako da ih svaki od ovih jezika može izvršiti. Kod većeg broja elemenata za Quicksort, kao što je 5.000, interpreter jezika Git Bash nije mogao dati rezultate zbog nadmašenja veličine memorije rezervirane za stog (koja se koristi za pozive funkcija). U slučaju 30og Fibonaccijevog broja Git Bash nije mogao završiti unutar prihvatljivog vremenskog perioda pa je na tom mjestu vrijeme označeno kao nepoznato. Iz tog razloga prikazan je rezultat za isti algoritam, ali manju vrijednost (12ti Fibonaccijev broj).

6. DISKUSIJA
6. DISCUSSION

Rezultati pokazuju da Lego++ ima performanse najbliže onima kod Powershell jezika, znatno bolje od Git Bash, ali i znatno lošije od Lue. Iako je Lua pokazala najbolje rezultate, treba



Slika 2 *Ukupan udio vremena svakog jezika za pojedinačne algoritme.*

Figure 2 *Total time share taken by each language for individual algorithms.*

uzeti u obzir to da taj jezik (kao ni Git Bash) nije predviđen za proširenja slična onakvim za kakva su napravljeni Lego++ i djelomično Powershell, odnosno dodavanje novih naredbi u kodu sustava putem novih klasa.

Powershell se pokazao bržim od Lego++ za iterativne algoritme, specifično za bubble sort kod kojeg je bio brži oko četiri puta, ali znatno sporiji za rekurzivne algoritme. Za Quicksort bio je sporiji oko 15 puta, slično i za izračunavanje *n*-tog Fibonaccijevog broja. Čini se da pozivi funkcija u Powershell sustavu iz nekog razloga zahtijevaju više vremena, ali dublja analiza tog sustava ovdje nije napravljena. To se vidi i u algoritmu za proste brojeve kod kojeg je Powershell malo sporiji od Lego++. Iako taj algoritam nije implementiran rekurzivno, kod njega se za svaki broj u zadanom intervalu poziva funkcija koja provjerava je li on prost. To je u kontrastu s implementacijom bubble sort algoritma koji se sastoji od dvije ugnježđene *for* petlje, bez dodatnih poziva funkcija.

Jedan način na koji bi mogli poboljšati performanse Lego++ interpretera je da se izbjegne obilaženje apstraktnog sintaksnog stabla jer se to sastoji od velikog broja poziva funkcija. To bi mogli postići na tri načina:

1. Prevođenjem izvornog koda pisanog u Lego++ jeziku na jednostavne instrukcije odnosno *bajt kôd* (engl. *byte code*) kao što rade interpreteri za neke druge programske jezike (primjerice, Python). Ovo bi omogućilo i dodatni korak prevođenja bajt koda na strojni kôd.
2. Prevođenjem izvornog koda pisanog u Lego++ jeziku na C++ nakon čega bi se direktno generirala izvršna datoteka pomoću prevodioca za C++. Ovo se ujedno čini i kao najjednostavniji pristup jer su sve naredbe u Lego++ jeziku ionako instance C++ klasa tako da bi se takav prevedeni kôd uglavnom sastojao od poziva virtuelne funkcije *process* i nekih pomoćnih funkcija.
3. Optimiziranjem samog interpretera prethodnom obradom apstraktnog sintaksnog stabla.

Način 2) čini se kao najbolji od ova tri jer bi omogućio maksimalne performanse koje bi dobili direktnim izvršavanjem naredbi, bez obilaženja apstraktnog sintaksnog stabla.

7. ZAKLJUČAK
7. CONCLUSION

U ovom radu opisan je programski okvir Lego++ napravljen s namjerom da se olakša implementacija domenskih jezika. Sintaksa jezika koju podržava osmišljena je tako da pri proširivanju jezika novim naredbama nije potrebno definirati novi sintaksni oblik nego samo funkcionalnost nove naredbe. Svaka se naredba definira kao klasa jezika C++, a pristup elementima izvornog koda potrebnim za implementaciju novih naredbi omogućen je API sučeljem bazne klase. Performanse interpretera usporedive su s nekim relativno sličnim sustavima i shell jezicima, kao što su Powershell i Bash. Međutim, u odnosu na neke programske jezike, kao Lua, Lego++ je znatno sporiji. Dizajn ovog programskog okvira omogućava daljnje poboljšanje performansi, prvenstveno kroz translaciju izvornog koda u C++, čime bi se izbjeglo obilaženje apstraktnog sintaksnog stabla. Ovo ostaje tema budućeg istraživanja i rada na ovom sustavu.

8. REFERENCE
8. REFERENCES

[1.] M. Fowler, Domain-Specific Languages, Addison-Wesley, ISBN-13: 978-0321712943, 2011.

[2.] M. Mernik, J. Heering i A. M. Sloane, »When and How to Develop Domain-Specific Languages,« Computing Surveys, svez. 37, br. 4, pp. 316-344, 2005.

[3.] L. Holmes, PowerShell Cookbook, 4th Edition, O'Reilly Media Inc., ISBN-13: 978-1098101602, 2021.

[4.] R. Ierusalimschy, Programming in Lua, 4th ed., Lua.org, ISBN 8590379868, 2016.

[5.] C. Newham i B. Rosenblatt, Learning the bash Shell: Unix Shell Programming, 3rd. ed., O'Reilly Media, Inc., ISBN-13: 978-0596009656, 2005.

[6.] A. V. Aho, M. S. Lam, R. Sethi i J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, ISBN-13: 978-0321486813, 2006.

[7.] B. Stroustrup, The C++ Programming Language, 4th ed., Addison-Wesley, ISBN-13: 978-0275967765, 2013.

[8.] P. H. Winston i B. K. P. Horn, LISP, Prentice

Hall, ISBN-13: 978-0201083194, 1988.

[9.] T. H. Cormen, C. E. Leiserson, R. L. Rivest i C. Stein, Introduction to Algorithms, 4th ed., The MIT Press, ISBN-13: 978-0262046305, 2022.

[10.] D. P. Friedman i M. Wand, Essentials of Programming Languages, 3rd ed., The MIT Press, ISBN-13: 978-0262062794, 2008.

AUTORI · *AUTHORS*

• Aleksandar Stojanović – je viši predavač na Tehničkom veleučilištu u Zagrebu i izvodi nastavu na predmetima iz područja programiranja i algoritama i struktura podataka. Godine 1996. diplomirao je informacijske i komunikacijske znanosti na Filozofskom fakultetu sveučilišta u Zagrebu, a 1999. godine magistrirao računarstvo u SAD-u na sveučilištu Midwestern State University te je radio kao programski inženjer u području telekomunikacija, financija i energetike. Godine 2019. doktorirao je na Filozofskom fakultetu sveučilišta u Zagrebu s disertacijom pod naslovom "Metoda automatske detekcije naglašenih riječi u zvučnom zapisu". Područja interesa su mu integracija podataka, razvoj uniformnih sučelja za heterogene izvore podataka te primjena evolucijskih algoritama u rješavanju problema. Autor je knjige “Elementi računalnih programa s primjerima u Pythonu i Scali”.

Korespondencija · *Correspondence*

aleksandar.stojanovic@tvz.hr

• Željko Kovačević - Nepromjenjena biografija nalazi se u časopisu Polytechnic & Design Vol. 10, No. 4, 2022

Korespondencija · *Correspondence*

zeljko.kovacevic@tvz.hr

• Silvio Plehati - Predavač je na Tehničkom veleučilištu u Zagrebu gdje sudjeluje u nastavi iz kolegija programskog inženjerstva, objektno-orijentiranog programiranja i naprednog programiranja. Na Tehničkom veleučilištu u Zagrebu 2003. godine završio je preddiplomski studij Informatike, te 2010. godine diplomski studij Informatike. Radi u struci od 2003. godine. Izbor u zvanje asistenta stječe 2021. godine, a izbor u zvanje predavača 2023. godine.
Trenutno pohađa zadnju godinu diplomskog studija na Grafičkom fakultetu Sveučilišta u Zagrebu. Godine 2023. dobiva na Grafičkom fakultetu Dekanovu nagradu za objavljena 2 znanstvena rada A kategorije (Q1 i Q2) u 2022. i 2023. godini. Objavljuje sa koautorima stručne i znanstvene radove povezane sa grafičkim i programskim inženjerstvom, programiranjem i projektiranjem grafičkih zaštita. Područja interesa su: programiranje u C jezicima, grafički programski jezici, 2D i 3D grafika, 3D tisak, mikro (embedded) elektronika i senzori.

Korespondencija · *Correspondence*

silvio.plehati@tvz.hr

• Branimir Barun - Zaposlen je u Ministarstvu obrane Republike Hrvatske na poslovima iz područja kibernetičke sigurnosti. Asistent je na Tehničkom Veleučilištu u Zagrebu na predmetu Objektno orijentirano programiranje. Diplomirao je na Veleučilištu Velika Gorica na diplomskom studiju Informacijski sustavi. Područja interesa su mu reverzno inženjerstvo, testiranje informacijske sigurnosti i podatkovna znanost.

Korespondencija · *Correspondence*

branimir.barun@tvz.hr