TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Jaagup Russak

Programmeerimine, kasutades sümboolseid automaate ja muundureid

Bakalaureusetöö (9 EAP)

Juhendaja: Vesal Vojdani, PhD

Tartu 2019

Programmeerimine, kasutades sümboolseid automaate ja muundureid

Lühikokkuvõte:

Sümboolsed automaadid ja muundurid on lõplike automaatide laiendused, mis lubavad üleminekuid ka predikaatide ja funktsioonidega. Töö eesmärgiks on uurida ja anda ülevaade programmeerimiskeeltest, mis on ehitatud sümboolsetele automaatidele ja muunduritele. Peamiselt keskendutakse programmeerimiskeelele Bek, kuid antakse lühiülevaade ka kahest Bekiga sarnasest keelest Bex ja Fast. Töö annab ka ülevaate, milliseid programme on võimalik muundureid kasutades luua.

Võtmesõnad: automaadid, muundurid, Bek, Bex, Fast
CERCS: P175 Informaatika, süsteemiteooria

Programming using symbolic automata and transducers

Abstract:

Symbolic automata and transducers are extensions to finite automata which allow transitions with predicates and functions. The aim of this thesis is to study and give an overview of programming languages that are built on symbolic automata and transducers. The main focus is on the programming language Bek but a short overview of two similar languages Bex and Fast is given as well. The thesis also gives an overview of the scope of the programs that can be written with tools based on automata and transducers.

Keywords: automata, transducers, Bek, Bex, Fast
CERCS: P175 Informatics, system theory

Sisukord

Sissejuhatus

Veebis on mitmeid turvariske. Neist üks enamlevinud on skriptisüst (*cross-site scripting, XSS*), mille tulemusena on võimalik varastada ohvri küpsised ja seeläbi ligi pääseda ohvri andmetele. Ründe vältimiseks on võimalik veebiarendajatel kasutada puhastusfunktsioone, mis takistavad ohtliku koodi käivitamist nende veebilehel. Puhastusfunktsioonide korrektne kirjutamine on keeruline ülesanne. Sümboolsetele olekumuunduritele ülesehitatud programmeerimiskeeled võimaldavad kontrollida puhastusfunktsioonide turvalisust.

Bakalaureusetöö eesmärgiks on uurida ja anda ülevaade programmeerimiskeeltest, mis on ehitatud sümboolsetele automaatidele ja muunduritele. Peamiselt keskendutakse programmeerimiskeelele Bek, kuid antakse lühiülevaade ka kahest Bekiga sarnasest keelest Bex ja Fast.

Töö esimene peatükk annab ülevaate kasutatavast terminoloogiast. Teises peatükis kirjeldatakse põhjalikult programmeerimiskeelt Bek ning tutvustatakse sealseid võimalusi. Kolmandas peatükis antakse lühiülevaade programmeerimiskeeltest Bex ja Fast. Neljandas peatükis kirjeldatakse aine „Automaadid, keeled ja translaatorid“ jaoks loodud lisaülesandeid, mis tutvustavad automaatide rakendusi.

1 Terminoloogia

1.1 Skriptisüst

Skriptisüst (*cross-site scripting, XSS*) on turvarisk veebis, mille käigus „süstitakse“ kasutaja poolt loodud skripte veebilehtedele. Kui kasutaja on rünnet teostava koodijupi sisestanud, jääb ta ootama, kuni ohver selle käivitab. Tulemusena käivitatakse koodijupp ohvri õigustes ning on võimalik varastada näiteks ohvri küpsised. Seeläbi saab üle võtta ohvri sessiooni ja on võimalik kätte saada tundlikud andmed, mis on kasutusel vastaval veebilehel. Tihti on turvariskiga foorumid, kus kasutaja sisendit ei puhastata ehk lubatakse sisendis kasutada HTML koodi. Skriptisüsti vastu on veebilehel võimalik ennast kaitsta, kasutades puhastusfunktsioone. [1]

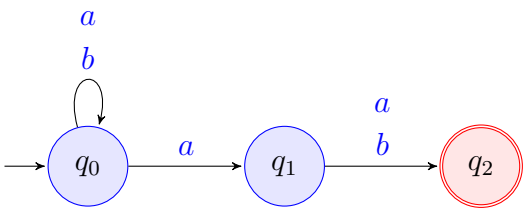
1.2 Puhastusfunktsioonid

Puhastusfunktsioonid *(sanitizer functions)* on kindla eesmärgiga sõnekodeerijad. Puhastusfunktsioonide ülesanne on kontrollida kasutaja sisendit ja eemaldada sealt sümbolid, mis võivad turvalisust ohustada. Need on laialdaselt kasutusel veebirakendustes kaitseks skriptisüstide vastu. Veebilehtedel on tavaliselt vähemalt kolm erinevat puhastusfunktsiooni: CssEncoder, UrlEncoder ja HtmlEncoder. Mõnikord kasutavad eelnimetatud lisaks ka teisi aluskodeerijaid, näiteks UrlEncoder kasutab esimeseks sammuks UTF-8 kodeerijat. [2, 3]

1.3 Lõplik automaat

Lõplik automaat on viisik $A = (Q, \Sigma, q, F, \delta)$, kus Q on lõplik olekute hulk, Σ on sisendsümbolite hulk, q on algolek, F on lõppolekute hulk ja $\delta : Q \times \Sigma \to Q$ on üleminekurelatsioon [4]. Lõplikke automaate kasutatakse sõnetuvastuseks ja mustrisobituseks. Joonisel 1 on toodud lihtne automaat, mis tunneb ära kõik sõned, mis koosnevad tähtedest a ja b ja eelviimane täht on a. Olekute hulk on $\{q_0, q_1, q_2\}$, sisendsümbolite hulk on $\{a, b\}$, algolek on q_0, lõppolekute hulk on $\{q_2\}$ ja joontega on kujutatud üleminekurelatsiooni ühest olekust teise. Joonisel on kujutatud kolme üleminekut. Algolekust on üleminekud $q_0 \times \{a, b\} \to q_0$ ja $q_0 \times \{a\} \to q_1$ ehk olekust q_0 saab olekusse q_0 tähtedega a ja b ja olekusse q_1 tähega a. Olekust q_1 saab olekusse q_2 tähtedega a

ja b ehk nende olekute vahel on üleminekurelatsioon $q_1 \times \{a, b\} \rightarrow q_2$



Joonis 1: Näide kolme olekuga lõplikust automaadist

Lõplikke automaate on kahte liiki – deterministlikud ja mittedeterministlikud. Determinism lõpliku automaadi puhul tähendab seda, et iga sümboli jaoks on ainult üks võimalik üleminek järgmisse olekusse saamiseks. Joonisel 1 toodud automaat on mittedeterministlik, kuna sümboliga a on võimalik jõuda nii olekusse q_0 kui ka olekusse q_1. [5]

1.4 Lõplik muundur

Lõplik muundur on lõpliku automaadi laiendus. Seda saab lisaks mustrisobitusele ja sõnetuvastusele kasutada ka parsimiseks ja kodeerimiseks. Lõpliku muunduri definitsioon on sarnane lõpliku automaadi definitsiooniga, aga seal on lisaks väljundsümbolite sõnastik. Lõplikuks muunduriks nimetatakse kuuikut $A = (Q, \Sigma, \Gamma, q, F, \delta)$, kus Q on lõplik olekute arv, Σ on sisendsümbolite hulk, Γ on väljundsümbolite hulk, q on algolek, F on lõppolekute hulk ja $\delta : Q \times \Sigma \rightarrow Q \times \Gamma$ on üleminekurelatsioon [3]. Seega toimib lõplik muundur sarnaselt lõpliku automaadiga, aga ei kontrolli, kas automaat võtab sõne vastu, vaid muundab sisendit vastavalt üleminekurelatsioonile. Näiteks on lõpliku muunduriga võimalik sisendist eemaldada HTML märke.

Töös tutvustatavad programmeerimiskeeled on ehitatud sümboolsetele lõplikele muunduritele. Viimane erineb lõplikust muundurist selle poolest, et üleminekud on märgendatud loogiliste valemite ja predikaatidega, mitte kindlate sümbolitega [3]. Sümboolne muundur lubab üleminekut kõikide sümbolitega, mis seda valemit rahuldavad. Töös kirjeldatavates programmeerimiskeeltes Bek, Bex ja Fast on kasutatud sümboolsete muundurite omadusi programmide analüüsimiseks.

2 Bek

Järgnev peatükk põhineb programmeerimiskeelt Bek tutvustavel lehekülgedel [6, 7] ja Hooimeijer jt artiklil [3], kui pole mainitud teisiti. Bek on programmeerimiskeel, mis on spetsiaalselt mõeldud tekstitöötlejate loomiseks ja analüüsiks. Näiteks kui on vaja eemaldada HTML märke või lisada või eemaldada kodeeringut, siis Bek võimaldab seda lihtsasti teha. Samuti võimaldab Bek kontrollida programmide ekvivalentsust, idempotentsust ja kommutatiivsust.

Tüüpiliseks Beki programmi konstruktsiooniks on iteratsioon üle kõikide sümbolite sisendsõnest. Iteratsioonisammudel vaadatakse läbi kõikvõimalikud juhud, kuidas programm erinevate sümbolite puhul käituma peaks. Vastavalt juhule väljastab programm mingi sümboli või jätab sümboli meelde. Näiteks sõne dekodeerimisel sisendist '<' väljundisse '<' saamiseks on vaja jätta sümbolid meelde ja sobiva järjestuse korral teostada transformatioon.

Programmi sõnetöötlusosale järgnevad erinevad päringud. Eraldajaks koodi ja päringute vahel on kaks võrdusmärki. Joonisel 2 toodud koodis on päringuteks *eq(), image()* ja *js()*. Päring *eq*(programm, join(programm, programm)) kontrollib idempotentsust (kas tulemused sama sisendiga ühekordsel ja kahekordsel käivitamisel on ekvivalentsed). Päring *image*(programm, sisend) väärtustab programmi antud sisendiga ning *js*(programm) genereerib Beki koodist vastava JavaScript programmikoodi.

```
program encode(input) {
  return iter(c in input){
    case (c == '<') : yield ('&','l','t',';');
    case (c == '>') : yield ('&','g','t',';');
    case (true)     : yield(c); };
}
==
eq(encode,join(encode,encode));
image(encode,"<script>");
js(encode);
```

Joonis 2: Sümbolite '<' ja '>' kodeerimine Bekis

Joonisel 2 toodud koodijupp asendab sümbolid '<' ja '>' neile vastava HTML kodeeringuga. Näiteks sisendi '<script>' puhul saab väljundiks '<script>'. Enne eraldajat '==' olev koodi-

osa teostab muutuse, pärast märki on päringud – *eq*(encode, join(encode, encode)) kontrollib idempotentsust, *image*(encode,“<script>“) väärtustab avaldise sisendiga ’<script>’ ja *js*(encode) teostab ülemineku keelde JavaScript

2.1 Programmeerimine Bekis

Beki programmi tüüpiline ülesehitus on järgnev:

```
program nimi(sisend){
   return iter (c in sisend)[algolek] {juhud}end{juhud};
}
```

Siin *c* on hetkel käsitletav sümbol, *algolek* on lokaalse muutuja deklaratsioon, *juhud* on mittetühi list võimalikest juhtudest, mida väärtustatakse tavalisel kui-siis-muidu (*if-then-else*) viisil – `case (tingimus1):avaldis1; case (tingimus2):avaldis2;` jne. Avaldistele võivad järgneda ka lõppjuhud ehk mida teeb programm, kui jõutakse lõppu. Kui lõppjuhtu ei ole täpsustatud, siis jääb selleks sisseehitatud `end {case(true): yield(); }` ehk väljundit enam ei muudeta.

Kõige tüüpilisem „Tere maailm!“ programm võiks välja näha järgmine:

```
program tereMaailm(sisend) }
    //käime läbi iga sümboli sisendis, aga väljundisse ei lisa midagi
    return iter (c in sisend) {
        case(true): yield();
    end {
    //lõppu jõudes lisame väljundisse "Tere maailm!"
    case(true): yield("Tere maailm!");
    };
}
```

Sellest programmist saab mõelda ka kui kahe olekuga muundurist, mis iga sümboli puhul väljastab tühja sõne, aga kui kõik sümbolid sisendist saavad läbi käidud, siis lisab väljundisse „Tere maailm!“.

Tihti ei piisa sellest, kui on teada ainult hetkel käsitletav sümbol, vaid on vaja ka meelde jätta eelnevad, et teada, mida vastava sisendi puhul teha. Seda läheb vaja näiteks sisendi dekodeerimisel. Sisendist ’<’ väljundisse ’<’ saamiseks on vaja kirjutada selline kood, mis on suuteline

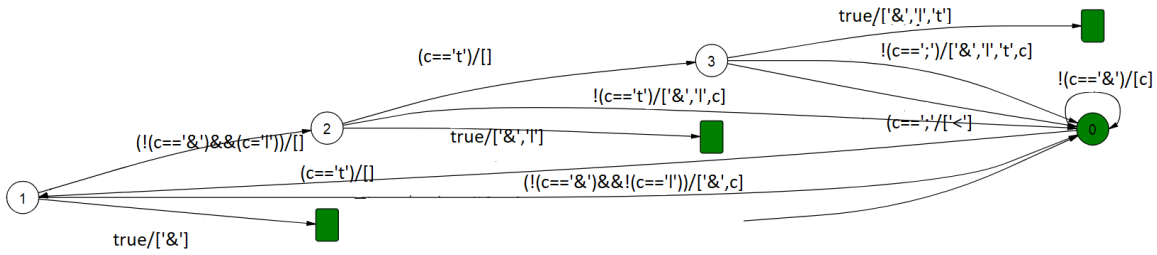
meelde jätma vähemalt kolm eelnevat sümbolit. Selle jaoks on Bekis võimalik kasutada registreid, mis vastavad lõpliku muunduri erinevatele olekutele. Järgnev näide demonstreerib, kuidas on võimalik Bekis liikuda olekute vahel, jättes samal ajal eelnevaid sümboleid meelde.

```
program decode(input) {
    return iter(c in input)[s := 0;]{
    //algolek
    case (s == 0) :
      //Programm jätab meelde & ja liigub olekusse 1
      if (c == '&') { s := 1; }
      else { yield (c); }
    case (s == 1) :
    //Programm jätab meelde &,l ja liigub olekusse 2
      if (c == 'l') { s := 2; }
      else { yield ('&',c); s := 0; }
    case (s == 2) :
    //Programm jätab meelde &,l,t ja liigub olekusse 3
      if (c == 't') { s := 3; }
      else { yield ('&','l',c); s := 0; }
    case (true) :
    //Olek 3, kui järgnev sümbol on ; siis muster lõpetatud, väljund '<'
      if (c == ';')
        { yield ('<'); s := 0; }
      else
        { yield ('&','l','t',c); s := 0; }
    //lõpetamata mustrid
  } end {
    case (s == 0) : yield ();
    case (s == 1) : yield ('&');
    case (s == 2) : yield ('&','l');
    case (true) : yield ('&','l','t');

  };
}
```

Joonisel 3 on visualiseeritud eelnevalt toodud programm „decode“. Programm alustab olekust 0, kus ta kontrollib, kas sisendsümbol on võrdne sümboliga ’&’. Kui ei ole võrdne, siis lisatakse

väljundisse hetkesümbol *c* ja tehakse järgmise sümboliga uus kontroll, kui on võrdne, siis jäetakse sümbol '&' meelde, liigutakse olekusse 1. Olekus 1 toimub kolm kontrolli. Kui järgmine sümbol on 'l', siis liigutakse olekusse 2. Kui tegu on mingi muu sümboliga peale '&', siis lisatakse väljundisse meeldejäetud sümbol '&' ja hetkel käsitletav sümbol *c*. Juhul kui sisend lõppes olekus 1, siis lisatakse väljundisse meeldejäetud sümbol '&'. Olekutes 2 ja 3 toimuvad kontrollid analoogselt.



Joonis 3: Bekis genereeritud muundur programmist „decode"

Rohelisega täidetud kastid on lõpetamata mustrid ehk koodis olevad lõppjuhud (*end cases*). Seega, kui on jõutud sisendi lõppu, aga muster jäi lõpetamata, siis lisatakse pooleliolev muster ehk meeldejäetud sümbolid väljundisse.

2.1.1 Regulaaravaldiste kasutamine Bekis

Bekis on võimalik tingimusavaldistes sümboleid kontrollida ka regulaaravaldistega. Selleks tuleks

```
case (c == sümbol):
```

asemel kirjutada

```
case (c in regex):,
```

kus `regex` on regulaaravaldis, mis on jutumärkide vahel. Juhul, kui regulaaravaldis sisaldab mitut tingimust, siis peaks panema talle ka nurksulud ümber. Seega kontrollimiseks, kas sõne on tühimik (*whitespace*), võib kirjutada

```
case(c in "\\S"):.
```

Kui aga soovitakse kontrollida, kas sümbol on täht a-st z-ni või A-st Z-ni, siis on võimalik kirjutada

```
case(c in "[a-zA-Z]"):.
```

Regulaaravaldised Bekis kasutavad .Net stiili. Täpsema kirjelduse leiab .Net avaldisi tutvustaval leheküljel [8].

2.2 Puhastusfunktsioonid Bekiga

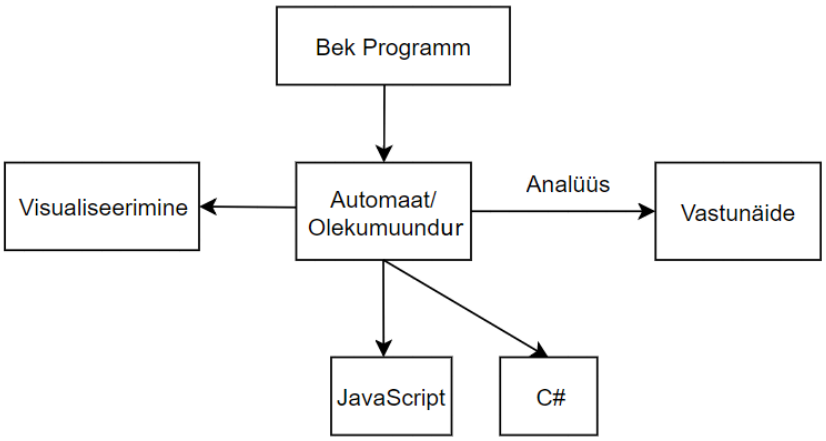
Tuginedes varasemale Saxena jt poolt koostatud artiklile, [9] saab väita, et arendajad kirjutavad tihti puhastusfunktsioone, mis ei taga veebilehe turvalisust. Ebaturvalisus väljendub näiteks selles, kui veebileht kasutab mitut puhastit, aga väljund sõltub nende rakendamise järjekorrast. Teisisõnu – puhastusfunktsioonid ei ole kommutatiivsed. Veel väljendub ebaturvalisus selles, kui puhastusfunktsiooni mitmekordne rakendamine annab erinevaid tulemusi. Näiteks kui sümbol '&' kodeeritakse kui '&', siis sama puhasti uuesti kasutamine selle väljundiga annaks tulemuseks '&', mis ei ole korrektne. Sellised puhastid ei ole idempotentsed ning see võib tähendada rakenduse jaoks turvariske.

Bekiga on võimalik kontrollida puhastusfunktsioonide idempotentsust, kommutatiivsust ning ka üldisemalt kahe puhasti vahelist ekvivalentsust. Juhul, kui koostatud funktsioon ei ole idempotentne või kommutatiivne, siis toob Bek vastunäite, mis selle rikkumise esile toob.

2.3 Üleminekud Bekist teistele keeltele

Bekis kirjutatud puhastusfunktsioone on võimalik teisendada nii JavaScripti kui ka C Sharp keelde ja seega on neid lihtne implementeerida olemasolevasse rakendusse. Kuna Beki programmi konstruktsioon on üles ehitatud juhtudele, mida mingi sümboli korral teha, siis võimaldab Bek genereerida koodile vastava lõpliku muunduri. Üleminek teistele keeltele toimubki muunduri põhjal ning tulemuseks on suur hulk kui-siis-muidu (*if-then-else*) avaldisi.

Beki genereeritud lõplikke muundureid on võimalik ka päringuga *display* vaadata. Päringu kasutamine ei ole vajalik enne teisele keelele üleminekut, küll aga annab ettekujutuse, kuidas Bek selle ülemineku teostab.



Joonis 4: Beki programmi poolt võimalikud läbitavad etapid

Joonisel 4 on toodud etapid, mida programm läbib. Alustatakse programmikoodi kirjutamisest Bekis, seejärel genereerib Bek selle koodi sisemisele ehitusele vastava lõpliku muunduri. See annab võimaluse visualiseerida automaati päringuga *display*. Saab ka analüüsida kirjutatud muundurit ehk kontrollida programmi idempotentsust ja kommutatiivsust ning ekvivalentsust mõne teise programmiga. Kui analüüsi käigus selgub, et mõni neist omadustest on rikutud, siis näitab Bek ära, millise sisendiga selline vastuolu tekib. Ühtlasi on muunduri põhjal võimalik kirjutatud programm teisendada JavaScripti või C Sharp keelde. [10]

3 Bekiga sarnased keeled

Käesolevas peatükis tutvustatakse veel kahte programmeerimiskeelt, mille semantika põhineb sümboolsetel lõplikel automaatidel ja muunduritel – Bex ja Fast. Bex on loodud sõnekodeerijate õigsuse kontrollimiseks ning Fast puu transformatsioonideks ja nende analüüsiks.

3.1 Bex

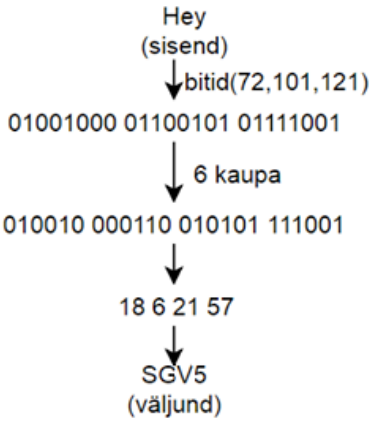
Järgnev peatükk põhineb programmeerimiskeelt Bex tutvustaval leheküljel [11] ja D'Antoni doktoritööl [12]. Bex on sarnaselt Bekiga programmeerimiskeel, mis on mõeldud sõne kodeerijate analüüsiks. Bexis on võimalik vähese ridade arvuga (< 50 rida) kirjutada keerulisi sõne kodeerijaid. Bex toetab regulaaravaldistel põhinevat mustrisobitust, bitivektorite operatsioone nagu *bit-shift, bit-or, bit-extract* ja lõplike muundurite põhiseid operatsioone nagu ekvivalentsuse kontroll. Bexi abil saab kontrollida UTF (*Universal Transformation Format*) ja Base64 kodeerijate ja dekodeerijate korrektsust.

Base64 kodeeringut kasutatakse binaarsete andmete kodeerimiseks, et edastada neid tekstilisel kujul. Kodeerimise eesmärk on jätta tekst transportimise käigus muutmata. Base64-s vaadatakse sisendbitijada 6 biti haaval, seega kokku on $2^6 = 64$ erinevat sümbolit.

Base64 kodeerija loeb ASCII (*American Standard Code for Information Interchange*) sümboleid (bitte) vahemikust 0-255, kus iga sümbol on 8 bitti (seega $2^8 = 256$ erinevat sümbolit). Seejärel jagab kodeerija sisendist saadud bitijada 6-bitisteks osadeks ja seab iga saadud 6-kohalise bitijada vastavusse Base64 sümboliga. Väljundi lõppu lisatakse '==', kui üle jääb 2 bitti ja '=', kui üle jääb 4 bitti (6 kaupa 8-bitilistest võttes võib üle jääda 2 või 4 bitti). Base64 sümbolite hulka kuuluvad nii suur- kui ka väiketähed a-st z-ni, numbrid 0-9 ning sümbolid '+' ja '/'.

Joonisel 5 on näha, kuidas kodeeritakse sõna „Hey“. Kõigepealt seatakse sisendi sümbolid vastavusse ASCII arvudega, mis antud sisendi korral on vastavalt 72, 101 ja 121. Saadud arve vaadatakse bitikujul ning seatakse 6 biti kaupa vastavusse õige Base64 sümboliga. Praeguse sisendi korral on saadud Base64 väärtused 17, 6, 21 ja 57 ning neile vastavad sümbolid on 'S', 'G', 'V' ja '5'. Tabelis 1 on toodud Base64 väärtustele vastavad sümbolid.

Järgnevalt on toodud näide, kuidas kirjutada Bexis Base64 kodeerijat.



Joonis 5: Sisendi „Hey“ kodeerimine Base64 kodeeringusse

```
program base64encode(_){
 replace {
  //3 sümbolit korraga
  @"[\0-\xFF]{3}"  ==> [Enc(#0>>2), Enc((Bits(1,0,#0)<<4)|Bits(7,4,#1)),
    Enc((Bits(3,0,#1)<<2)|Bits(7,6,#2)), Enc(Bits(5,0,#2))];
  //2 sümbolit lõpust
  @"[\0-\xFF]{2}$" ==> [Enc(Bits(7,2,#0)),
    Enc((Bits(1,0,#0)<<4)|Bits(7,4,#1)), Enc(Bits(3,0,#1)<<2), '='];
  //üks sümbol
  @"[\0-\xFF]$"    ==> [Enc(Bits(7,2,#0)), Enc(Bits(1,0,#0)<<4), '=', '='];
 }
}
```

Programm „base64encode“ loeb korraga sisse kolm 8-bitist sümbolit ning teisendab selle neljaks 6-bitiseks sümboliks. Sümboliga `@` algavad read tähistavad erinevaid võimalike juhte. Esimene juht `@"[\0-\xFF]{3}"` on peajuht, kus `"{3}"` näitab, et loetakse korraga 3 sümbolit, tähistusega `"[\0-\xFF]"` aktsepteeritakse ASCII sümboleid vahemikust 0-255. Järgmised juhud `"[\0-\xFF]$"` ja `"[\0-\xFF]{2}$"` on selleks, kui sisendi lõppu jääb vastavalt üks või kaks sümbolit (sümbol `"$"` tähistab sisendi lõppu).

Juhtudes kasutatud sisseehitatud funktsioon `Bits(i,j,s)` eraldab bitivektorist `s` bitivektori, mis algab indeksist `i` ja lõppeb indeksis `j` (lugemine toimub paremalt vasakule). Näiteks, kui tahta eraldada jadast 01110111 viimased neli bitti 0111, siis saab seda teha järgnevalt: `Bits`

```
(3,0,01110111).
```

Märgistus `#0>>2` (ekvivalentne `Bits(7,2,#0)` tulemusega) tähendab seda, et esimeseks 6-bitiseks jadaks võetakse esimesena loetud sümbolist kõik peale viimase kahe biti. Teiseks 6-bitiseks jadaks võetakse esimese sümboli viimased 2 bitti ja teisena loetud sümbolist esimesed neli bitti. Kolmandaks jadaks võetakse teisena loetud sümbolist viimased 4 bitti ja kolmandana loetud sümbolist esimesed 2 bitti. Viimaseks jadaks jääb allesjäänud 6 bitti kolmandana loetud sümbolist. Iga saadud 6-bitine sümbol teisendatakse funktsiooni `Enc` abil Base64 sümboliks. Kui sisendi lõppu jääb 2 sümbolit, siis lisatakse lõppu üks '=' sümbol, kui aga jääb üks sümbol, siis lisatakse kaks '=' sümbolit.

```
function Enc(x)=(ite(x<=25,x+65,ite(x<=51,x+71,ite(x<=61,x-4,ite(x==62,'+',
    '/')))));
```

Programmis kasutatud funktsioon `Enc` kodeerib saadud 6-bitise jada Base64 sümboliks. Funktsioonis `Enc` kasutatud sisseehitatud funktsioon `ite(b,s1,s2)` tagastab `b` tõesuse korral `s1` ning vastasel juhul `s2`. Funktsioon `Enc` kontrollib `ite` abil nelja juhtu. Esimesena vaatab, kas saadud väärtus `x` on mittesuurem kui 25. Kui on, siis lisab talle 65 ja saab tulemuseks ASCII tabelis vastava sümboli. Näiteks Base64-s on $A \rightarrow 0$ ja ASCII-s on $A \rightarrow 65$. Sellest vahemikust saadakse kätte suurtähed. Kui väärtus on suurem kui 25, siis kontrollib, kas väärtus `x` on mittesuurem kui 51. Kui on, siis lisab saadud väärtusele 71 ja saab tulemuseks jälle arvule vastava ASCII sümboli. Sellest vahemikust saadakse kätte väiketähed. Ülejäänud kontrollid on analoogsed, nende kontrollidega saadakse kätte numbrid 0-9 ning sümbolid '+' ja '/'.

Tabel 1: Base64 väärtused ja kodeeringud [13]

Väärtus	Kodeering	Väärtus	Kodeering	Väärtus	Kodeering	Väärtus	Kodeering
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

3.2 Fast

Järgnev peatükk põhineb programmeerimiskeelt Fast tutvustavatel lehekülgedel [14, 15] ja D'Antoni jt artiklil [16], kui pole mainitud teisiti. Fast (*Functional abstraction of symbolic transducers*) on programmeerimiskeel, mis on mõeldud puudel põhinevate programmide analüüsimiseks. Fasti abil on võimalik kirjutada HTML-i puhastusfunktsioone ja neid analüüsida, seda saab kasutada näiteks turvaaukude avastamiseks HTML-i puhastusfunktsioonides. Samuti saab Fasti kasutada järjenditele ja puudele ehitatud funktsionaalsete programmide analüüsimiseks ja optimiseerimiseks.

Fastis kirjutatud puhastusfunktsioonid on lihtsamini mõistetavad kui PHP-s kirjutatud puhastusfunktsioonid, kuna Fastis kirjutatud puhastid (∼200 rida koodi) on oluliselt lühemad, kui PHP-s kirjutatud puhastid (∼10000 rida koodi). Järgnevalt tehakse läbi mõned Fasti tutvustavad näited.

Võtmesõna `Fun` abil saab defineerida funktsioone. Funktsioon `duubelda` võtab argumendiks reaalarvu `x` ja tagastab tulemuseks `2*x`.

```
Fun duubelda (x:real) := (* 2.0 x)
```

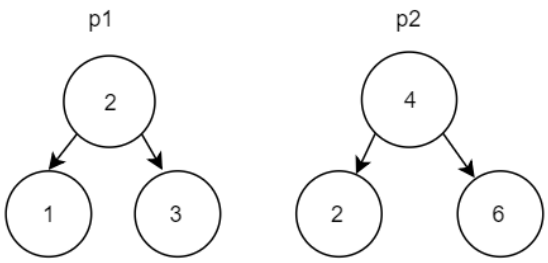
Võtmesõnaga `Alphabet` saab defineerida puu tüübi. Funktsiooniga `BTReal` on defineeritud kahendpuu, mille tipud `r` on reaalarvud ja luuakse, kasutades konstruktorit `B` või `L`. Konstruktoriga `L` loodud tipud on lehed ehk ühtegi alluvat neil ei ole. Konstruktoriga `B` loodud tippudel on 2 alluvat.

```
Alphabet BTReal[r:real]{L(0),B(2)}
```

Võtmesõnaga `Tree` saab luua juba varasemalt defineeritud tüübiga puu.

```
Tree p1 : BTReal := (B [2.0] (L [1.0]) (L [3.0]))
```

Loodud puu on kolme tipuga kahendpuu. Juurtipp on väärtusega 2, tema vasaku alluva väärtus on 1 ja parema alluva väärtus 3. Tulemuseks saadud puu `p1` on joonisel 6 vasakul.



Joonis 6: Funktsiooni `Tree` abil loodud kahendpuu p1 (vasakul) ja sellele funktsiooni `map_duubelda` rakendamisel saadud kahendpuu p2 (paremal)

Võtmesõna `Lang` abil on võimalik seada piiranguid defineeritud puule. Näiteks, kui tahetakse, et tippudes saaksid olla ainult reaalarvud, mis on suuremad kui 1, siis saab seda teha järgnevalt:

```
Lang l1 : BTReal {
  L() where (> r 1.0)
  | B(x,y) where (> r 1.0) given (l1 x) (l1 y)
}
```

Funktsioon `l1` vaatab kahte juhtu. Kui tegemist on lehega, siis kontrollib tipu väärtust, sügavamale ei lähe. Kui tegemist on mingi vahepealse tipuga, siis kontrollib tipu väärtust ja kontrollib rekursiivselt vasaku ja parema haru. Näiteks puu `p1` korral tagastab kontroll `Print (contains l1 p1)`, et `p1` ei sisaldu keeles `l1`.

Puu transformatiooni funktsiooni saab defineerida võtmesõnaga `Trans`.

```
Trans map_duubelda : BTReal -> BTreal{
  B(x,y) to (B [(duubelda r)] (map_duubelda x) (map_duubelda y)) | L() to (
     L [(duubelda r])
}
```

Puule saab rakendada defineeritud transformatsiooni võtmesõnaga `apply`.

```
Tree p2 : BTReal := (apply map_duubelda p1)
```

Puu `p2` on saadud funktsiooni `map_duubelda` rakendamisel `p1`-le. Tulemuseks on samasugune kahendpuu, mille tippude väärtused on kaks korda suuremad. Saadud puu on joonisel 6 paremal.

4 Loodud ülesanded

Bakalaureusetöö raames valmis ka kaks ülesannet programmeerimiskeeles Bek. Loodud ülesannete eesmärgiks on tutvustada Beki võimalusi ja näidata, milliseid muundurite omadusi on võimalik Bekis kasutada, et kontrollida kirjutatud puhastusfunktsioonide turvalisust. Ülesanded on mõeldud lahendamiseks lisaülesandena Tartu Ülikooli arvutiteaduse instituudi aines „Automaadid, keeled ja translaatorid“. Ülesannete aluskoodid pärinevad leheküljelt [7] ning lahendamine toimub veebilehel https://rise4fun.com/Bek.

4.1 Ülesanne 1

Esimese ülesandena tuleb tudengil parandada peatükis 2 toodud programm. Sealne programm dekodeerib sisendis olevad järjestikused sümbolid ’&’, ’l’, ’t’,’;’ sümboliks ’<’. Olemasolev lahendus töötab küll sisendi ’<’ puhul, aga ei tööta juhtudel, kus muster ’<’ rikutakse sümboliga ’&’. Näiteks sisendi ’&l<’ korral peaks väljundiks tulema ’&l<’, aga tuleb ’&l<’. Vihjeks on antud tudengile mõelda, millisesse olekusse saadetakse programm, kui olekutes 1, 2 või 3 on sisendsümboliks ’&’.

Järgnevalt on toodud ülesanne 1 tudengile esitatavas vormis:

> On antud järgmine programm:

```
program decode(input) {
    return iter(c in input)[s := 0;]{
    case (s == 0) :
    //Programm jätab meelde & ja liigub olekusse 1
      if (c == '&') { s := 1; }
      else { yield (c); }
    case (s == 1) :
    //Programm jätab meelde &,l ja liigub olekusse 2
      if (c == 'l') { s := 2; }
      else { yield ('&',c); s := 0; }
    case (s == 2) :
    //Programm jätab meelde &,l,t ja liigub olekusse 3
      if (c == 't') { s := 3; }
      else { yield ('&','l',c); s := 0; }
```

```
    case (true) :
    //Olek 3, kui järgnev sümbol on ; siis muster lõpetatud, vä
        ljund '<'
      if (c == ';') { yield ('<'); s := 0; }
      else { yield ('&','l','t',c); s := 0; }
    //lõpetamata mustrid
  } end {
    case (s == 0) : yield ();
    case (s == 1) : yield ('&');
    case (s == 2) : yield ('&','l');
    case (true) : yield ('&','l','t');
  };
}
```

Programm töötleb sisendit selliselt, et kui sisendis tulevad järjest sümbolid '&', 'l', 't',';', siis need neli sümbolit teisendatakse sümboliks '<', muul juhul lisatakse väljundisse töödeldud sümbol. Näiteks sisendi 'hobune' puhul on väljundiks 'hobune', aga sisendi 'hobune<' puhul saab väljundiks 'hobune<'. See programm kahjuks alati ei tööta. Näiteks sisendi '&l<' puhul peaks olema väljund '&l<', aga on '&l<'.

Täiusta programmi selliselt, et ta töötaks kõikide sisendite puhul. Beki koodi on võimalik jooksutada leheküljel https://rise4fun.com/Bek. Programmi tulemuse kindla sisendiga vaatamiseks on kaks võimalust. Esimene võimalus on jooksutada päring `image (decode,sisend)`, mis väärtustab programmi antud sisendiga. Teine võimalus on jooksutada päring `js(programm)` ja katsetada sisendeid tekkinud aknas.

Vihje: mõtle, millisesse olekusse saadab programm praegu sisendi, kui olekutes 1, 2 või 3 on sisendsümboliks '&'.

Esimese ülesande eesmärk on tutvustada Beki süntaksit ning anda ülevaade tüüpilise programmi ülesehitusest. Samuti aitab ülesande lahendamine mõista automaadipõhist programmeerimist ja näitab, kuidas Bekis toimub olekutevaheline liikumine. Veel tutvustab see, milliseid ülesandeid saab muunduritega lahendada. Ülesannet on võimalik lahendada lisades olekutesse 1, 2 ja 3 ühe lisakontrolli, mis suunab programmi sisendi '&' puhul oleku 0 asemel olekusse 1.

4.2 Ülesanne 2

Teise ülesandena tuleb tudengil parandada programm, mis kodeerib sümbolid ’<’,’>’ ja ’&’ neile vastavatesse HTML kodeeringutesse. Toodud programm saab hakkama sümbolite kodeerimisega, aga ei ole idempotentne ning see võib rakenduse jaoks tähendada turvariske.

Tudengi ülesandeks muuta programmi, et ta oleks idempotentne.

Järgnevalt on toodud ülesanne 2 tudengile esitatavas vormis.

> On antud järgmine programm:
>
> ```
> program encode(input) {
>   return iter(c in input){
>     case (c == '<') : yield ('&','l','t',';');
>     case (c == '>') : yield ('&','g','t',';');
>     case (c == '&') : yield ('&','a','m','p',';');
>     case (true)     : yield(c); };
> }
> ```
>
> Programm kodeerib sisendis sümbolid ’<’, ’>’ ja ’&’ neile vastavate HTML kodeeringutega. Paraku on ka see programm vigane. Nimelt see programm ei ole idempotentne. See tähendab, et kui programmi käivitada kaks korda järjest nii, et teisel käivitamisel on sisendiks esimesest käivitamisest saadud tulemus, siis need tulemused peavad olema samad. Näiteks sisendiga ’&’ käivitamisel saame tulemuseks ’&’ ja kui sellega uuesti käivitada, saame tulemuseks ’&’.
>
> Paranda programm selliselt, et see oleks idempotentne. Idempotentsust saab kontrollida päringuga `eq(decode,join(decode,decode);`. Kui programm ei ole idempotentne, siis toob Bek ka näited, millise sisendiga seda tingimust rikutakse. See võiks anda ideid, mis viisil programmi parandama peaks.

Ülesande eesmärk on demonstreerida Beki võimekust tuvastada potentsiaalseid turvariske puhastusfunktsioonides ja näidata, kuidas on võimalik kasutada Bekis ekvivalentsuse kontrolli kirjutatud programmi paremaks tegemisel. Ülesande lahendamiseks on vaja kasutada olekuid (registreid) sümbolite meelde jätmiseks. Olekute abil saab kontrollida sümboleid, mis idempotentsust rikuvad. Toodud ülesandes on selleks sümboliks ’&’, seega on vaja kontrollida, mis

järgnevad sellele sümbolile. Kui on järjest sümbolid ’&’, ’l’, ’t’,’;’, ’&’, ’g’, ’t’,’;’ või ’&’, ’a’, ’m’,’p’,’;’, siis ei tohi programm sümbolit ’&’ kodeerida.

5 Kokkuvõte

Bakalaureusetöö eesmärk oli tutvustada sümboolsetele automaatidele ja olekumuunduritele ehitatud programmeerimiskeeli. Töös on antud ülevaade kolmest keelest: Bek, Bex ja Fast.

Töö keskendub peamiselt programmeerimiskeele Bek tutvustamisele. Töös on kirjeldatud, milline näeb välja tüüpiline Bek programmi ülesehitus ning kuidas nende programmide kirjutamisele läheneda. Samuti on töös välja toodud, milliseid eelisi on sõnetöötlusprogrammide kirjutamisel Bekis teiste programmeerimiskeelte ees.

Bakalaureusetöös on kirjeldatud ka kahte Bekile sarnast programmeerimiskeelt: Bex ja Fast. Välja on toodud, mille jaoks on programmeerimiskeeled Bex ja Fast loodud ning läbi on tehtud mõningad näited, kuidas on sealsed programmid ülesehitatud. Bexis on ka läbi tehtud ja põhjalikult seletatud näide, kuidas kirjutada Base64 kodeerijat.

Töö raames valmisid ka lisaülesanded, mida saab kasutada aine „Automaadid, keeled ja translaatorid“ raames automaatide ja muundurite tutvustamiseks.

Viited

[1] Cross-site Scripting (XSS). OWASP.
https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (09.03.2019)

[2] Veanes, M. Applications of Symbolic Finite Automata. 2013. a.
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ciaa13.pdf
(09.03.2019)

[3] Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.
Fast and precise sanitizer analysis with Bek
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/paperUSENIXSEC11.pdf

[4] Vojdani, Vesal. Automaadid elus.
https://courses.cs.ut.ee/LTAT.03.006/2018_spring/uploads/Main/l3-vesal.pdf, 2018.

[5] Deterministic Finite Automata.
http://www.cse.chalmers.se/ coquand/AUTOMATA/o2.pdf (09.03.2019)

[6] Bek - guide
https://rise4fun.com/Bek/tutorial/guide (09.03.2019)

[7] Bek - guide2
https://rise4fun.com/Bek/tutorial/guide2 (09.03.2019)

[8] Regular Expression Language - Quick Reference
https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference (10.04.2019)

[9] Saxena, P., Molnar, D., Livshits, B.
SCRIPTGARD: Preventing Script Injection Attacks in Legacy Web Applications with Automatic Sanitization
https://pdfs.semanticscholar.org/4377/0f1cbc9c47d3c09f752df93bc963239d9b37.pdf?_ga=-2.214846935.1702352369.1553278097-577135974.1553278097 (22.03.2019)

[10] Molnar, D., Veanes, M.
Bek - domain specific language for string manipulation functions
https://channel9.msdn.com/Blogs/Peli/David-Molnar-and-Margus-Veanus-BEK-Domain-Specific-Language-for-String-Manipulation-Functions (09.04.2019)

[11] Bex - guide
https://rise4fun.com/Bex/tutorial/guide (18.04.2019)

[12] Loris D'Antoni Programming using automata and transducers
https://www.cis.upenn.edu/ alur/loris-thesis.pdf (17.04.2019)

[13] Base64 Data Encodings
https://tools.ietf.org/html/rfc3548 (18.04.2019)

[14] Fast - examples https://rise4fun.com/fast (03.05.2019)

[15] Fast - guide https://rise4fun.com/Fast/tutorial/guide (03.05.2019)

[16] D'Antoni, L., Veanes, M., Livshits, B., Molnar, D.
Fast: A Transducer Based Language For Tree Manipulation
http://pages.cs.wisc.edu/ loris/papers/pldi14.pdf (03.05.2019)

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, *Jaagup Russak*,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
   „*Programmeerimine, kasutades sümboolseid automaate ja muundureid*“,

   mille juhendaja on *Vesal Vojdani*,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, alates **08.05.2019** kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Jaagup Russak

08.05.2019