T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Tarkvarasüsteemide õppetool
Informaatika eriala

Kristo Heero

Parserigeneraator JavaCC ja skeem-modelleerimine

Magistritöö

Juhendaja: prof. Jüri Kiho

Autor: . . . . . “. . . .” mai 2002
Juhendaja: . . . . . ”. . . .” mai 2002
Õppetooli juhataja: . . . . . .”. . . .” mai 2002

Tartu 2002

Sisukord

1 Sissejuhatus

1.1 Proloog

Tänapäeval puututakse tihti kokku erinevate arvutitekstidega, mida soovitakse mingil viisil töödelda, kas siis redigeerida, lugeda vms. Tekstid võivad olla aga väga erinevalt süntaktiliselt kirjeldatud ning nende spetsiifiline töötlemine nõuaks ka vastavaid erilisi töötlusvahendeid ja lähenemisi. Eritoimetite “algusest peale” loomine või uute süsteemide integreerimine on enamasti kulukas. Üheks alternatiiviks on skeem-tehnoloogia ja sellel baseeruv skeem-toimeti Amadeus-fRED (*Amadeus for Rapid Editor Development*). Viimasele saab teatud viisil juurde integreerida erinevaid arvutitekstide baaskeeli, mille edasine töötlemine järgib siis ühtset kontseptsiooni. Samas ei tohiks olla uue baaskeele lisamine oluliselt keerulisem ja rohkem ressurssi nõudvam, kui mingi muu toimeti loomine või kohaldamine eriotstarbeks. Järelikult vajatakse selleks efektiivset integreerimisprotsessi.

1.2 Käesolev väitekiri

Käesoleva töö eesmärgiks on uurida võimalusi, kuidas skeemtoimetile Amadeus-fRED lihtsamalt ja automatiseeritumalt uusi baaskeeli lisada. Sobivaimaks abivahendiks tuleb pidada parserigeneraatorit JavaCC, mis on suhteliselt populaarne ja töökindel. Pealegi on see Java-põhine nagu vastav skeemtoimetigi. Töö käigus on välja töötatud esmane metoodiline lähenemine uute baaskeelte integreerimise ülesandele. Perspektiiviks on selle protsessi täielikum automatiseerimine.

Teises peatükis tutvustatakse parserigeneraatorit JavaCC. Sissejuhatuseks koostatakse pisike looksulgude keele grammatikafail ning tutvustatakse lakooniliselt sealjuures tehtut. Edasistes jaotistes kirjeldatakse parseri grammatikafaili loomist juba detailsemalt. Tutvustatakse parseri tööd ja genereerimist mõjutavaid parameetreid ning kirjeldatakse erinevaid grammatika produktsioone. Muuhulgas seletatakse parsimisprotsessi töökäiku, eelkõige parsimistee valikalgoritme, mis sõltuvad eelvaadatavate lekseemide arvust, ning lekseemikoostamise protsessi. Samuti

tutvustatakse, kuidas on võimalik sekkuda lekseemihalduri töösse. Vaatluse all on ka veatöötluse võimalused. Peatüki lõpus antakse ülevaade JavaCC käsurea süntaksist ning tutvustatakse dokumenteerimisvahendit JJDoc. Antud peatükk on enamus osas referatiivne ning pärineb erinevatest JavaCC õppematerjalidest. Oma detailsuse ja ohtrate näidete tõttu võiks see olla abimaterjaliks algajale JavaCC kasutajale.

Kolmas peatükk kirjeldab, kuidas parserigeneraatorit JavaCC saab kasutada skeem-modelleerimisel. Tutvustatakse põgusalt skeemtöötluse ideed, skeemteksti üldist süntaksi ning selleks otstarbeks loodud vastavat skeemtoimetit Amadeus-fRED. Suurema näitena on defineeritud reaalne arvutiteksti keel EKFG ning koostatud sellele skeem-mudel. Seletatakse arvutiteksti skeemistamiseks tehtavaid vajalikke tegevusi ning konstrueeritakse näidetena kaks skeemistusparserit. Peatüki lõpus tuuakse välja ideid ja lahendusi skeemistusparseri automaatse(ma)ks loomiseks.

Lisas on toodud keele EKFG skeemistusparser skeemkujul koos antud keele pesastatud skeem-mudeliga.

Lugejalt eeldatakse elementaarseid teadmisi formaalsetest keeltest ning skeemkeelest. Väitekirjas kasutatud programmeerimiskeele Java terminid pärinevad leksikonist [VJL]. Skeemtekstid on koostatud Amadeus-fRED abil.

Käesolev magistritöö elektronkujul asub veebis http://www.math.ut.ee/~kristo/mag.

2 JavaCC

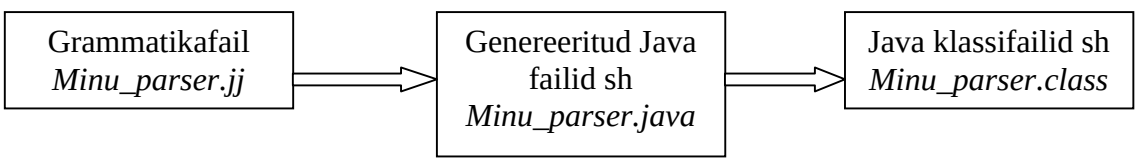
Selles peatükis käsitletakse parserigeneraatorit JavaCC. Antakse ülevaade JavaCC võimalustest ning seletatakse üksikasjalikult koos näidetega, kuidas koostada arvutikeele grammatikat, mille baasil genereeritakse vastav parser.

2.1 Sissejuhatus JavaCC valdkonda

JavaCC (*Java Compiler Compiler*) on arenev Java parserite generaator [JavaCC]. Algselt oli see Metamata toodang, kuid hiljem osteti ära WebGain'i ja Sun Microsystem'i poolt. JavaCC on puhas Java rakendus, mis töötab Java virtuaalmasinal (JVM) alates versioonist 1.1. Lisaks parseri genereerimisele võimaldab JavaCC ka puude ehitamist (lisatud vahendi JJTree abil, mida edaspidi ei käsitleta), koodi silumist jms.

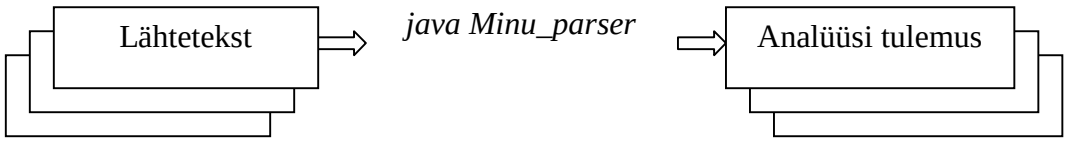
JavaCC poolt genereeritud parseri töö võib jagada kolme ossa: leksiline analüüs, süntaktiline analüüs ning koodi genereerimine või täitmine. Leksilise analüüsi käigus püütakse sisendvoogu jagada lekseemideks, kus lekseem on arvutitekstis tähendust omav elementaarüksus. Nagu näiteks võtmesõnad, kirjavahemärgid, literaalid (numbrid, sõned, ...) jms. Lekseemide eraldajana kasutatakse enamasti tühikut. Süntaktilise analüüsi käigus kontrollitakse arvutiteksti süntaktilist korrektsust vastavatele reeglitele ning konstrueeritakse vastav parsimispuu.

Üldiselt loeb JavaCC grammatika kõrge taseme kirjeldusi ning konverteerib need Java programmiks, mis suudab töödelda grammatikale vastavaid arvutitekste. Järgnevalt on illustreeritud parseri genereerimist:



javacc Minu_parser.jj *javac *.java*

Järgnevalt on illustreeritud parseri rakendamist lähtetekstile, mille tulemusel saadakse vastav analüüs:



2.2 JavaCC grammatika lühitutvustus näidete varal

Selles jaotises käsitletakse mõningaid lihtsaid JavaCC grammatikafaile, et tekiks esmane ettekujutus grammatikafaili struktuurist ja arvutitekstide analüüsimise võimalustest. See hõlbustab järgmistes jaotistes detailsemalt käsitletud grammatikafailide ehitusest arusaamist.

Näitena vaadeldakse siin keelt, milles tekstideks on “{}”, “{{}}”, “{{{}}}”, ... ehk hulk vasakpoolseid looksulge, millele järgneb sama palju parempoolseid looksulge. Vastav grammatika on esitatav ühe produktsioonina, nt:

MatchedBraces ::= “{“ [MatchedBraces] “}”

Selle grammatika jaoks on näiteks legaalsed tekstid: “{}”, “{{{{}}}}” ja mittelegaalsed: “{}}”, “{“ , “{}{}”.

Kui neid tekste lugeda failidest, siis tuleb arvestada veel failiterminaatoriga (<EOF>) ja reaterminaatoritega (“\n”, “\r”):

Input ::= MatchedBraces (“\n” | “\r”)* <EOF>
MatchedBraces ::= “{“ [MatchedBraces] “}”

See sätestab, et teksti lõpus võivad olla reaterminaatorid ja peab olema failiterminaator. Ülaltoodud kaheproduktsioonilisele grammatikale vastab järgmine JavaCC grammatikafail:

```
PARSER_BEGIN(Braces)

public class Braces {

  public static void main(String args[]) throws ParseException {
      Braces parser = new Braces(System.in);
      parser.Input();
  }
}

PARSER_END(Braces)



void Input() :
{}
{
      MatchedBraces() ("\n"|"\r")* <EOF>
}

void MatchedBraces() :
{}
{
      "{" [ MatchedBraces() ] "}"
}
```

Võtmesõnade *PARSER_BEGIN(name)* ja *PARSER_END(name)* vahel asub Java kompileerimisüksus, mis võib olla suvalise keerukusega. Ainukeseks kitsenduseks on see, et antud kompileerimisüksus peab defineerima klassi nimega *name*, kus antud nimi peab ühtima *PARSER_BEGIN* ja *PARSER_END* argumendi nimega. Genereeritava parseri kood pannakse klassi *name* lõpetava looksulu ette.

Antud näites koosneb klass, kuhu parser genereeritakse, peameetodist *main*. Selles luuakse parseri isend konstruktoriga, mis võtab *java.io.InputStream* tüüpi argumendi (antud juhul *System.in*). Seejärel kutsub *main* välja parsida soovitava grammatika mitteterminaali (antud grammatika algussümboli *Input*).

Olgu näiteks vaadeldav kaheproduktsiooniline JavaCC grammatika salvestatud faili “Braces.jj”. Siis käsuga `javacc Braces.jj` genereeritakse vastav parseri Java-kood ja `javac *.java` abil see kompileeritakse. Käivitanud tekkinud klassifaili käsuga `java Braces`, saab konsoolilt ette anda analüüsitavaid tekste. Näiteks andes “{}}”, saame vastava veateate, mis annab tunnistust mittelegaalsest tekstist. Kuid sisendi “{{}}” korral ei väljastata üldjuhul midagi, seega antud tekst on legaalne.

Jätkame endiselt eelpool toodud grammatikafaili vaatlemist. Java kompileerimisüksusele järgnevad produktsioonid. Antud näites on kaks produktsiooni, mis defineerivad mitteterminaalid *Input* ja *MatchedBraces*. JavaCC grammatikas on mitteterminaalid kirjutatud kui Java meetodid, seega nende deklaratsioon järgib vastavat Java süntaksit. Kui mitteterminaal esineb aga ise mingis produktsioonis, siis käsitletakse seda Java meetodi väljakutsumisena.

Iga produktsioon defineerib koolonile järgnevalt oma vasakpoolse mitteterminaali. See koosneb Java blokist ja sellele järgnevast looksulgude vahel olevast laienduste (*expansions*) hulgast. Java blokis võib esineda hulk deklaratsioone ja avaldisi (antud näites need puuduvad ning seetõttu on ainult Java bloki kohustuslikud looksulud “{ }”).

Lekseemid (regulaaravaldised) JavaCC grammatikas on kas lihtsad sõned (“{“, “}”, “\n”, “\r”) või siis keerukamad regulaaravaldised (noolsulgudesse võetud). Antud näites on selliseks regulaaravaldiseks failiterminaator <EOF>, mis ühtib (*match*) faili lõpuga.

Esimene produktsioon antud näites määrab, et mitteterminaali *Input* laienduses on mitteterminaal *MatchedBraces*, millele järgneb null või enam reaterminaatorit ja failiterminaator. Teine produktsioon laiendab lekseemi “{“, millele järgneb mittekohustuslik pesastatud (*nested*) laiendus *MatchedBraces* ja lekseem “}”.

Nurksulud [*exp*] näitavad, et *exp* on mittekohustuslik; [*exp*] võib kirjutada ka kui (*exp*)?. Need kaks vormi on ekvivalentsed (va regulaaravaldistes, mida käsitletakse hiljem). Laiendustes esineda võivad struktuurid on järgmised:

[*exp*] ehk (*exp*)?	- null või üks *exp* esinemine.
*exp*1 | *exp*2 | *exp*3 | ...	- valik üks neist, kas *exp*1 või *exp*2 või *exp*3 ...
(*exp*)+	- üks või rohkem *exp* esinemist.
(*exp*)*	- null või rohkem *exp* esinemist.

Antud konstruktsioonid võivad kõik ka üksteise sees esineda, näiteks:

((*exp*1 | *exp*2)* [*exp*3]) | *exp*4

Täiendame eelnevat näidet, võttes kasutusele SKIP tüüpi leksilised olemid, mida nimetatakse tühilekseemideks:

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

void Input() :
{}
{
  MatchedBraces() <EOF>
}

void MatchedBraces() :
{}
{
  "{" [ MatchedBraces() ] "}"
}
```

Nüüd lubatakse keeles looksulgude seas ka tühemikke, näiteks legaalne sõne on ka “{{ }\n}\n\n”. Lisatud on SKIP tüüpi leksiline spetsifikatsioon, mis algab võtmesõnaga SKIP. See määrab nn tühilekseemid (antud juhul tühik, tabulaator, reaterminaatorid), mida parsimise käigus ühtimisel lihtsalt ignoreeritakse (heidetakse kõrvale). Tühilekseemide kasutamine lihtsustab oluliselt grammatika kirjeldamist.

Lisaks tüübile SKIP on olemas veel järgmised leksilise spetsifikatsiooni tüübid:

TOKEN – üldotstarbeline; kasutatakse lekseemide spetsifitseerimiseks.

SPECIAL_TOKEN – kasutatakse nn erilekseemide spetsifitseerimiseks, mida parsimisel ignoreeritakse. Analoogne SKIP-iga, kuid neid lekseeme saab parseri aktsioonides (*action*) käsitleda.

MORE – spetsifitseerib osalise lekseemi. Täielik lekseem koostatakse MORE ja sellele järgnevast TOKEN või SPECIAL_TOKEN lekseemist.

Täiendame eelmist näidet, spetsifitseerides veel mõned lekseemid:

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN :
{
  <LBRACE: "{">
| <RBRACE: "}">
}

void Input() :
{ int count; }
{
  count=MatchedBraces() <EOF>
  { System.out.println("The levels of nesting is " + count); }
}

int MatchedBraces() :
{ int nested_count=0; }
{
  <LBRACE> [ nested_count=MatchedBraces() ] <RBRACE>
  { return ++nested_count; }
}
```

Nüüd on sisse toodud ka TOKEN tüüpi leksilised olemid. Antud juhul on “{“ ja “}” defineeritud lekseemidena ning neile on antud vastavalt nimed ehk märgendid *LBRACE* ja *RBRACE*. Neid nimesid saab nüüd kasutada noolsulgudes, et viidata vastavale lekseemile. Tavaliselt kasutatakse selliseid lekseemide spetsifikatsioone komplekslekseemide (nagu identifikaatorid) või literaalide defineerimiseks.

Antud näide illustreerib ühtlasi ka aktsioonide ehk tegevuste kasutamist grammatika produktsioonis. Ülesandeks on loendada klappivate looksulupaaride arv. Vastavates Java blokkides deklareeritakse muutujad *count* ja *nested_count*. Mitteterminaal *MatchedBraces* tagastab *int* tüüpi väärtuse nagu Java funktsioon oma tagastusväärtuse.

Edasi vaadeldakse veel mõningaid lekseemide spetsifitseerimise näiteid.

```
TOKEN :
{
  < Id: ["a"-"z","A"-"Z","_"](["a"-"z","A"-"Z","_","0"-"9"])* >
}
```

Siin kasutatakse komplekssemat regulaaravaldist, mille nimeks on pandud Id. Sellele lekseemile võib <Id> abil viidata ka mujal grammatikafailis. Nurksulgudes on määratud

hulk lubatud sümboleid: antud juhul ühtib sellega sisend, mis on üks vastavatest väike- või suurtähtedest või alakriips, millele võib järgneda null või enam väike- või suurtähte, numbrit või alakriipsu.

Regulaaravaldistes võib esineda veel järgmisi konstruktsioone:

(*exp*) +	: üks või rohkem *exp* esinemist
(*exp*) ?	: mittekohustuslik *exp* esinemine
(*exp*1 | *exp*2 | ...)	: suvaline üks avaldistest *exp*1, *exp*2, ...

Regulaaravaldises ei saa (…)? asemel kasutada […]. Viimasel on eritähendus, nimelt konstruktsioon […] on muster, millega ühtib sümbol, mis on spetsifitseeritud nurksulgude sees. Nurksulgude sees võivad olla (komaga eraldatud) üksikud sümbolid või ka sümbolite vahemikud. Kui enne nurksulge on “~”, siis mustriga ühtivad kõik sümbolid, mis pole spetsifitseeritud selle sees. Näiteid:

[“a”-“z”]	- ühtivad kõik väiketähed a-st z-ni.
~[]	- ühtivad kõik sümbolid.
~[“\n”,”\r”]	- ühtivad kõik sümbolid, va reaterminaatorid.

Veel üks näide:

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN :
{
  < Id: ["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z","0"-"9"] )* >
}

void Input() :
{}
{
  ( <Id> )+ <EOF>
}
```

Antud juhul on legaalseks sisendiks tekst “abc xyz123 A B C \t \n aaa”, kuna kahe

lekseemi vahel on lubatud suvaline hulk tühilekseeme. Samas sisend “xyz 123” ei ole legaalne, sest tühilekseem pärast “xyz”-i põhjustab ühe lekseemi lõppemise ja teise algamise. Seega “123” peaks olema omaette lekseem, mis aga ei ole lubatud vastavas grammatikas.

Järgmises näites juhitakse tähelepanu aga sellele, et SKIP tüüpi regulaaravaldisi ignoreeritakse lekseemide vahel, kuid mitte lekseemide sees. Selleks piisab, kui kirjutatakse lekseem <Id> ümber järgmiselt:

```
TOKEN :
{
  < Id: ["a"-"z","A"-"Z"] ((" ")* ["a"-"z","A"-"Z","0"-"9"])* >
}
```

Tühikute lubamine lekseemi spetsifikatsioonis ei keela kasutada tühikut SKIP tüüpi leksilise olemina. Antud näite võib kirja panna ka (lekseemi defineerimata) produktsioonina:

```
void Id():
{}
{
<["a"-"z","A"-"Z"]>(["a"-"z","A"-"Z","0"-"9"])*
}
```

Nüüd on antud mõningane ettekujutus sellest, milline üks lihtne grammatikafail välja võiks näha ning järgmises jaotises võib juba asuda JavaCC grammatika detailse kirjelduse vaatlemisele.

2.3 JavaCC grammatikafaili detailne kirjeldus

JavaCC grammatikafail kujutab endast sisuliselt teatud reeglistikku sisaldavat tekstifaili, mille baasil genereeritakse vastav parser. Grammatikafailis defineeritavad lekseemid järgivad Java programmeerimiskeele tavasid, st. muutujad (*identifier*), sõned (*string*), sümbolid (*character*) jms, mida kasutatakse antud grammatika kirjeldamiseks, on samad, mis Java muutujad, sõned ja sümbolid. Ka grammatikas esinevad tühikud, kommentaarid jms järgivad Java konventsioone. Enamus grammatikafailis olevaid kommentaare kantakse üle genereeritavasse parseri koodifaili.

Analoogiliselt Java failidele, läbib ka JavaCC grammatikafail eeltöötluse, kus enne leksilist analüüsi asendatakse *unicode* arvkoodis kirjeldatud sümbolid (sõned kujul \uxxxx, kus xxxx on neljakohaline 16-süsteemi arv) vastavateks sümboliteks.

Lekseemide nimekirjast on erandina välja jäetud operaatorid “<<”, “>>”, “>>>”, “<<=”, “>>=” ja “>>>=” selleks, et lubada mugavat lekseemide pesastamist. Juurde on lisatud ka järgmised uued võtmesõnad: EOF, IGNORE_CASE, JAVACODE, LOOKAHEAD, MORE, options, PARSER_BEGIN, PARSER_END, SKIP, SPECIAL_TOKEN, TOKEN ja TOKEN_MRG_DECLS.

2.3.1 JavaCC grammatikafaili üldine struktuur

Kõik grammatikafaili kirjeldavad konstruktsioonid on edaspidi esitatud XBNF (*eXtended BNF*) kujul. Grammatikafaili üldine struktuur, mille komponente hakatakse edasistes jaotistes detailselt vaatlema, on järgmine:

javacc_input ::=
javacc_options
“PARSER_BEGIN” “(“ <IDENTIFIER> ”)”
java_compilation_unit
“PARSER_END” “(” <IDENTIFIER> “)”
(production)*
<EOF>

Grammatikafail algab mittekohustuslike parameetrite (*options*) loendiga. Sellele järgneb Java kood – kompileerimisüksus, mis asub *PARSER_BEGIN(name)* ja *PARSER_END(name)* vahel. Pärast seda tulevad grammatika produktsioonid. Võtmesõnade *PARSER_BEGIN* ja *PARSER_END* järel sulgudes asub genereeritava parseri nimi. Näiteks, kui antud nimeks on “MyParser”, siis JavaCC genereerib järgmised failid: “**MyParser**.java” – genereeritud parser, “**MyParser**TokenManager.java” – genereeritud lekseemihaldur (leksiline analüsaator), “**MyParser**Constants.java” – konstantide kogum. Lisaks genereeritakse ka “Token.java”, “ParseError.java” jm, kuid nende genereeritud failide kood on iga grammatika jaoks sama. *PARSER_BEGIN* ja *PARSER_END* vahel asuv tekst on Java kood, mis peab sisaldama vähemalt genereeritava parseri nimelist klassikirjeldust. Näiteks:

```
PARSER_BEGIN(MyParser)
...
class MyParser ... {
...
}
...
PARSER_END(MyParser)
```

JavaCC ei kontrolli detailselt Java kompileerimisüksuse koodi, mistõttu võib selle vigane süntaks läbi minna ning vead avastatakse alles Java failide kompileerimisel. Kui kompileerimisüksuses on paketi deklaratsioon, siis see lisatakse kõikidesse genereeritavatesse failidesse. Kui kompileerimisüksus sisaldab aga impordi deklaratsioone, siis need lisatakse genereeritavatesse parseri ja lekseemihalduri failidesse. Genereeritud parseri fail sisaldab kogu kompileerimisüksust ja lisaks vastavate produktsioonide baasil genereeritud parseri koodi, mis lisatakse parseri klassi koodi lõppu:

```
...
class MyParser ... {
      ...
      // genereeritud parseri kood lisatakse siia
      // klassi lõpetava looksulu ette
}
...
```

Genereeritud parser sisaldab vastavalt igale grammatikafailis olevale mitteterminaalile avaliku (*public*) meetodi kirjelduse. Mitteterminaali parsides kutsutakse välja antud mitteterminaalile vastav meetod.

2.3.2 JavaCC mittekohustuslikud parameetrid

Järgnevalt on kirjeldatud mittekohustuslike parameetrite loend, millega saab mõjutada JavaCC parseri tööd ja genereeritava parseri omadusi:

```
javacc_options ::= [ “options” “{” (option_binding)* “}” ]

option_binding ::=
                  “LOOKAHEAD” “=” java_integer_literal “;”
                | “CHOICE_AMBIGUITY_CHECK” “=” java_integer_literal “;”
                | . . .
                | “OUTPUT_DIRECTORY” “=”java_integer_literal “;”
```

JavaCC mittekohustuslikud parameetrid algavad võtmesõnaga *options*, millele järgneb looksulgudes asuv parameetrite ja nende väärtuste loend. Sama parameetrit ei või loendis mitu korda esineda. Parameetreid võib lisada nii grammatikafaili kui ka ette anda käsurealt. Eesõiguse saavad antud juhul käsurealt sisestatud parameetrid. JavaCC-s on järgmised parameetrid:

LOOKAHEAD – määrab parsimisel ettevaadatavate lekseemide arvu, mille põhjal valikpunktis (*choice point*) edasine parsimistee valitakse. Vaikeväärtus on 1. Mida väiksem on see arv, seda kiirem on parser ja vastupidi. Seda väärtust on võimalik hiljem ka grammatikas spetsifitseeritud produktsioonis üle defineerida. Lekseemide eelvaatuse (*lookahead*) detailsem kirjeldus leidub jaotises 2.4.

CHOICE_AMBIGUITY_CHECK – määrab lekseemide arvu, mida “A | B | …” liiki valikpunktides ebaselguste kontrolliks vaadelda tuleks. Vaikeväärtus on 2. Näiteks oletame, et antud parameetri väärtus on muudetud 3-ks ja A-l ja B-l on kaks esimest lekseemi sarnased ja kolmas erinev, siis parseri genereerimisel soovitatakse kasutada eelvaatlust kaugusega 3, et kaotada parsimise valikpunktis ebaselgus. Aga kui ka kolmandad lekseemid on sarnased, siis soovitatakse kasutada endiselt eelvaatlust kaugusega 3 või enam. Selle parameetri suurendamine võib anda rohkem informatsiooni ebaselgete valikpunktide kohta grammatikas, kuid samas pikendab see parseri genereerimise aega.

OTHER_AMBIGUITY_CHECK – määrab, mitu lekseemi muudes valikpunktides

((A)+, (A)* ja (A)?) ebaselguste kaotamiseks ette vaadatakse. Vaikeväärtus on 1. See kontroll võtab rohkem aega, kui CHOICE_AMBIGUITY_CHECK ja sellepärast on ka vaikeväärtus 1, mitte 2.

STATIC – parameetri vaikeväärtus on tõene (*true*). Sel juhul on kõik klassi muutujad ja meetodid genereeritavas parseris ja lekseemihalduris staatilised. Nii lubatakse ainult ühte parseri objekti, mis tõhustab parseri tööd. Selleks, et teha mitmeid parsimisi ühes täimisel olevas Java programmis, tuleb kasutada parseri taas-initsialiseerimiseks meetodit *ReInit*. Kui parameetri väärtus on väär (*false*), siis võib kasutada operaatorit *new* nii mitme parseri loomiseks kui vaja.

DEBUG_PARSER – kasutatakse silumisinformatsiooni saamiseks parserilt. Vaikeväärtus on väär. Kui see parameeter seada tõeseks, siis analüsaator kogub oma töö kohta andmeid ja väljastab need. Meetodite *disable_tracing* ja *enable_tracing* abil saab seda genereeritud parseri klassis lisaks vastavalt keelata ja lubada.

DEBUG_LOOKAHEAD – parameetri vaikeväärtus on väär. Kui see on tõene, siis lisaks DEBUG_PARSER informatsioonile näidatakse ka LOOKAHEAD meetodi täitmiskulgu.

DEBUG_TOKEN_MANAGER – kasutatakse silumisinformatsiooni saamiseks genereeritud lekseemihaldurilt. Vaikeväärtus on väär. Kui see parameeter seada tõeseks, siis lekseemihaldur väljastab oma täitmiskäigu kohta lisainformatsiooni.

OPTIMIZE_TOKEN_MANAGER – parameetri vaikeväärtus on väär. Kui parameeter seada tõeseks, siis viiakse läbi lekseemihalduri optimeerimine.

ERROR_REPORTING – parameetri vaikeväärtus on tõene. Sel juhul on analüüsimise veateated mõnevõrra detailsemad. Ainsaks põhjuseks, miks seda parameetrit vääraks seada, on parseri töökiiruse tõstmine.

JAVA_UNICODE_ESCAPE – parameetri vaikeväärtus on väär. Kui parameetri väärtus on tõene, siis kasutab parser sellist objekti sisendvoo lugemiseks, mis töötleb Java *unicode* arvkoodid enne nende saatmist lekseemihaldurile. Seda parameetrit

ignoreeritakse, kui USER_TOKEN_MANAGER või USER_CHAR_STREAM on tõene.

UNICODE_INPUT – parameetri vaikeväärtus on väär. Kui parameeter on tõene, siis parser kasutab sisendvoogu, mis loeb *unicode* faile. Vaikimisi loetakse sisendist ASCII faile. Seda parameetrit ignoreeritakse, kui USER_TOKEN_MANAGER või USER_CHAR_STREAM on tõene.

IGNORE_CASE – parameetri vaikeväärtus on väär. Kui parameeter on tõene, siis lekseemihaldur ei erista suur- ja väiketähti. Seda on võimalik grammatikas ka ainult lokaalselt määrata.

USER_TOKEN_MANAGER – parameetri vaikeväärtus on väär. Sel juhul genereeritakse lekseemihaldur, mis töötab spetsifitseeritud grammatika lekseemidel. Kui parameetri väärtus on seatud tõeseks, siis genereeritakse parser, mis on võimeline aktsepteerima lekseeme igast lekseemihaldurist, mis on *TokenManager* tüüpi (vastav liides genereeritakse parseriga samasse kataloogi).

USER_CHAR_STREAM – parameetri vaikeväärtus on väär. Sel juhul genereeritakse parameetrite JAVA_UNICODE_ESCAPE ja UNICODE_INPUT põhjal sümbolite voo lugeja (*character stream reader*). Genereeritav lekseemihaldur hakkab saama oma sümboleid sellest voost. Kui parameeter on tõene, siis genereeritav lekseemihaldur võib lugeda sümboleid suvalisest voost, mis on *CharStream* tüüpi (vastav liides genereeritakse parseriga samasse kataloogi). Seda parameetrit ignoreeritakse juhul, kui USER_TOKEN_MANAGER on seatud tõeseks.

BUILD_PARSER – parameetri vaikeväärtus on tõene. Sel juhul genereeritakse parseri fail. Kui parameetri väärtus on väär, siis parserit ei genereerita. Kasutatakse siis, kui soovitakse genereerida ainult lekseemihaldur ja kasutada seda ilma vastava parserita.

BUILD_TOKEN_MANAGER – parameetri vaikevääratus on tõene. Sel juhul genereeritakse lekseemihalduri fail. Kui parameetri väärtus on väär, siis lekseemihaldurit ei genereerita. Kasutatakse parseri genereerimise kiirendamiseks, kui parandatakse parseri vigu ning leksilisi spetsifikatsioone ei muudeta.

SANITY_CHECK – parameetri vaikeväärtus on tõene. Sel juhul kontrollitakse parseri genereerimisel grammatika süntaktilist ja semantilist korrektsust, vasakpoolset rekursiooni, ebaselgust valikpunktides jms. Selle mittekontrollimine võimaldab küll kiiremat parseri genereerimist, kuid vastavate vigade ignoreerimisel võib parser käituda määramatult.

FORCE_LA_CHECK – parameetri vaikeväärtus on väär. Sel juhul kontrollitakse ebaselgusi ainult valikpunktides, kus LOOKAHEAD väärtus on 1. Tõese väärtuse korral kontrollitakse ebaselgusi kõikides valikpunktides hoolimata LOOKAHEAD spetsifikatsioonist grammatikafailis.

COMMON_TOKEN_ACTION – parameetri vaikeväärtus on väär. Kui parameetri väärtus on tõene, siis meetodi *getNextToken* iga väljakutsumise järel kutsutakse välja ka kasutaja poolt defineeritud meetod *CommonTokenAction*. Kasutaja peab selle meetodi defineerima lekseemihalduri deklaratsioonides TOKEN_MGR_DECLS (vt jaotis 2.3.7). Antud meetodi tagastustüübiks on *void* ja signatuur on:

```
CommonTokenAction(Token)
```

CACHE_TOKENS – parameetri vaikeväärtus on väär. Kui parameetri väärtus on tõene, siis see põhjustab genereeritud parseris ennetähtaegset lisalekseemide eelvaatlust, mis parandab mõningate tegevuste teostamist. Kuid sel juhul ei pruugi interaktiivsed rakendused töötada, sest parser tahab toimetada sünkroonselt sisendist saadavate lekseemidega.

OUTPUT_DIRECTORY – parameetri vaikeväärtuseks on vastav töökataloog. Parameetri sõneväärtusega saab määrata genereeritavate failide asukoha.

2.3.3 JavaCC produktsioonid

Produktsioonid on üldiselt reeglid, millega kirjeldatakse grammatikas esinevad lekseemid ja süntaks. Produktsiooni definitsioon on järgmine:

```
production ::=
                javacode_production
              | regular_expr_production
              | bnf_production
              | token_manager_decl
```

JavaCC-s on nelja tüüpi produktsioone. Java-koodi (*javacode_production)* ja BNF produktsiooni (*bnf_production*) kasutatakse grammatika defineerimiseks, millest genereeritakse parser. Regulaaravaldise produktsiooni (*regular_expr_production*) kasutatakse grammatika lekseemide defineerimiseks, millest genereeritakse lekseemihaldur. Lekseemihalduri deklaratsioone (*token_manager_decl*) kasutatakse lekseemihaldurisse deklaratsioonide lisamiseks. Järgnevates jaotistes käsitletakse kõiki produktsioone detailsemalt.

2.3.4 Java-koodi produktsioon

Java-koodi produktsiooni definitsioon on järgmine:

```
javacode_production ::=
                "JAVACODE"
                java_return_type java_identifier "(" java_parameter_list ")"
                java_block
```

Java-koodi produktsiooni abil on võimalik luua keerulisemaid produktsioone, kui seda võimaldab BNF produktsioon. Java-koodi produktsioon on kasulik juhul, kui on tegemist kontekstist sõltuva grammatikaga või kui mingil muul põhjusel on grammatikat raske kirjutada. Järgnevas näites loeb mitteterminaal *skip_to_matching_brace* sisendvoost lekseeme nii kaua, kuni leitakse lõpetav looksulg:

```
JAVACODE
void skip_to_matching_brace(){
   Token tok;
   int nesting = 1;
   while (true){
      tok = getToken(1);
      if (tok.kind == LBRACE)
           nesting++;
      if (tok.kind == RBRACE){
           nesting--;
           if (nesting == 0)
                break;
      }
      tok = getNextToken();
   }
}
```

Java-koodi produktsioonidega peab olema ettevaatlik, sest valikpunktides asudes on nad JavaCC jaoks sisuliselt mustadeks kastideks. Näiteks:

```
void NT():
{}
{
  skip_to_matching_brace()
|
  some_other_production()
}
```

Antud juhul JavaCC ei tea, kumba parsimisteed valida, seega minnakse alati esimesse. Järgnevas näites probleemi aga enam ei teki, sest edasine parsimistee valitakse “(” ja “{” järgi.

```
void NT():
{}
{
  “{” skip_to_matching_brace()
|
  “(” parameter_list() “)”
}
```

Kui Java-koodi produktsiooni kasutatakse grammatikas valikpunktis, siis JavaCC väljastab vastavasisulise hoiatuse. Sel juhul on mõistlik lisada ilmutatult mõni LOOKAHEAD spetsifikatsioon, mis probleemi lahendab (vt jaotis 2.4).

2.3.5 BNF produktsioon

BNF produktsiooni definitsioon on järgmine:

```
bnf_production ::=
        java_return_type java_identifier “(“ java_parameter_list “)” “:”
        java_block
        “{” expansion_choices “}”
```

BNF produktsioon on JavaCC grammatika spetsifitseerimise standardne vahend. Iga selline produktisoon defineerib uue mitteterminaali. Mitteterminaale kirjutatakse samuti, nagu deklareeritakse Javas meetodit. Iga mitteterminaal transleeritakse meetodiks genereeritavasse parserisse. Mitteterminaali nimest saab meetodi nimi ning vastavaid parameetreid ja tagastamist kasutatakse väärtuste liigutamiseks mööda parsimispuud. BNF produktsiooni parem pool koosneb kahest osast, millest esimese moodustab hulk Java deklaratsioone ja koodi (Java blokk), mis paigutatakse mitteterminaali jaoks genereeritud meetodi algusesse. Iga kord, kui parsimise protsess kasutab seda mitteterminaali, täidetakse vastavad deklaratsioonid ja kood. Deklaratsiooni osa on nähtav kogu vastavas BNF produktsioonis. Kuna JavaCC ei töötle seda osa (eemaldab ainult algavad ja lõpetavad loogelised sulud), siis sellepärast võibki Java kompilaator antud koodiosast hiljem vigu leida. BNF produktsiooni parema poole teine osa koosneb BNF laiendustest (vt jaotis 2.3.11).

2.3.6 Regulaaravaldise produktsioon

Regulaaravaldise produktsiooni definitsioon on järgmine:

```
regular_exp_production ::=
        [lexical_state_list]
        regexpr_kind [ “[” “IGNORE_CASE” “]” ] “:”
        “{“ regexp_spec ( “|” regexp_spec)* “}”
```

Regulaaravaldise produktsiooni kasutatakse leksiliste olemite defineerimiseks, mida hakkab töötlema genereeritav lekseemihaldur. Detailsemalt kirjeldatakse lekseemihalduri tööd jaotises 2.5. Regulaaravaldise produktsioon algab leksiliste olekute spetsifikatsiooniga, millesse lekseemihaldur võib sattuda (vt jaotis 2.3.8).

Standardne leksiline olek on DEFAULT. Kui leksilist olekut ei ole määratud, siis regulaaravaldise produktsioon kasutab DEFAULT olekut. Järgneb regulaaravaldise produktsiooni tüübi kirjeldus (TOKEN, SKIP,...) (vt jaotis 2.3.9) ning mittekohustuslik [IGNORE_CASE], mille olemasolul on regulaaravaldise produktsioon tõstutundetu. Sellel on täpselt sama toime IGNORE_CASE parameetriga, kuid esimene on mõeldud lokaalselt konkreetse regulaarvaldise jaoks. Lõpuks on loetelu regulaaravaldiste spetsifikatsioonidest (vt jaotis 2.3.10).

2.3.7 Lekseemihalduri deklaratsioonid

Lekseemihalduri deklaratsiooni definitsioon on järgmine:

token_manager_decls ::= “TOKEN_MGR_DECLS” “:” java_block

Lekseemihalduri deklaratsioonid algavad võtmesõnaga “TOKEN_MGR_DECLS”, millel järgneb koolon ja siis hulk Java deklaratsioone ja avaldisi (Java blokk). Java bloki osa pannakse genereeritavasse lekseemihaldurisse. JavaCC grammatikafailis võib olla ainult üks lekseemihalduri deklaratsiooni kirjeldus.

2.3.8 Leksiliste olekute loetelu

Leksiliste olekute loetelu definitsioon on järgmine:

lexical_state_list ::=
“<” “*” “>”
| “<” java_identifier (“,” java_identifier)* “>”

Leksiliste olekute loetelu kirjeldab leksiliste olekute hulka, mis kehtivad regulaaravaldise produktsioonide kohta. Kui on kirjutatud “<*>”, siis regulaaravaldise produktsioon vastab kõigile leksilistele olekutele. Vastasel korral aga kõigile leksilistele olekutele, mis asuvad noolsulgude vahel.

2.3.9 Regulaaravaldise produktsiooni tüübid

Regulaaravaldise produktsiooni tüüpide definitsioon on järgmine:

```
regexp_kind ::=
                  “TOKEN”
                | “SPECIAL_TOKEN”
                | “SKIP”
                | “MORE”
```

TOKEN – selles regulaaravaldise produktsioonis asuv regulaaravaldis kirjeldab grammatika lekseeme. Lekseemihaldur genereerib *Token* tüüpi objekti igale vastavale regulaaravaldisele ja tagastab selle parserile.

SPECIAL_TOKEN – selles regulaaravaldise produktsioonis asuvad regulaaravaldised kirjeldavad erilekseeme. Need on nagu tavalised lekseemid, kuid ei oma tähendust parsimisel, s.t. BNF produktsioonid ignoreerivad neid. Siiski antakse need edasi parserile, mille meetodites saab neid kasutada. Erilekseemid ühendatakse naabruses asuvate tõeliste lekseemidega, kasutades selleks *Token* tüüpi välja *specialToken.* Erilekseemid on kasulikud selliste lekseemide käsitlemisel, mis ei oma tähendust parsimisel, kuid on siiski oluliseks osaks arvutitekstis (näiteks kommentaarid).

SKIP – siin regulaaravaldised kirjeldatavad tühilekseeme, mida lekseemihalduri poolt lihtsalt ignoreeritakse. See aga ei tähenda, et tühilekseeme ei või esineda omakorda teistes lekseemides.

MORE – mõnikord on kasulik koostada lekseeme järk-järgult, mis alles lõpuks antakse edasi parserile. Siin kirjeldatud lekseemidele leitud ühtivused jäetakse meelde puhvris, kuni leitakse järgmine lekseem või erilekseem. Siis ühendatakse kõik vasted puhvris lõplikuks lekseemiks/erilekseemiks ja edastatakse parserile. Kui leitakse tühilekseem, siis puhver tühjendatakse.

2.3.10 Regulaaravaldise spetsifikatsioon

Regulaaravaldise spetsifikatsioon on järgmine:

regexp_spec ::= regular_expression [java_block] [“:” java_identifier]

Regulaaravaldise spetsifikatsioon algab tegelike leksiliste olemite kirjeldusega (*regular_expression*) (vt jaotis 2.3.13). Väljad *java_block* ja *java_identifier* (leksilise oleku identifikaator) pole kohustuslikud. Kui leksilist olekut pole määratud, siis lekseemihaldur töötleb lekseemi olekus DEFAULT. Kui leitakse vastava leksilise oleku mingi regulaaravaldisega ühtiv lekseem, siis esmalt käivitatakse olemasolul vastav Java blokk (leksilised aktsioonid) ja pärast seda muudab lekseemihaldur oma olekut vastavalt spetsifikatsioonile.

2.3.11 Laiendusvalikud

Laiendusvalikute definitsioon on järgmine:

```
expansion_choices ::= expansion (“|” expansion)*

expansion ::= (expansion_unit)*

expansion_unit ::=
        local_lookahead
      | java_block
      | “(“ expansion_choices “)” [“+” | “*” | “?”]
      | “[“expansion_choices “]”
      | [java_assignment_lhs “=”] regular_expression
      | [java_assignment_lhs “=”] java_identifier “(“ java_expression_list “)”
```

Laiendusvalikud (*expansion_choices*) koosnevad ühest või mitmest laiendusest (*expansion*), mis on eraldatud üksteisest sümboliga “|”.

Laiendus koosneb laiendusüksuste (*expansion_unit*) järjendist. Näiteks laiendus “{“ decls() “}” koosneb kolmest laiendusüksusest: “{“, decls() ja “}”. Laienduse ühtimine on konkatenatsioon üksikute laiendusüksuste ühtimisest.

Laiendusüksuseks võib olla (1) lokaalne LOOKAHEAD spetsifikatsioon, mis annab genereeritud parserile instruktsioone valikpunktides toimimiseks; (2) hulk Java deklaratsioone ja koodi, mis on ümbritsetud loogeliste sulgudega ja mida nimetatakse ka parseri aktsioonideks (täidetakse alati, kui analüüsimine jõuab edukalt sellesse punkti, kuid mitte eelvaatluse ajal); (3) üks või mitu sulgudesse pandud laiendusvalikut. Sulgudele järgneb mittekohustuslikult üks järgmistest sümbolitest:

"+" – sulgudes asuv blokk võib esineda üks või rohkem korda;
"*" – sulgudes asuv blokk võib esineda null või rohkem korda;
"?" – sulgudes asuv blokk võib esineda null või üks korda. Alternatiivseks variandiks kirjutisele (...)? on [...].

Laiendusüksuseks võib olla (4) regulaaravaldis. Sel juhul luuakse vastavalt regulaaravaldisele leitud lekseemi jaoks *Token* tüüpi objekt, mille võib omistada mõnele muutujale, pannes antud regulaaravaldisele prefiksiks "muutuja =". Omistamist ei toimu eelvaatluse ajal. Laiendusüksuseks võib olla (5) ka mitteterminaal. Sel juhul esineb laiendusüksus meetodi väljakutse nimena. Mitteterminaali edukal parsimisel asendatakse parameetrid meetodisse ja tagastatakse meetodi tagastusväärtus (kui meetod polnud *void* tüüpi). Tagastusväärtuse võib omistada muutujale, kuid eelvaatlusel seda ei tehta. JavaCC kontrollib ka vasakrekursiooni olemasolu.

2.3.12 Lokaalne LOOKAHEAD

Lokaalne LOOKAHEAD on defineeritud järgmiselt:

```
local_lookahead ::=
                "LOOKAHEAD" "(" [java_integer_literal] [","]
                [expansion_choice] [","] ["{" java_expression "}"] ")"
```

LOOKAHEAD-i kasutatakse selleks, et mõjutada parseri otsuseid valikpunktides (vt jaotis 2.4). On olemas kolme tüüpi LOOKAHEAD parameetreid, millest vähemalt üks peab olemas olema:

Tõkestatud LOOKAHEAD – sellega määratakse maksimaalne ettevaadatavate lekseemide arv enne otsuse tegemist valikpunktis. Sellise kohaliku eelvaatluse kasutamisega kehtestatakse ümber globaalne eelvaatlus. LOOKAHEAD-i limiit mõjub ainult valikpunktis, mujal seda ignoreeritakse.

Süntaktiline LOOKAHEAD – sisaldab laiendusvalikuid, mille ühtimisel valitakse parsimisteeks vastava LOOKAHEAD-i mõjus olev valik. Kui LOOKAHEAD pole valikpunktis, siis teda ignoreeritakse.

Semantiline LOOKAHEAD – sisaldab loogilist avaldist, mille väärtus arvutatakse iga kord, kui parsimine selleni jõuab. Kui avaldis on tõene, siis parsimine jätkub normaalselt. Kui avaldis on väär ja antud LOOKAHEAD asub valikpunktis, siis seda valikut ei valita vaid hakatakse järgmisi proovima. Kui avaldis on väär ja antud LOOKAHEAD ei asu valikpunktis, siis parser lõpetab töö ja väljastab veateate.

2.3.13 Regulaaravaldis

Regulaaravaldise definitsioon on järgmine:

```
regular_expression ::=
        java_string_literal
      | “<” [[“#”] java_identifier “:”] complex_regular_expression_choices “>”
      | “<” java_identifier ”>”
      | “<” “EOF” ”>”
```

Grammatikafailis võib regulaaravaldist kirjutada kas regulaaravaldise spetsifikatsiooni (vt jaotis 2.3.10) või laiendusüksusena laiendusse (vt jaotis 2.3.11). Regulaaravaldis võib olla sõneliteraal (*java_string_literal*). Sel juhul parsitav sisend ühtib regulaaravaldisega, kui lekseemihaldur on vastavas leksilises olekus ning sisendist tulev sümbolite hulk on sama, mis sõneliteraal (vajadusel tõstutundlikkuse täpsusega). Regulaaravaldis võib olla ka palju keerulisem, sisaldades omakorda teisi regulaaravaldisi. Sellised avaldised pannakse noolsulgude vahele ja soovi korral võib nad märgendada edaspidise viidatavuse huvides. Kui märgendile eelneb sümbol “#”, siis sellele avaldisele ei saa viidata laiendusüksustes. Taoliselt märgendatud regulaaravaldist nimetatakse privaatseks ja lekseemihaldur ei ühita seda kui lekseemi. Privaatsete regulaaravaldiste eesmärgiks ongi ainult keerulisemate regulaaravaldiste

kirjutamise hõlbustamine. Regulaaravaldis võib viidata ka eeldefineeritud regulaaravaldisele “<EOF>”, mis tähendab failiterminaatorit. Järgnevalt näide ühest regulaaravaldistest:

```
TOKEN :
{
< FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
            (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
>
|
< #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}
```

Antud näites defineeritakse lekseem *FLOATING_POINT_LITERAL*, kasutades teist defineeritud lekseemi *EXPONENT*. Lekseem *EXPONENT* on kirjutatud privaatsena, sest seda kasutatakse üksnes teiste lekseemide defineerimisel.

complex_regular_expression_choices ::=
complex_regular_expression (“|” complex_regular_expression)*

Konstruktsioon *complex_regular_expression_choices* koosneb loetelust, kus on üks või mitu sümboliga “|” eraldatud avaldist *complex_regular_expression*. Kogu konstruktsioon ühtib, kui mõni loetelus olev avaldistest ühtib.

complex_regular_expression ::= (complex_regular_expression_unit)*

Kompleksne regulaaravaldis koosneb komplekssete regulaaravaldiste üksuste järjendist ning ühtimine tähendab vastavate üksuste ühtimiste konkatenatsiooni.

complex_regular_expression_unit ::=
java_string_literal
| “<” java_identifier “>”
| caracter_list
| “(“ complex_regular_expression_choice “)” [“+” | “*” | “?”]

Kompleksne regulaaravaldise üksus võib olla sõneliteraal ning sel juhul on täpselt üks ühtimine – seesama sõne ise. Üksus võib olla viide teisele regulaaravaldisele. Vastav regulaaravaldis peab olema muidugi märgendatud, et talle saaks viidata (tsükleid ei tohi

tekkida). Üksuseks võib olla sümbolite järjend, millega saab sümbolite hulka defineerida. Seda tüüpi üksus ühtib sümbolitega, mis asuvad vastavas hulgas. Üksus võib olla sulgudes olev *complex_regular_expression_choices*. Sel juhul iga üksuse legaalne ühtimine tähendab legaalset alamühtimist. Sulgudele võib järgneda veel üks järgnevatest sümbolitest:

"+" – üks või rohkem vastavat avaldist
"*" – null või rohkem vastavat avaldist
"?" – null või üks vastavat avaldist

Erinevalt BNF laiendustest ei ole regulaaravaldises konstruktsioon [...] sama tähendusega mis "(...)?", sest [...] kasutatakse sümbolite järjendi kirjeldamiseks.

caracter_list ::= ["~"] "[" [caracter_descriptor ("," caracter_descriptor)*] "]"

Sümbolite järjend (*caracter_list*) kirjeldab hulga sümboleid. Sellega ühtib suvaline sümbol, mis sinna kuulub. Iga *caracter_descriptor* kirjeldab ühte sümbolit või sümbolite vahemikku. Prefiks "~" mõjub eitusena, st sümbolite hulga moodustavad kõik ülejäänud sümbolid.

caracter_descriptor ::= java_string_literal ["-" java_string_literal]

Deskriptor (*caracter_descriptor*) võib olla üksik sõneliteraal või vahemik, mis kirjeldab kõigi sümbolite hulka, mis jäävad vahemikku koos vahemikku määrava kahe sümboliga.

2.4 LOOKAHEAD

Parseri põhitöö on lugeda sisendvoogu ja kindlaks teha, kas see vastab grammatika reeglitele või mitte. Vastavuse kindakstegemine kõige üldisemalt võib olla aga üsnagi ajamahukas. Näide:

```
void Input():
{}
{
“a” BC() “c”
}

void BC():
{}
{
“b” [“c”]
}
```

On ilmne, et antud grammatikale vastab täpselt kaks sõne: “abc” ja “abcc”.
Kontrolliks läbime grammatika sõnega “abc”.

1. Grammatikast näeme, et esimene sisendsümbol peab olema “a”, seega sobib.
2. Nüüd jätkame mitteterminaaliga *BC*. Seal peab olema järgmine sisendsümbol “b”, seega sobib.
3. Nüüd jõuame grammatikas valikpunkti, kus võime siseneda konstruktsiooni [...] ja ühtida sealse sümboli või hoopis ignoreerida seda. Otsustame siiski siseneda, seega järgmine sümbol peab olema “c”, seega sobib.
4. Nüüd oleme lõpetanud mitteterminaali *BC* töötlemise ja läinud tagasi mitteterminaali *Input*. Edasi määrab grammatika, et järgmine sümbol peab olema veel üks “c”. Seega tekib probleem.
5. Üldjuhul sellise probleemi tekkimisel järeldame, et oleme teinud kuskil ebaõnnestunud valiku. Antud juhul oli selleks valik punktis 3. Seega lähme tagasi sinna ja teeme teise valiku ning proovime edasi. Seda tegevust nimetatakse tagurduseks (*backtracking*).
6. Nüüd oleme tagurdanud ja teinud teise valiku punktis 3, nimelt ignoreerinud [...]. Nüüd on mitteterminaal *BC* töödeldud ja lähme tagasi mitteterminaali *Input*. Edasi määrab grammatika, et järgmine sümbol peab olema veel üks “c”, nii ka on ja järelikult “abc” ühtib grammatikaga.

Seega oleme jõudnud edukalt grammatika (mitteterminaali *Input*) lõppu ja ühtinud edukalt sõne “abc”.

Antud näite põhjal võib järeldada, et põhiliseks probleemiks sisendi vastavuse kontrollimisel grammatikaga on tekkida võiv suur arv valikuid ja tagurdusi, mis omakorda kulutavad palju aega. Samuti võib ajakulu sõltuda sellest, kui hästi on

grammatika koostatud. Erinevatest ühe ja sama keele grammatikatest võib saada oluliselt erineva efektiivsusega parsereid. Näiteks järgmise grammatika põhjal parsitakse kiiremini sama sisendit, kui eelmise näite korral:

```
void Input():
{}
{
“a” “b” “c” [“c”]
}
```

Järgmine näide on aga parsimise seisukohalt veelgi aeglasem kui üle-eelmine, sest parser peab tagurdama kogu tee kuni alguseni:

```
void Input():
{}
{
“a” “b” “c” “c”
|
“a” “b” “c”
}
```

Grammatika võib kirjutada ka kujul:

```
void Input():
{}
{
“a” ( BC1() | BC2() )
}

void BC1():
{}
“b” “c” “c”
}

void BC2():
{}
“b” “c” [“c”]
}
```

Antud grammatika korral võib ühtida sõne “abcc” kahel erineval viisil, seega tuleb arvestada ebaselgusega.

Taolise tagurduse jõudluse tagamine pole aga jõukohane enamusele süsteemidest, mis sisaldavad parserit. Sellepärast ei tagurda parserid üldisel meetodil (või ei tagurda üldse), vaid teevad valikpunktides piiratud informatsiooni põhjal otsuseid ning töötavad edasi nende põhjal. JavaCC abil genereeritud parserid teevad samuti valikpunktides

otsuseid, baseerudes sisendvoost tulevatele edasistele lekseemidele, ning otsuse vastuvõtu korral sellest enam ei loobuta ehk selles punktis enam tagurdust ei teostata. Protsessi, milles vaadatakse valikpunktides otsustuste tegemiseks lisaks ette lekseeme sisendvoost, nimetatakse eelvaatluseks. Kui grammatikas tuleb õige otsustuse tegemiseks valikpunktis informatsioonist puudu, siis sel juhul JavaCC väljastab hoiatuse. Grammatika kirjutamisel tuleks seega silmas pidada, et grammatika oleks kirjutatud võimalikult lihtsalt, ja keerulisemates valikpunktides võiks sisse kirjutada vihjeid parserile, et aidata teha õigeid otsuseid.

2.4.1 Valikpunktid JavaCC grammatikas

Grammatikas on nelja erinevat tüüpi valikpunkte:

- Laiendus (*exp*1 | *exp*2 | ....). Sel juhul peab genereeritud parser kuidagi kindlaks tegema, millist ühte järgmistest *exp*1, *exp*2, ... valida parsimise jätkamiseks.
- Laiendus (*exp*)? ehk [*exp*]. Sel juhul peab genereeritud parser kuidagi kindlaks tegema, kas valida *exp* või jätkata selleta.
- Laiendus (*exp*)*. Siin on tegu sama juhuga, mis eelminegi, kuid lisaks sellele tuleb lekseemi ühtimise korral (kui valiti *exp*) sama valikut järgmise lekseemi korral uuesti kaaluda.
- Laiendus (*exp*)+ on analoogne eelmisega, kuid esimene *exp* valik on kohustuslik.

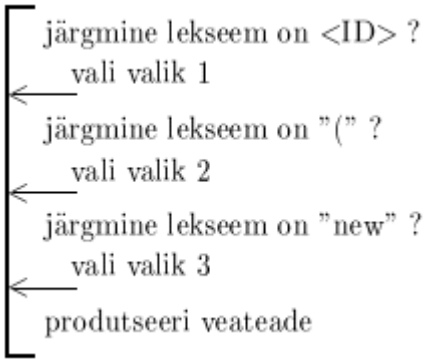
Meenutame, et ka lekseemide spetsifikatsioonid, mis esinevad noolsulgude vahel omavad samuti valikpunkte. Kuid sealsed otsused teeb juba lekseemihaldur (vt jaotis 2.5).

2.4.2 Eelvaatluse vaikealgoritm

Eelvaatluse vaikealgoritm võtab sisendist ainult ühe lekseemi ning kasutab seda valikpunktis otsuse tegemiseks. Järgnev näide illustreerib situatsiooni:

```
void basic_expr():
{}
{
<ID> “(“ expr() “)”           //valik 1
|
“(“ expr() “)”                //valik 2
|
“new” <ID>                    //valik 3
}
```

Algoritm töötab järgmiselt:



See grammatika oli koostatud nii, et vaikealgoritm töötas korrektselt. Samuti võis tähele panna, et algoritm töötab ülevalt alla, st. kui valiti valik 1, siis teisi valikuid isegi ei arvestatud. Oletame, et eelnenud grammatika on modifitseeritud järgmisele kujule:

```
void basic_expr():                                    (*)
{}
{
<ID> “(“ expr() “)”     //valik 1
|
“(“ expr() “)”          //valik 2
|
“new” <ID>              //valik 3
|
<ID> “.” <ID>           //valik 4
}
```

Nüüd näeme, et vaikealgoritm valib alati valiku 1, kui järgnev lekseem on <ID>, ja mitte kunagi ei valita valikut 4, isegi siis, kui lekseemile <ID> järgneb “.”. Kui sellel grammatikal rajanevale parserile ette anda “id1.id2”, siis väljastatakse viga, et “.” asemel oodati hoopis “(“. Seega hoiatab JavaCC juba parseri genereerimise ajal valiku konfliktist:

Warning: Choice conflict involving two expansions at line 48, column 1 and line

54, column 1 respectively.

A common prefix is: <ID>

Consider using a lookahead of 2 for earlier expansion.

JavaCC hoiatab selle eest, et grammatikafailis on situatsiooni, mis võib põhjustada eelvaatluse vaikealgoritmi ebaadekvaatset käitumist. Genereeritud parser küll töötab, kuid ei tee otseselt seda, mida oodatakse. Olgu järgnev näide:

```
void identifier_list():                                    (**)
{}
{
<ID> ( “,” <ID> )*
}
```

Oletame, et esimene <ID> on juba ühtinud ning parser on jõudnud valikpunkti (konstruktsiooni (...)*). Eelvaatluse algoritm töötab järgmiselt:

järgmine lekseem on ”,” ?
  vali pesastatud laiendus (st sisene (...)* konstruktsiooni)
  tarbi lekseem ”,”
    ? (järgmine lekseem on <ID>) tarbi see : produtseeri veateade

Antud näites (**) on näha, et valiku tegemisel ei vaadata (...)* konstruktsioonist välja. Olgu nüüd järgmine grammatika:

```
void funny_list():                                         (***)
{}
{
identifier_list() “,” <INT>
}
```

Kui vaikealgoritm teeb valiku (“,” <ID>)*, siis ta läheb alati konstruktsiooni (...)*, sest järgmine lekseem oli “,”. Oletame, et parseri sisendiks on “id1, id2, 5”, siis väljastatakse viga, et loeti “5”, kuid oodati hoopis <ID>. Seega annab JavaCC parseri loomisel vastavasisulise hoiatuse, et grammatikafailis avastati situatsioon, kus eelvaatluse vaikealgoritm ei taga ootuspärast töötlust. Analoogiliselt käitub eelvaatluse vaikealgoritm ka konstruktsioonide (...)+ ja (...)? korral.

2.4.3 Mitut lekseemi kasutav eelvaatluse algoritm

Siiani käsitleti eelvaatluse vaikealgoritmi, mis enamuses olukordades töötas korrektselt. Seal, kus ta ei töötanud ootuspäraselt, genereeriti vastavasisuline hoiatus. Kui JavaCC töötleb grammatikafaili ilma ühegi hoiatuseta, siis nimetatakse antud grammatikad LL(1). Seega LL(1) grammatikad on need, mida suudavad käsitleda ülalt alla töötavad parserid, kasutades eelvaatuse algoritmis valiku tegemisel ainult ühte lekseemi. Kui JavaCC annab eelpool mainitud hoiatusi, siis võib toimida järgmiselt:

1) Grammatikat võib modifitseerida nii, et hoiatusi enam ei anta. Näiteks modifitseerime näite (*) grammatikat nii, et ta oleks LL(1):

```
void basic_expr():
{}
{
<ID> (“(“ exp() “)” | “.” <ID> )
|
“(“ expr() “)”
|
“new” <ID>
}
```

Seega on likvideeritud neljanda valiku arvestamata jätmine, kombineerides neljanda valiku esimese valikuga. Antud viisil grammatika modifitseerimist LL(1) grammatikaks nimetatakse vasaktegurduseks (*left factoring*).

Järgnevalt on tehtud eelmise jaotise viimane näide (***) LL(1) grammatikaks:

```
void funny_list():
{}
{
<ID> “,” (<ID> “,”)*  <INT>
}
```

2) Teine võimalus on sättida globaalse parameetri LOOKAHEAD väärtust suuremaks (vaikeväärtus oli 1) või määrata selle parameetri väärtust lokaalselt konstruktsioonis LOOKAHEAD(...). Variandi 1 eelis 2 ees on see, et grammatika esitus on parem, st. JavaCC poolt genereeritud parser suudab käsitleda LL(1) konstruktsioone palju kiiremini kui muid. Variant 2 on eelistatavam siis, kui tahetakse kirjutada lihtsat grammatikat, mis oleks kergesti arendatav ja eeskätt inimsõbralik, mitte niivõrd

arvutisõbralik. Kuid vahel on see ka ainuke võimalus. Näiteks juhul, kui on esindatud kasutaja aktsioonid:

```
void basic_expr():
{}
{
{initMethodTables();} <ID> “(“ expr() “)”
|
“(“ expr() “)”
|
“new” <ID>
|
{initObjectTables();} <ID> “.” <ID>
}
```

Kuna aktsioonid *initMethodTables* ja *initObjectTables* on erinevad, siis vasaktegurdust ei saa kasutada.

2.4.4 Globaalse eelvaatluse spetsifikatsioon

Eelvaatluse spetsifikatsiooni võib määrata parameetri LOOKAHEAD kaudu, kas siis käsurealt või otse grammatikafaili sisse kirjutatult. Parameetri väärtus on tüüpi *int* ja mis tähistab lekseemide arvu, mida vaadeldakse valikpunktis enne otsuse tegemist. Nagu varemgi mainitud, on LOOKAHEAD parameetri vaikeväärtus 1, mis vastab eelnevalt kirjeldatud eelvaatluse vaikealgoritmile. Kui oletame, et selle parameetri väärtus on 2, siis eelvaatluse algoritm vaatab senise ühe asemel kaks lekseemi edasi enne otsuse tegemist valikpunktis. Jaotises 2.4.2 toodud näite (*) korral valitakse nüüd valik 1 ainult siis, kui kaks järgmist lekseemi on <ID> ja “(“. Kuna valik 4 tehakse ainult siis, kui kaks järgmist lekseemi on <ID> ja “.”, siis parser töötab korrektselt. Samuti kaob jaotise 2.4.2 näite (***) korral probleem, sest parser siseneb konstruktsiooni (...)* ainult siis, kui kaks järgmist lekseemi on “,” ja <ID>.

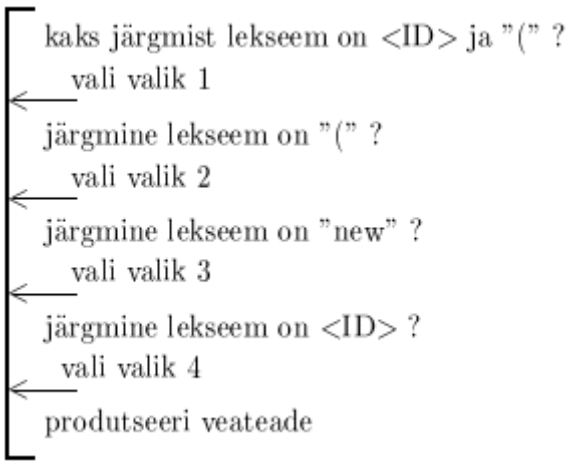
Määrates globaalse parameetri LOOKAHEAD väärtuseks 2, muutub parsimise algoritm LL(2)-ks. Kuna vastavat parameetrit saab vabalt valida (0..maxInt), siis JavaCC poolt genereeritud parsereid nimetatakse LL(k) parseriteks.

2.4.5 Lokaalse eelvaatluse spetsifikatsioon

Eelvaatlust saab spetsifitseerida ka lokaalselt ainult ühe konkreetse valikpunkti jaoks. Sel juhul jääb enamus grammatikast LL(1)-ks ning parser on kiirem, võimaldades samas LL(k) paindlikkust. Näiteks on modifitseeritud jaotise 2.4.2 näite (*) grammatikat, kaotades seal segasuse valikpunktis:

```
void basic_expr():
{}
{
LOOKAHEAD(2)
<ID> “(“ expr() “)”          //valik 1
|
“(“ expr() “)”               //valik 2
|
“new” <ID>                   //valik 3
|
<ID> “.” <ID>                //valik 4
}
```

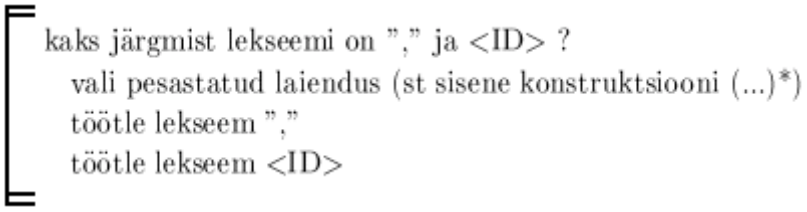
Seejuures ainult esimene valik on LOOKAHEAD mõjupiirkonnas, kõik teised kasutavad aga endiselt ühte lekseemi eelvaatlusel:



Jaotises 2.4.2 oleva näite (**) võib modifitseerida kujule:

```
void identifier_list():
{}
{
<ID> (LOOKAHEAD(2) “,” <ID>)*
}
```

LOOKAHEAD on asetatud valiku tegemise algusesse ja toimib järgmiselt:



Põhimõtteliselt ongi tungivalt soovitatav vältida globaalset eelvaatlust. Sest enamus grammatikat on LL(1) ja pole mõtet asjata parserit koormata, konverteerides kogu grammatika LL(k)-ks. Piisab ainult selle vähese osa, mis pole LL(1), viimisest lokaalse eelvaatluse abil LL(k)-ks. Kui grammatika ja sisendandmed on mahult väikesed, siis pole sel muidugi olulist tähtsust. JavaCC genereerib hoiatusteated siis, kui leitakse segasust valikpunktides, kuid lokaalseid LOOKAHEAD konstruktsioone ei kontrollita. Eeldatakse, et grammatika koostaja teab, mida ta teeb. Järgnev näide sellest:

```
void IfStm():
{}
{
“if” C() S() [ “else” S() ]
}

void S():
{}
{
...
|
IfStm()
}
```

Siin kerkib tuntud probleem: millise “if” juurde kuulub “else” direktiivis “if C1 if C2 S1 else S2”. Põhimõtteliselt võib “else S2” kuuluda nii esimese kui ka teise “if” juurde, kuid standardses interpretatsioonis nähakse, et “else” kuulub lähima “if” juurde. Vaikealgoritm töötab antud juhul ootuspäraselt, kuid siiski annab JavaCC vastavasisulise hoiatuse. Hoiatuse eemaldamiseks tuleb lihtsalt ilmutatult öelda, et teame mida tahame:

```
void IfStm():
{}
{
“if” C() S() [LOOKAHEAD(1) “else” S()]
}
```

Selleks, et sundida JavaCC-d kõikide valikpunktide segasusi kontrollima (ka lokaalseid), tuleb parameetri FORCE_LA_CHECK väärtus tõeseks seada.

2.4.6 Süntaktiline eelvaatlus

Olgu järgnevaks näiteks programmeerimiskeele Java enda grammatika üks produktsioonidest:

```
void TypeDeclaration():
{}
{
ClassDeclaratsion()
|
InterfaceDeclaration()
}
```

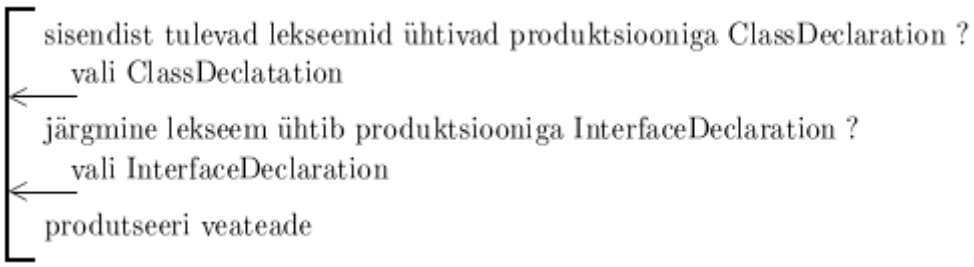
Süntaktiliselt võib *ClassDeclaration* alata suvalise arvu sõnedega “abstract”, “final” ja “public”. Alles hilisem semantika kontroll produtseerib veateate, kui sama piiritleja esines mitu korda. Seda aga ei juhtu enne, kui parsimine on täielikult lõpetatud. Oletame, et suur arv (näiteks 100) järgmisi lekseeme sisendist on “abstract”, mis eelnevad lekseemile “interface”. On ilmne, et fikseeritud pikkusega LOOKAHEAD (näiteks LOOKAHEAD(100)) ei kvalifitseeru. Lahenduseks peaks määrama LOOKAHEAD-i lõpmatuseks. Üks võimalus selleks on LOOKAHEAD-i väärtustamine nii suure arvuga kui võimalik (näiteks suurim *int* väärtus):

```
void TypeDeclaration():
{}
{
LOOKAHEAD(2147482647)
ClassDeclaratsion()
|
InterfaceDeclaration()
}
```

Teine võimalus sama efekti saavutamiseks on aga süntaktiline eelvaatlus, milles spetsifitseeritakse laiendus ning selle ühtimisel võetaksegi järgnev valik. Järgmine näide illustreerib seda:

```
void TypeDeclaration():
{}
{
LOOKAHEAD(ClassDeclaration())
ClassDeclaratsion()
|
InterfaceDeclaration()
}
```

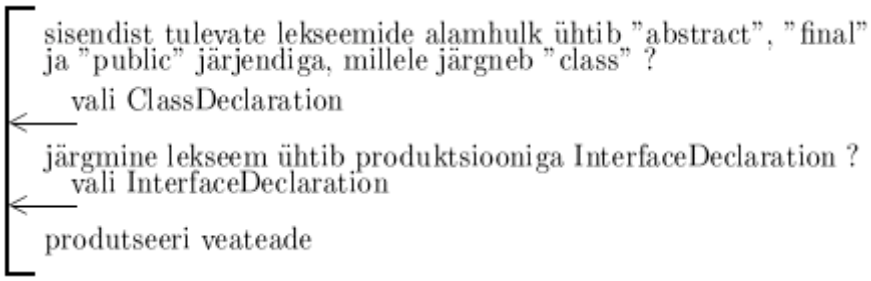
Seega püütakse öelda, et:



Antud juhul kulutab süntaktiline eelvaatlus liigselt parseri aega ja põhjustab ülearuseid kontrollimisi. Idee kohaselt võiks eelvaatlus lõpetada kontrolli juba lekseemi “class” leidmisel. Kuid praegusel juhul teostatakse täielik klassi deklaratsiooni parsimine, mis on ilmne ajakulu. Selle probleemi saab lahendada kitsama LOOKAHEAD laienduse spetsifitseerimisega:

```
void TypeDeclaration():
{}
{
LOOKAHEAD((“abstract” | “final” | “public”)* “class”)
ClassDeclaratsion()
|
InterfaceDeclaration()
}
```

Seega püütakse öelda, et



Niimoodi tehes peatub valiku määramise algoritm kohe, kui ta näeb lekseemi “class” ja seega tehaksegi valiku otsus nii kiiresti kui võimalik. Samas võime endiselt lisada ka LOOKAHEAD arvulise väärtuse, määrates ettevaadatavate lekseemide arvu:

```
void TypeDeclaration():
{}
{
LOOKAHEAD(10, (“abstract” | “final” | “public”)* “class”)
ClassDeclaratsion()
|
InterfaceDeclaration()
}
```

Antud juhul on ettevaadatavate lekseemide arv piiratud 10-ga. Kui see limiit täitub ja käsitletavad lekseemid ühtivad laiendusega (“abstract” | “final” | “public”)* “class”, siis valitakse klassi deklaratsioon. Näiteks, kui limiit on 5, siis sisendi: “abstract abstract final final final” korral valitakse klassi deklaratsioon, mistõttu sisendi “abstract abstract final final final interface” korral antakse veateade, sest kuuendaks lekseemiks oodati hoopis lekseemi “class”. Kui LOOKAHEAD arvulist väärtust pole määratud, siis vaikimisi on selleks suurim täisarvu väärtus (2147483647).

2.4.7 Semantiline eelvaatlus

Olgu taas järgmine näide:

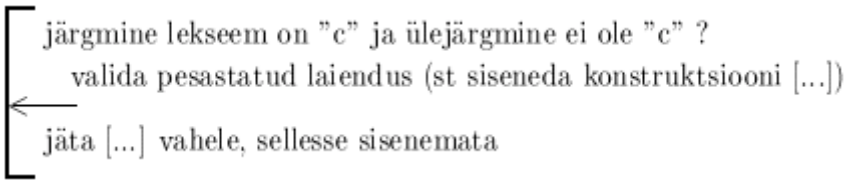
```
void Input():
{}
{
“a” BC() “c”
}

void BC():
{}
{
“b” [“c”]
}
```

Oletame, et on mingi põhjus grammatikat just nii kirjutada. Probleem oli selles, et LL(1) algoritm valis alati [“c”], kui nähakse “c”-d ja seetõttu ei ühtinud kunagi sõne “abc”, vaid ainult “abcc”. Vaja oleks aga, et antud valik tehtaks ainult siis, kui järgmine lekseem on “c” ja ülejärgmine ei ole “c”. Probleemi lahendamiseks võibki kasutada semantilist eelvaatlust, milles saab spetsifitseerida suvalise tõeväärtusavaldise. Selle lahendusväärtus määrab, milline otsus tehakse valikpunktis. Järgnev näide illustreerib situatsiooni:

```
void BC():
{}
{
“b”
[ LOOKAHEAD( {getToken(1).kind == C && getToken(2).kind != C} )
<C:”c”>
]
}
```

Kõigepealt antakse lekseemile “c” tähistus C, et saaks mujalt sellele viidata. Tõeväärtusavaldis annab soovitud tulemuse:



Antud näite võib ümber kirjutada süntaktilist ja semantilist eelvaatlust kombineerides:

```
void BC():
{}
{
“b”
[ LOOKAHEAD( “c”, {getToken(2).kind != C} )
<C:”c”>
]
}
```

2.4.8 Üldine LOOKAHEAD struktuur

Üldine LOOKAHEAD struktuur on järgmine:

LOOKAHEAD(amount, expansion, {boolean_expression})

Väli *amount* spetsifitseerib ettevaadatavate lekseemide arvu, *expansion* spetsifitseerib laienduse süntaktilise eelvaatluse teostamiseks ja *boolean_expression* on tõeväärtusavaldis semantilise eelvaatluse jaoks. Vähemalt üks kolmest väljast peab alati olemas olema. Väljad eraldatakse üksteisest komadega. Puuduvatele väljadele antakse aga järgmised vaikeväärtused:

amount – kui *expansion* on esindatud, siis 2147483647, vastasel korral 0 (*boolean_expression* peab olema esindatud). Kui *amount* on 0, siis süntaktilist eelvaatlust ei teostata. Samas *amount* ei mõjuta semantilist eelvaatlust.

expansion – vaikimisi laienduseks saab parajasti vaadeldav laiendus.

boolean_expression – vaikimisi tõene.

2.5 Lekseemihalduri töö

Lekseemihalduri põhiliseks tööeesmärgiks on sisendvoo andmetest lekseemide koostamine ning nende edastamine parserile. JavaCC poolt genereeritud lekseemihaldur on igal ajamomendil parajasti ühes leksilises olekus. Iga leksiline olek on määratud identifikaatoriga. Standardset leksilist olekut nimetatakse DEFAULT. Kui lekseemihaldur initsialiseeritakse, siis vaikimisi alustab ta olekust DEFAULT (alustavat leksilist olekut saab spetsifitseerida parameetrina lekseemihalduri objekti konstrueerimisel). Iga leksiline olek koosneb järjestatud regulaaravaldiste loendist, kus järjekorra määrab grammatikafail.

Kõiki nelja liiki regulaaravaldisi (SKIP, MORE, TOKEN ja SPECIAL_TOKEN), mis esinevad grammatikas laiendusüksustena, käsitletakse DEFAULT olekus ja nende järjekord on määratud positsiooniga grammatikafailis.

Lekseeme koostatakse järgmiselt. Kõiki kehtivas leksilises olekus esinevaid regulaaravaldisi käsitletakse potentsiaalsete sobivate kanditaatidena. Lekseemihaldur kasutab võimalikult palju sümboleid sisendist, et need ühtiks ühegagi regulaaravaldistest. Seega lekseemihaldur eelistab pikimat võimalikku ühtivust. Kui lekseemihaldur leiab mitu sama pikka ühtivust, siis ta valib selle ühtinud regulaaravaldise, mis esines grammatikafailis esimesena. Pärast lekseemi ühtimist võib toimuda leksilise oleku muutus, kui see on grammatikas spetsifitseeritud. Kui uut leksilist olekut ei ole määratud, siis lekseemihaldur jätkab tööd endises olekus.

Regulaaravaldise tüüp spetsifitseerib vastavad tegevused (ühtimise korral):

- SKIP – ühtinud sõne heidetakse kõrvale (pärast suvalist leksilise aktsiooni täitmist).
- MORE – jätkatakse (sõltumata järgmisest olekust) ühtinud sõnest edasi, kusjuures antud sõne saab prefiksiks uuele ühtinud sõnele.
- TOKEN – ühtinud sõnest koostatakse lekseem ja saadetakse see parserile (või muule väljakutsujale).
- SPECIAL_TOKEN – luuakse erilekseem, mis ei osale parsimisel.

Kui avastatakse failiterminaator, siis luuakse lekseem <EOF>, hoolimata kehtivast leksilisest olekust. Kui failiterminaator tuvastatakse regulaaravaldise ühtimise ajal või kohe pärast MORE edukat ühtimist, siis genereeritakse veateade.

Kui regulaaravaldis on ühtinud, siis täidetakse leksiline aktsioon, milles kõik lekseemihalduri deklaratsioonides olevad muutjad ja meetodid on kättesaadavad. Kohe pärast seda muudab lekseemihaldur oma olekut vastavalt spetsifikatsioonile. Lõpuks täidetakse regulaaravaldise tüübi poolt spetsifitseeritud tegevused. Kui tüübiks on TOKEN, siis tagastatakse ühtinud lekseem. Kui tüübiks on SPECIAL_TOKEN, siis ühtinud lekseem säilitatakse, et tagastada koos järgmise ühtiva TOKEN tüüpi lekseemiga.

2.5.1 Leksilistes aktsioonides kasutatavad muutujad ja meetodid

Leksilistes aktsioonides kasutatavad muutujad on järgmised:

StringBuffer image (saab lugeda/kirjutada):

> muutuja *image* on *StringBuffer* tüüpi (ei ole sama muutuja, mis on ühtinud lekseemis *Token* väljaks) ning sisaldab kõiki sisendist ühtinud sümboleid alates viimasest SKIP, TOKEN või SPECIAL_TOKEN tüüpi lekseemist. Selle muutuja väärtust on võimalik suvaliselt muuta, ainult nulliga väärtustamist ei tohiks teha, sest antud muutujat kasutab ka genereeritud lekseemihaldur. Kui teha muutujas *image* muutusi, siis see muutus kantakse regulaaravaldise tüübi MORE korral edasi järgnevatesse ühtimistesse. Muutuja *image* sisu ei omistata automaatselt ühtinud lekseemi väljale *image*. Selleks tuleb ilmutatult teha vastav

omistamine TOKEN või SPECIAL_TOKEN tüüpi regulaaravaldise leksilises aktsioonis. Näide:

```
<DEFAULT> MORE: { “a”: S1}

<S1> MORE:
{
“b”                              (1)
{
int i = image.length() - 1;
image.setCharAt(i, image.charAt(i).toUpperCase());
}
:S2                              (2)
}

<S2> TOKEN:
{
“cd” {x = image;} : DEFAULT    (3)
}
```

Antud näites on muutuja *image* väärtused kolmes tähistatud punktis järgmised:

(1) “ab”

(2) “aB”

(3) “aBcd”

int lengthOfMatch (saab ainult lugeda):

hoiab kehtivat ühtinud sümbolite arvu (ei ole kumulatiivne üle MORE-i). Seda muutujat ei tohiks muuta. Eelnevat näidet kasutades oleks *lengthOfMatch* väärtused järgmised:

(1) 1 (“b” pikkus)

(2) 1 (ei muutu, vastavalt leksilistele aktsioonidele)

(3) 2 (“cd” pikkus)

int curLexState (saab ainult lugeda):

kehtiva leksilise oleku indeks, mis on sisuliselt täiarvuline konstant, mille nimeks võivad olla vastavad leksilised olekud. Viimased on genereeritud faili ”...Constants.java”. Seega võib viidata vabalt leksilistele olekutele, teadmata nende indeksite tegelikke väärtusi. Seda muutujat ei tohiks muuta.

inputStream (saab ainult lugeda):

vastavat tüüpi sisendvoog (ASCII_CharStream,

ASCII_UCodeESC_CharStream, UCode_CharStream või UCode_UCodeESC_CharStream, sõltuvalt parameetrite UNICODE_INPUT ja JAVA_UNICODE_ESCAPE väärtustest). Võib kasutada sisendvoo *inputStream* meetodeid, nagu näiteks *getEndLine* ja *getEndColumn*, millega saab teada rea ja veeru numbrilist informatsiooni kehtiva ühtimise kohta. Muutujat *inputStream* ei tohiks modifitseerida.

Token matchedToken (saab lugeda/kirjutada):

võib kasutada ainult aktsioonides, mis on seotud TOKEN või SPECIAL_TOKEN regulaaravaldistega. Väärtuseks on lekseem, mis tagastatakse parserile. Kuna muutuja väärtust võib muuta, siis saab tagastada ka parserile muudetud lekseemi, omistades väljale *matchedToken.image* uue väärtuse. Näiteks võime muuta eelnenud näites viimase regulaaravaldise spetsifikatsiooni järgnevaks:

```
<S2> TOKEN :
{
“cd” { matchedToken.image = image.toString(); } :DEFAULT
}
```

Nüüd on parserile tagasi antava lekseemi väljal *image* väärtus “aBcd”. Kui vastavat omistamist poleks tehtud, siis *image* väli oleks jäänud endiseks “abcd”.

Leksilistes aktsioonides kasutatav ainuke meetod:

void SwitchTo(int):

meetodi väljakutsumine lülitab lekseemihalduri vastavasse olekusse *int*. Seda meetodit võib lisaks leksilistele aktioonidele välja kutsuda ka parseri aktsioonides. Leksiliste olekute muutmisega peab olema ettevaatlik, sest leksilise analüüsimise käigus võidakse eelvaatluse spetsifikatsioonist tingituna palju lekseeme edasi vaadata. Kui kasutada antud meetodit leksilistes aktsioonides, siis peaks see olema viimane täidetav avaldis, sest muidu võib esineda anomaaliaid. Kui olekumuutus on spetsifitseeritud vastava süntaksi abil, siis määratakse eelnevad olekulülitused üle ning antud meetodit on mõttetu kasutada. Võimaluse korral soovitatakse seda meetodit üldse vältida, sest muidu kaotaksime osa JavaCC semantilistest kontrollimistest.

Leksilistel aktsioonidel on juurdepääs hulgale klassitaseme deklaratsioonidele, mis asuvad grammatikafailis produktsioonis *token_manager_decl* (vt jaotis 2.3.7).

2.5.2 Näited

Järgmises näites on spetsifitseeritud reaterminaatoreid lubavad kommentaarid (/* ... */):

```
SKIP:
{
“/*”: WithinComment
}

<WithinComent> SKIP:
“*/”: DEFAULT
}

<WithinComment> MORE:
{
<~[]>
}
```

Järgmises näites on spetsifitseeritud sõneliteraal koos aktsiooniga, mis väljastab sõne pikkuse:

```
TOKEN_MGR_DECLS:
{
int stringSize;
}

MORE:
{
“\”” {stringSize = 0;}: WithisString
}

<WithinString> TOKEN:
{
<STRLIT: “\””> {System.out.println(“Size = “ + stringSize);}:
DEFULT
}

<WithinString> MORE:
{
<~[“\n”, “\r”] {stringSize++;}
}
```

2.5.3 Erilekseemide saatmine parserile

Erilekseemid on sarnased tavaliste lekseemidega, kuid neil on lisaks lubatud sisendfailis ilmuda ükskõik kus (suvalise kahe lekseemi vahel). Erilekseemid spetsifitseeritakse grammatikas võtmesõnaga SPECIAL_TOKEN (tavaline lekseem oli TOKEN), nagu näiteks:

```
SPECIAL_TOKEN:
{
<SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

Nendele regulaaravaldistele pääseb ligi teatud viisil leksika ja grammatika spetsifikatsioonides olevates aktsioonides. See võimaldab antud lekseeme parsimise ajal töödelda, kuigi samal ajal need ei osale parsimises.

Klassil *Token* on lisaväli *Token specialToken*. See väli viitab erilekseemile, mis eelneb vahetult antud lekseemile. Kui antud lekseemile vahetult eelnev lekseem on regulaarne lekseem (mitte erilekseem), siis see väli määratakse nulliks. Regulaarse lekseemi väli *next* viitab järgmisele regulaarsele lekseemile, va <EOF> lekseemi korral, kus *next* on null. Erilekseemi *next* väli viitab erilekseemile, mis koheselt järgneb antud lekseemile. Kui antud lekseemile vahetule järgnev lekseem on regulaarne, siis *next* väli on pandud nulliks. Selgituseks olgu veel järgmine näide. Oletame, et soovitakse väljastada kõiki erilekseeme enne regulaarset lekseemi "t", kuid ainult neid, mis asuvad peale eelnevat regulaarset lekseemi ja enne "t"-d:

```
if (t.specialToken == null) return;
//erilekseeme puudumisel antakse juhtimine väljakutsujale

Token tmp_t = t.specialToken;
//järgmine tsükkel läheb tagasi mööda erilekseemide ahelat,
//kuni leiab esimese erilekseemi, mis paikneb peale eelnevat
//regulaarset lekseemi
while (tmp_t.specialToken != null)
      tmp_t = tmp_t.specialToken;

//järgmine tsükkel läheb nüüd mööda erilekseemide ahelat
//edaspidi ja väljastab need
while (tmp_t != null) {
      System.out.println(tmp_t.image);
      tmp_t = tmp_t.next;
}
```

2.6 Veatöötlus

Java programmi töötlusel tekkivatele eriolukordadele vastavad *Throwable* alamklassid ja need jagatakse kahte kategooriasse: vead (*error*) ja erindid (*exception*). Vead on eriolukorrad, millest pole võimalik paraneda: *ThreadDeath* või *OutOfMemoryError*. Vead on kõik *Error* klassi alamklassid. Erindid on defineeritud tavaliselt klassi *Exception* alamklassidena. Need erindid on tüüpiliselt käsitletavad ja deklareeritakse meetodi kirjelduses *throws* klauslis.

JavaCC-s on defineeritud kaks erindit: *ParseException* ja *TokenMgrError*. Kui lekseemihaldur avastab mingi probleemi, siis seatakse erind *TokenMgrError*, mille püüdmisel väljastatakse tavaliselt järgmise kujuga teade:

```
Lexical Error ...
```

Kui aga parser avastab mingi probleemi, siis seatakse erind *ParseException*, mille püüdmisel väljastatakse tavaliselt järgmise kujuga teade:

```
Encountered ... Was expecting one of  ...
```

Erinditöötluses ei väljastata teateid kunagi ilmutatult, vaid vastav informatsioon on paigutatud erindi objekti, mis erindiseade korral luuakse. Täpsemalt saab vaadata vastavate erindiobjektide sisu JavaCC poolt genereeritud failidest “ParseException.java” ja “TokenMgrError.java”. Kui seatud erindit kinni ei püüta, siis käitub Java virtuaalmasin standardselt ja väljastab pinu jälje (*stack trace*) ning samuti erindi objektile rakendatud meetodi *toString* tagastusväärtuse. Kui aga erind ise kinni püütakse, siis tuleb ka erindi teade ise väljastada.

Erind *TokenMgrError* on klassi *Error* alamklass, kuigi erind *ParseException* on klassi *Exception* alamklass. Põhjuseks on see, et lekseemihaldurilt ei eeldata kunagi erindi seadmist – peab tähelepanelikult defineerima lekseemide deklaratsioonid, et kõik juhud saaksid kaetud. Süntaks mitteterminaalidele vastavate meetodite muude erindite spetsifitseerimiseks on identne Java *throws* klausliga:

```
void VariableDeclaration() throws ParseException, IOException:
{...}
{
...
}
```

Antud näites on *VariableDeclaration* defineeritud seadma lisaks erindile *ParseException* veel erindit *IOException*.

2.6.1 Vigade raporteerimine

Üheks vigade standardse raporteerimise modifitseerimise võimaluseks on muuta "ParseException.java" faili. Tüüpiliselt tuleks muuta meetodit *getMessage*, mille abil saab luua endale meelepärase vigade teatamise. Genereeritud parseril on olemas ka meetod *generateParseException*, mille abil saab vajadusel ise luua *ParseException* objekti. See objekt sisaldab parseri kõiki proovitud valikuid pärast viimast edukalt ühtinud lekseemi.

2.6.2 Vigade taastamine

JavaCC võimaldab kahesugust veatöötlust: pinnapealset (*shallow recovery*) ja sügavat (*deep recovery*). Esimesel juhul töödeldakse veaolukorda valikpunktis, kui seal polnud ühtki sobivat valikut. Teisel juhul töödeldakse veaolukorda, mis tekkis valikpunktis, kui valik oli tehtud ning selle valiku parsimisel tekkis mingi viga.

2.6.2.1 Pinnapealne veataastus

Olgu järgmine näide:

```
void Stm():
{}
{
IfStm()
|
WhileStm()
}
```

Oletame, et mitteterminaal *IfStm* algab reserveeritud sõnaga "if" ja *WhileStm* algab

võtmesõnaga “while”. Nüüd tahetakse näiteks taastamise käigus jätta vahele kõik, kuni järgmise semikoolonini, kui *IfStm* või *WhileStm* ei ühti järgmise sisendist tuleva lekseemiga (LOOKAHEAD olgu 1). Seega kirjutatakse järgmiselt:

```
void Stm():
{}
{
IfStm()
|
WhileStm()
|
error_skipto(SEMICOLON)
}
```

Kuid esmalt tuleb defineerida ka mitteterminaal *error_skipto*. Selleks võib kasutada Java-koodi produktsiooni:

```
JAVACODE
void error_skipto(int kind) {
      //genereerime erindi objekti
      ParseException e = generateParseException();
      //väljastame veateade
      System.out.println(e.toString());
      Token t;
      //tsükkel kasutab kõik lekseemid kuni lekseemini “kind”
      do {
            t = getNextToken();
      } while (t.kind != kind);
}
```

2.6.2.2 Sügav veataastus

Kasutame sama näidet, mis oli eelmises jaotises. Nüüd soovitakse samuti viga taastada, kuid lisaks isegi siis, kui see esineb sügavamal parsimisel. Näiteks oletame, et järgmine lekseem oli “while”, mistõttu valiti mitteterminaal *WhileStm*. Oletame, et selle parsimise käigus tekkis mingi viga, näiteks oli sisend “while (foo { stm;};”, st üks sulgev sulg oli puudu. Pinnapealne veataastus sel juhul ei töötaks. Vaja läheb sügavamat veataastust, mida saab konstrueerida katsendidirektiivi abil:

```
void Stm():
{}
{
try {
(
IfStm()
|
WhileStm()
)
}
catch (ParseException e) {
error_skipto(SEMICOLON);
}
}
```

Seega, kui nüüd esineb mõni viga mitteterminaali *IfStm* või *WhileStm* parsimisel, siis püünis (*catch*) võtab täitmisjärje üle. Püüniseid võib olla suvaline arv koos mittekohustusliku epiloogiga (*finally*) lõpus (nagu Javas ikka). Püünisesse kirjutatakse Java kood, mitte JavaCC laiendus. Näiteks eelmine näide tuleks ümber teha järgmiselt:

```
void Stm():
{}
{
try {
(
IfStm()
|
WhileStm()
)
}
catch (ParseException e) {
      System.out.println(e.toString());
      Token t;
      do {
            t = getNextToken();
      } while (t.kind != SEMICOLON);
}
}
```

Püünisesse oleks mõistlik panna suhteliselt vähe Java koodi. Selleks võib eraldi meetodid defineerida, mida siis püünise blokis hiljem rakendada. Samuti võib tähele panna, et püünises pole vaja ise kutsuda välja meetodit *generateParseException*, sest see juba edastatakse püünisesse parameetrina (*ParseException e*).

2.7 JavaCC käsurea süntaks

JavaCC käsurea süntaks on järgmine:

```
javacc [parameetrite seaded] <sisendfail>
```

Kui käsureale anda lihtsalt korraldus “javacc”, siis väljastatakse kasutusjuhend. Parameetrite seadete järjendis `[parameetrite seaded]`on seaded üksteisest eraldatud tühikutega ja järgivad ühte järgmistest vormidest:

–parameetri_nimi=väärtus (näiteks: –STATIC=false)
–parameetri_nimi:value (näiteks: –STATIC:false)
–parameetri_nimi (ekvivalentne –parameetri_nimi=true, näiteks: –STATIC)
–NOparameetri_nimi (ekvivalentne –parameetri_nimi=false, näiteks: –NOSTATIC)

Parameetri seaded ei ole tõstutundlikud, st. “–NOSTATIC” asemel võib kirjutada ka “–nOsTaTiC”. Parameetrite väärtused peavad vastama parameetritele, olles kas *int*, *boolean* või *String* tüüpi (LOOKAHEAD, STATIC, OUTPUT_DIRECTORY, ...). Näide:

```
javacc -STATIC=false -LOOKAHEAD:2 -debug_parser Minu_grammatika.jj
```

2.8 JJDoc

JJDoc produtseerib JavaCC parseri spetsifikatsioonist dokumentatsiooni BNF grammatika jaoks. JJDoc’il on kolm režiimi, mis määratakse käsurea parameetritega.

TEXT – parameetri vaikeväärtus on väär (*false*). Kui parameetri väärtus on tõene, siis genereeritakse BNF kirjeldus lihtteksti formaadis. Vastasel korral genereeritakse aga hüperlingitud HTML dokument.

ONE_TABLE – parameetri vaikeväärtus on tõene (*true*). Kui parameetri väärtus on

tõene, siis dokument genereeritakse ühe HTML tabelina. Vastasel korral genereeritakse iga grammatika produktsiooni jaoks oma tabel.

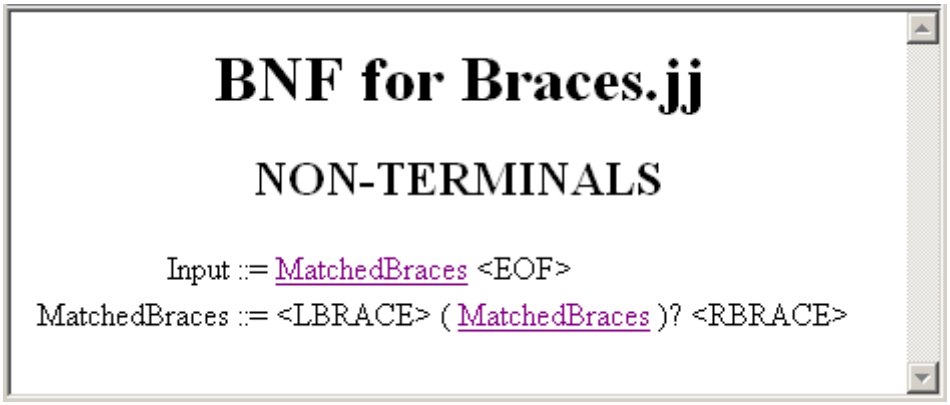
OUTPUT_FILE – selle parameetriga määratakse väljundfaili nimi. Vaikimisi moodustatakse see sisendfaili nimest, milles laiend "jj" asendatakse vastavalt sufiksitega "html" või "txt".

Kui TEXT on tõene või ONE_TABLE on väär, siis vahetult produktsioonidele eelnevad kommentaarid JavaCC koodis lisatakse genereeritavasse dokumenti.

Järgnevalt on illustreeritud JJDoc'i rakendamist jaotises 2.2 esinevale grammatikafailile "Braces.jj":

```
jjdoc -ONE_TABLE:true Braces.jj
```

Tulemusfail "Braces.html" näeb veebilehitsejas välja järgmiselt:



3 JavaCC kasutamine skeem-modelleerimises

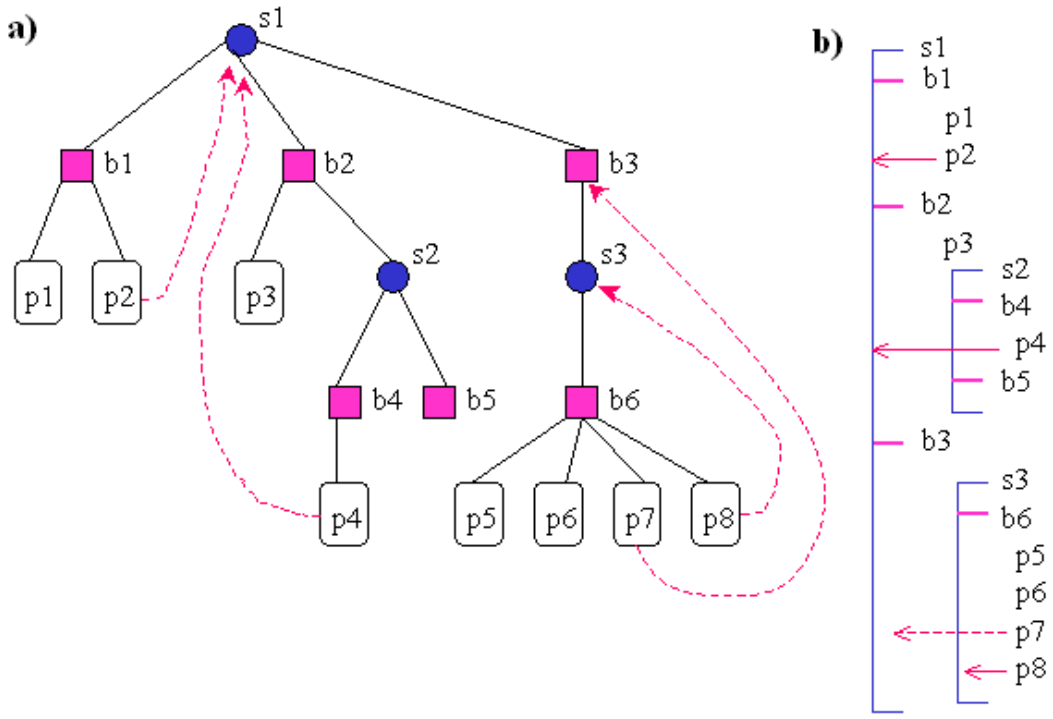
Selles peatükis tutvustatakse skeem-modelleerimise olemust ning näidatakse, kuidas on võimalik kasutada eelpool tutvustatud parserigeneraatorit JavaCC skeem-modelleerimisel.

3.1 Skeemtöötluse idee

Tänapäeval puutuvad vastava eriala inimesed tihti kokku mitmesuguste arvutitekstidega, erijuhul programmidega. Neid programme või arvutitekste soovitakse mingil viisil töödelda, kas siis redigeerida, lihtsalt lugeda vms. Tekstid võivad olla aga väga erinevalt süntaktiliselt kirjeldatud ning nende spetsiifiline töötlemine nõuaks ka vastavaid erilisi töötlusvahendeid ja lähenemisi. Skeem-modelleerimise eesmärk ongi kujutada süntaktiliselt erinevaid arvutitekste ühtse skeem-mudeli baasil. Modelleeritud arvutitekstide töötlemine põhineks siis vaid ühel kontseptsioonil [SKM]. See tähendab, et olemuselt erinevate arvutitekstide käsitlemisel saab kasutada ühte ja seda sama töötlussüsteemi.

3.1.1 Üldine skeem-mudel

Skeem-mudeli [SKM] aluseks on puu, milles saab noolte abil näidata ka teatavaid tagasisidestusi. Seega skeem-modelleerimisel esitatakse suvaline arvutitekst vastava puustruktuurilise skeemtekstina (*sketchy text*), milles teksti lihtsamad osad esinevad puu tippude tekst-atribuutidena. Mudelis defineeritakse need atomaarsed tekstiosad tavateksti üldistusena, lubades ka hüpersümbolite (piltide ja omakorda skeemtekstide) kasutamist reasümbolite rollis. Lähteteksti keel (baaskeel) ja modelleeritud teksti (ekraani)vaade antakse skeemtippude vastavate lisa-atribuutidena. Järgnevalt näide skeempuust (a) ja vastavast skeemtekstist (b):



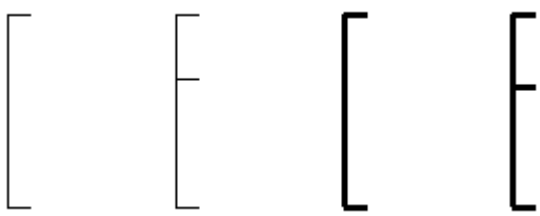
Skeemtippudele vastavad lisa-atribuudid on:

skeem	●	kommentaar tüüp päis ikoniseeritus vaade baaskeel
haru	■	kommentaar tüüp päis ikoniseeritus vaade baaskeel
primitiiv	▢	tüüp primitiivtekst
nool	↰	tüüp primitiivtekst lõputase

Järgnevalt on antud skeemteksti üldine süntaks BNF-na:

skeemtekst :: = skeem
kavand ::= skeem | haru
skeem ::= skeemi_keha kavandi_atribuudid
skeemi_keha ::= (haru)+
haru ::= haru_keha kavandi_atribuudid
haru_keha ::= (liige)*
liige :: = skeem | primitiivliige
primitiivliige ::= primitiiv | nool
primitiiv ::= tüüp primitiivtekst
nool ::= primitiiv lõputase
lõputase ::= **non-negative_integer**
tüüp ::= **non-negative_integer**
primitiivtekst ::= (rida)* kommentaar
kommentaar ::= rida
rida ::= (rea_element)+
rea_element ::= sümbol | hüpersümbol
sümbol ::= **character font color**
hüpersümbol ::= pilt | hüperelement
pilt ::= ikoniseeritus **image**
hüperelement ::= skeemtekst
kavandi_atribuudid ::= tüüp kommentaar päis ikoniseeritus **view base_language**
päis ::= (primitiiv_päis)*
primitiiv_päis ::= primitiiv
ikoniseeritus ::= **true** | **false**

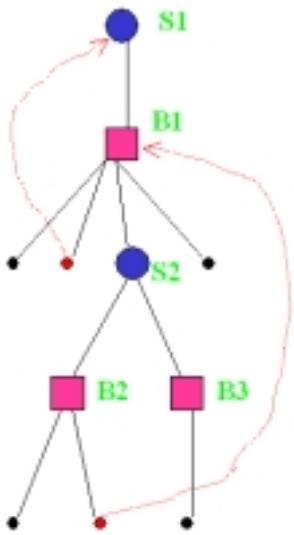
Antud töös enamkasutatavad skeemteksti elemendid on vastavalt päiseta ja päisega lihtskeem ja moodulskeem (esitatuna nn skeemvaates):



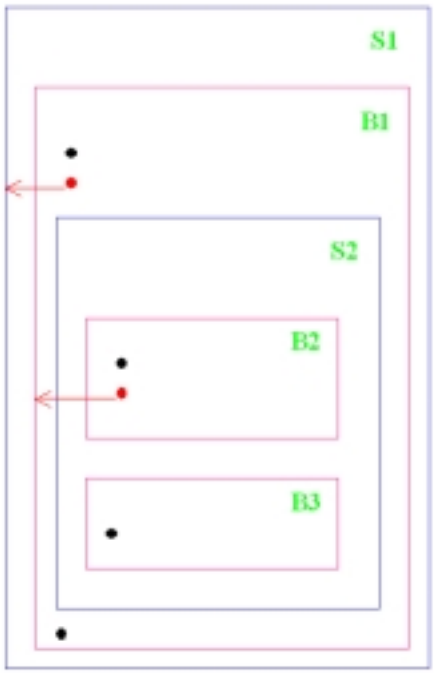
Skeemtekstide käsitlemiseks on loodud (veel arenduses olev) vastav Java-põhine süsteem Amadeus-fRED [fRED], mida tutvustatakse põgusalt järgmises jaotises.

3.1.2 Skeemtoimeti Amadeus-fRED

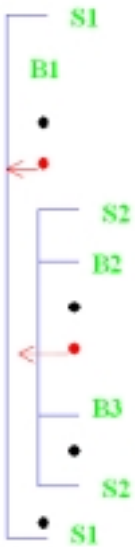
Amadeus-fRED on Javal baseeruv mitmevaateline skeemteksti WYSIWYM toimeti, kus saab vastavaid elemente lisada, eemaldada, redigeerida, skeemi osi ikoniseerida jne. Toimeti võimaldab toetatud baaskeelte (Java, LATEX, HTML, XML, ...) skeemistamist ning taas tekstualiseerimist (*textualize*) jms. Iga skeemiga on seotud vaade olemasolevate (toetatavate) vaadete hulgast. Vaadet kasutatakse skeemi kuvamiseks, rakendades vaate meetodeid ekraani planeerimiseks ja skeemi joonistamiseks. On olemas skeemvaade (E-algoritmi [A&A] analoog), kastvaade (b*oxes*), puuvaade (*simple tree-view*) jmt. Skeemi jooksva vaate saab hõlpsasti asendada mingi teise vaatega. Järgnevalt näide kahest vaatest:



Skeempuu



Kastvaade



Skeemvaade

3.1.3 Arvutiteksti näide

Käesolevas jaotises tutvustatakse ühte spetsiifilist arvutitekstide keelt. Viimast nimetatakse keeleks EKFG, sest valdkonnaks on eesti keele formaalne grammatika. Tegemist on süntaktilise analüsaatori [EKFG] väljundiga, mis saadakse mingi konkreetse eestikeelse teksti töötlemisel. Näiteks teksti:

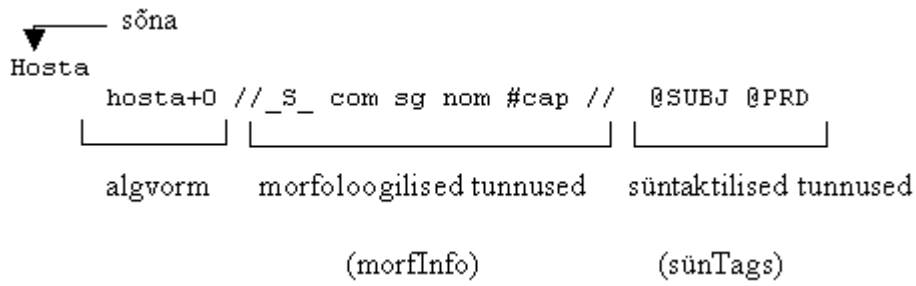
```
Hostad ehk funkiad (Hosta) on 50...80 cm kõrgused taimed.
```

korral väljastab süntaktiline analüsaator nn süntaktiliselt analüüsitud teksti:

```
Hostad
    hosta+d //_S_ com pl nom #cap //  **CLB @SUBJ @NN>
ehk
    ehk+0 //_J_ crd //  @J
funkiad
    funkiad+0 //_S_ com sg nom //  @SUBJ @PRD @NN>
$(
    $( //_Z_ Opr //
Hosta
   hosta+0 //_S_ com sg nom #cap //  @SUBJ @PRD
$)
    $) //_Z_ Cpr //
on
    ole+0 //_V_ aux indic pres ps3 pl ps af #FinV #Intr // @+FCV
50...80
    50...80+0 //_N_ card #? digit //  @ADVL @NN>
cm
    cm+0 //_Y_ nominal #? //  @ADVL @NN>
kõrgused
    kõrgus+d //_S_ com pl nom //  @SUBJ @NN>
taimed
    taim+d //_S_ com pl nom //  @PRD
$.
    $. //_Z_ Fst //
```

Sellist laadi väljundite hulk moodustabki keele EKFG. Põhimõtteliselt on EKFG-tekst sisendiks järgmistele keeletöötluse protsessoritele, näiteks semantilisele analüsaatorile. Kuid enne järgmist töötlusetappi on vaja EKFG-tekst inimese poolt üle vaadata ja korrigeerida, sest süntaktiline analüsaator ei ole täiuslik. Inimene peab teksti kontrollima, parandama vead ja lahendama võimalikud mitmesused ning küsitavused. Loomulikult saab korrigeerimist teha mõne tavalise tekstitoimetiga. Kuid mugavam oleks töötada eritoimetiga, mis võimaldaks näiteks sisuliste konstruktsioonide tervikuna käsitlemist. Antud juhul oleks sellisteks struktuurseteks konstruktsioonideks muuhulgas

osalause (EKFG-tekst koosneb osalausetest), sõna analüüs (osalause koosneb sõnade analüüsidest), sõna ise (sõna analüüs algab sõnaga), sõna algvorm jmt. Näiteks sõna analüüs:



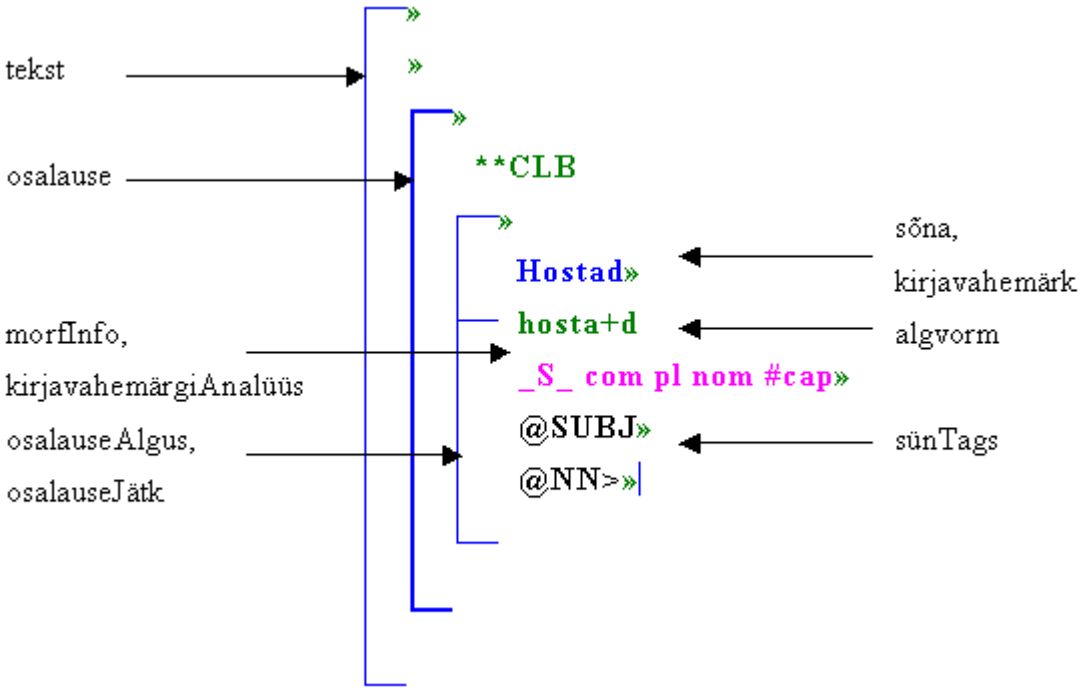
Täpsemalt kirjeldab EKFG-tekstide struktuuri järgnev BNF grammatika:

tekst ::= (osalause)+
osalause ::= osalauseAlgus (osalauseJätk)*
osalauseAlgus ::= (sõna | <DOL> kirjavahemärk) (tühikud)? reavahetus
(morfAnalüüs ("**CLB" | "**CLB-C")
(tühikud sünTags (tühikud)?)? reavahetus)+
osalauseJätk ::= (sõna | <DOL> kirjavahemärk) (tühikud)? reavahetus
(morfAnalüüs (sünTags (tühikud)?)? reavahetus)+
morfAnalüüs ::= tühikud (algvorm tühikud morfInfo
| <HASH> <HASH> <HASH> <HASH>
| kirjavahemärgiAnalüüs) (tühikud)?
algvorm ::= sõna ((<EQ> | <LL> | <PLUS>) sõna)*
morfInfo ::= <SLASH> <SLASH> morfTag (tühikud morfTag *
(tühikud)? <SLASH> <SLASH>
morfTag ::= (<LL> morfSona <LL> | <HASH> morfSona
| morfSona)
morfSona ::= (<BASECHAR> | <DIGIT> | <MINUS> | <QUE>)+
kirjavahemärk ::= (<EXC> | <LPA> | <RPA> | <COMMA> | <MINUS> |
<MINUSMINUS> | <DOT> | <DOTDOT> |
<DOTDOTDOT> | <SLASH> | <COLON> |
<SEMICOLON> | <QUE> | <QUO> | <AT> | <LSQ> |
<RSQ> | <LAN> | <RAN> | "</s>" | "<p>")
kirjavahemärgiAnalüüs ::= <DOL> kirjavahemärk tühikud morfInfo

sünTags ::=	<SYNTAG> (tühikud <SYNTAG>)*
tühikud ::=	(<SPACE> | <TAB>)+
reavahetus ::=	(<NEWLINE> | <RETURN>)+
sõna ::=	(<BASECHAR> | <DIGIT> | <ELSEFORCHAR> | <MINUS> | <MINUSMINUS> | <DOT> | <DOTDOT> | <DOTDOTDOT> | <AT> | <COLON> | <SLASH> | <EXC> | <COMMA> | <SEMICOLON> | <QUE> | <SYNTAG>)+

kus noolsulgude vahel on vastavad lekseemid, esitatuna juba JavaCC stiilis.

Taoliste tekstide struktuuri arvestava, kuid muus osas tavafunktsionaalsusega eritoimeti loomine oleks väga kulukas. Skeem-modelleerimise rakendamisel langeb aga üldse ära vajadus eritoimeti loomiseks (“nullist alates”). Piisab sellest, kui fikseerida kujutus keelest EKFG skeem-tekstide keelde (nn skeem-mudel) ning sellele mudelile vastavalt koostada ja lisada süsteemi Amadeus-fRED uus baaskeele klass. Saadud täiendatud Amadeus-fRED omab muuhulgas ka EKFG-tekstide eritoimeti funktsionaalsust. Järgnevates jaotistes kirjeldatakse skeem-modelleerimise protsessi lähemalt. Siinkohal märgime veel, et keele EKFG skeem-mudel on järgmine:



Skeem-mudel on välja töötatud koostöös TÜ keeletehnoloogia õppetooliga. Sealjuures oli kaalumisel ka teisi mudeli variante (näiteks sõna paiknemine mitte skeemi päises vaid selle kommentaaris jmt).

Antud suvalise EKFG-teksti saab skeemtoimetis teisendada skeemtekstiks, mida on mugav üle vaadata ja korrigeerida. Tulemuse saab tagasi tavatekstiks tekstualiseerimise teel. Teisendused käivituvad lihtsalt menüüvalikute kaudu.

3.2 Skeem-mudel ja skeemistamine

3.2.1 Skeem-mudeli mõiste

Olgu *A* mingi arvutitekstide keel ja *M* kujutus hulgast *A* skeemtekstide hulka *ST**:

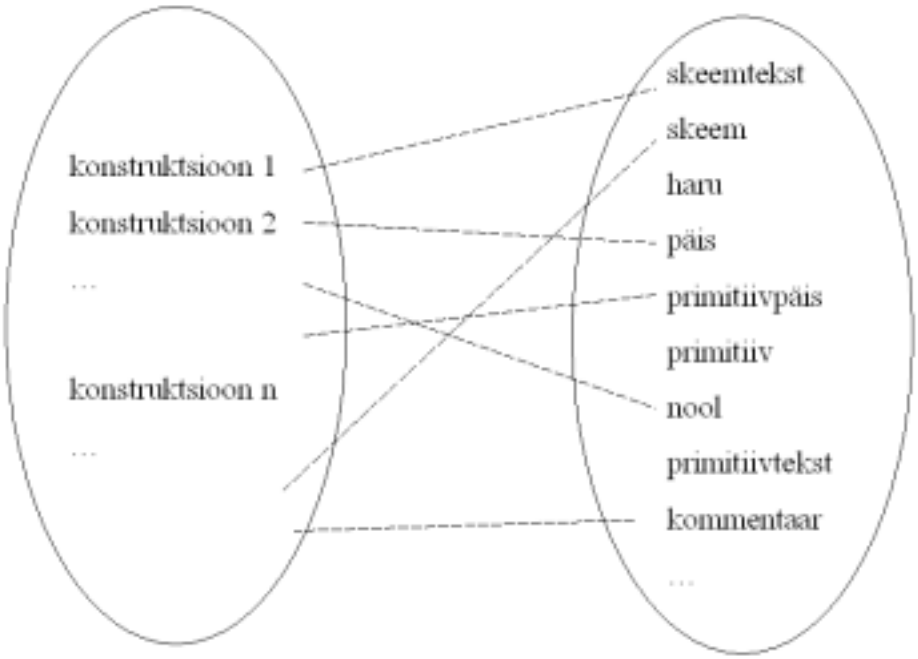
$$M : A \rightarrow ST^*$$

Kujutust *M* nimetatakse keele *A* skeem-mudeliks, kui on defineeritud kujutise *M* pöördkujutus M^{-1}.

Sageli piisab skeem-mudeli esitamisest vastavustena:

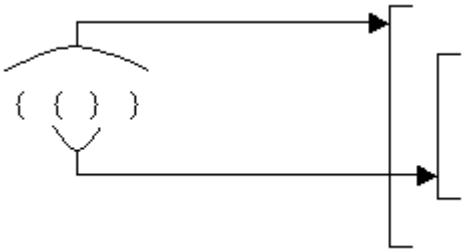
{keele *A* konstruktsioon} ↔ {skeemteksti element}

ehk piltlikult:



Tõsi küll, selline esitus on mõnevõrra mitteformaalne ega võimalda skeem-mudeli automaatset töötlemist.

Väga lihtsa näitena esitatakse jaotises 2.2 kirjeldatud “looksulgude keele” skeem-mudel:



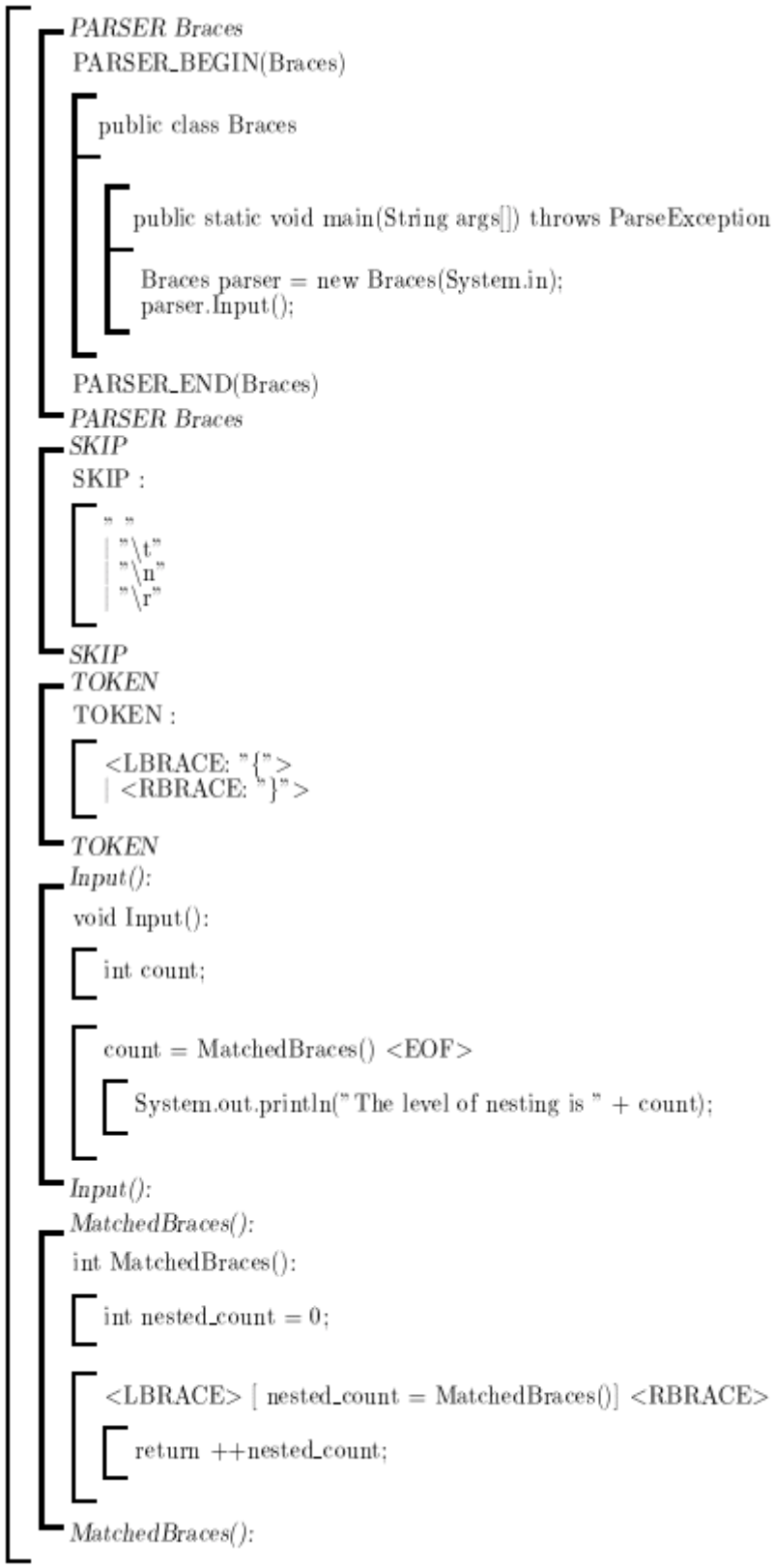
Seega mudel koosneb ühest vastavusest: { } ↔ lihtskeem.

Küllaltki keerukaks mitteformaalse mudeli näiteks on Java-programmide skeem-mudel [SKM]. Viimast saab muuseas rakendada ka JavaCC grammatikafaili korral. Näiteks võib kõik looksulgudega määratud blokid esitada lihtskeemidena, suvalised grupeeringud aga päiseta moodulskeemidena. Parseri klassi võib kirjutada nagu skeem-Java klassi. Taoliselt esitatud JavaCC grammatikafaili korral saab tekstualiseerimiseks (pöördkujutuseks) vahetult kasutada Java-programmide tekstualiseerimise meetodit. Näide JavaCC skeemistatud grammatikafailist leidub järgmisel leheküljel. Taolist JavaCC grammatikafaili skeem-vormi kasutatakse ka edaspidi.

3.2.2 Skeemistamine

Skeem-mudel (sõltumata selle ranguse astmest) on aluseks uue baaskeele lisamisel süsteemi Amadeus-fRED. Eeskätt tuleb lahendada kaks ülesannet, milleks on vastavalt skeemistamine ja tekstualiseerimine (ning vastavate meetodite lisamine uue baaskeele klassi). Skeemistamise meetod realiseerib kujutuse M, tekstualiseerimise meetod aga selle pöördkujutuse M^{-1}. Käesolevas töös vaadeldakse just esimest, st skeemistamise ülesannet ning skeemistamise meetodi automaatsema konstrueerimise võimalusi.

Varem toimus uue baaskeele lisamine süsteemi Amadeus nö “käsitsi”. Skeemtoimetisse uue baaskeele integreerimisel tuli muuhulgas üksikasjalikult kirjutada ka skeemistamismeetod - teisendusoperatsioonide kogum, mis lähtekeele vastavatele elementidele seadis vastavusse sobivad skeemteksti elemendid (koos vastavate


PARSER Braces
PARSER_BEGIN(Braces)
public class Braces
public static void main(String args[]) throws ParseException
Braces parser = new Braces(System.in);
parser.Input();
PARSER_END(Braces)
PARSER Braces
SKIP
SKIP :
" "
| "\t"
| "\n"
| "\r"
SKIP
TOKEN
TOKEN :
<LBRACE: "{">
| <RBRACE: "}">
TOKEN
Input():
void Input():
int count;
count = MatchedBraces() <EOF>
System.out.println("The level of nesting is " + count);
Input():
MatchedBraces():
int MatchedBraces():
int nested_count = 0;
<LBRACE> [nested_count = MatchedBraces()] <RBRACE>
return ++nested_count;
MatchedBraces():

atribuutidega). Teisenduste kirjutamine oli küllaltki aeganõudev ja tülikas, sest iga baaskeele jaoks tuli, lisaks muule, algusest lõpuni konstrueerida ka teksti parsiv osa. Seega kerkis vajadus leida meetodeid ja vahendeid, kuidas uue baaskeele integreerimist lihtsustada ja automatiseerida.

On ilme, et iga baaskeelt peab iseloomustama tema grammatika. Siin tulebki mängu JavaCC, mis suudab teatud reeglistiku abil konstrueeritud grammatikast genereerida vajaliku parseri. Seega taandub esialgne ülesanne baaskeele grammatika teisendamiseks JavaCC-le sobivale kujule ning vastavate skeemteksti elementide konstrueerimise määramisele.

Konkreetse baaskeele klass süsteemis Amadeus-fRED pärineb abstraktsest klassist *BaseLanguage* ja peab (muuhulgas) realiseerima viimases kirjeldatud abstraktse skeemistamismeetodi *sketchify*. Käesolevale väljatöötlusele tuginedes (st alates JavaCC kasutuselevõtust) hakkavad uute lisatavate baaskeelte skeemistamismeetodid erinema põhiliselt vaid selle poolest, et igaüks neist rakendab oma baaskeele-spetsiifilist parserit. Viimane on genereeritud loomulikult JavaCC abil ja seda nimetatakse skeemistusparseriks ehk skeemparseriks.

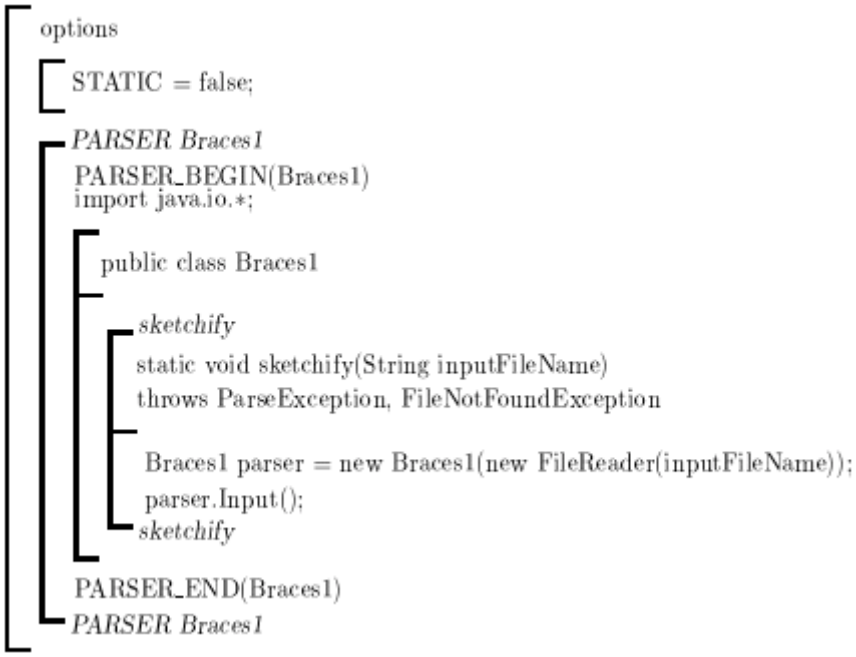
Skeemistusparser on saadud JavaCC sellisest grammatikafaililist, milles aktsioonid realiseerivad vastava skeem-mudeli. Seega aktsioonide ülesandeks on konkreetse skeemteksti konstrueerimine parsitavast arvutitekstist lähtudes.

3.3 Näited konkreetsete skeemistusparserite loomisest

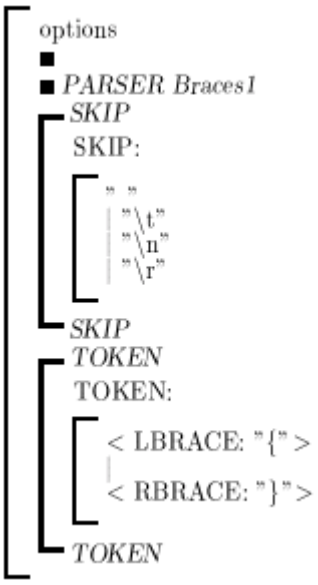
Selles jaotises kirjeldatakse skeemistusparseri koostamist kahe näitekeele abil, milleks on eelnevalt tutvustatud looksulgude keel (Braces1) ning EKFG.

3.3.1 Keele Braces1 skeemistusparseri loomine

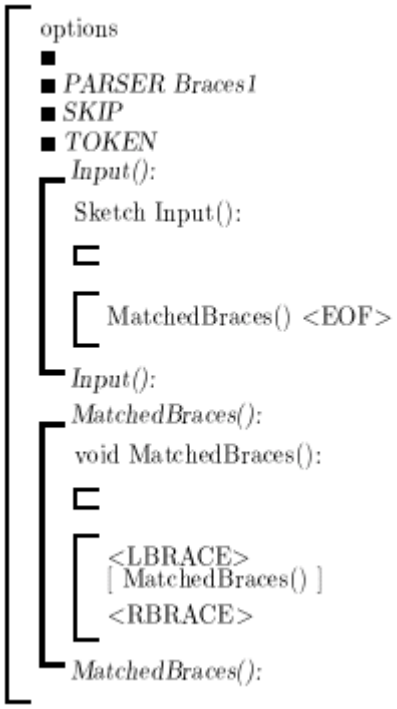
Esmalt loome parseri klassi, mille meetod *sketchify* loeks sisendi etteantud failist:



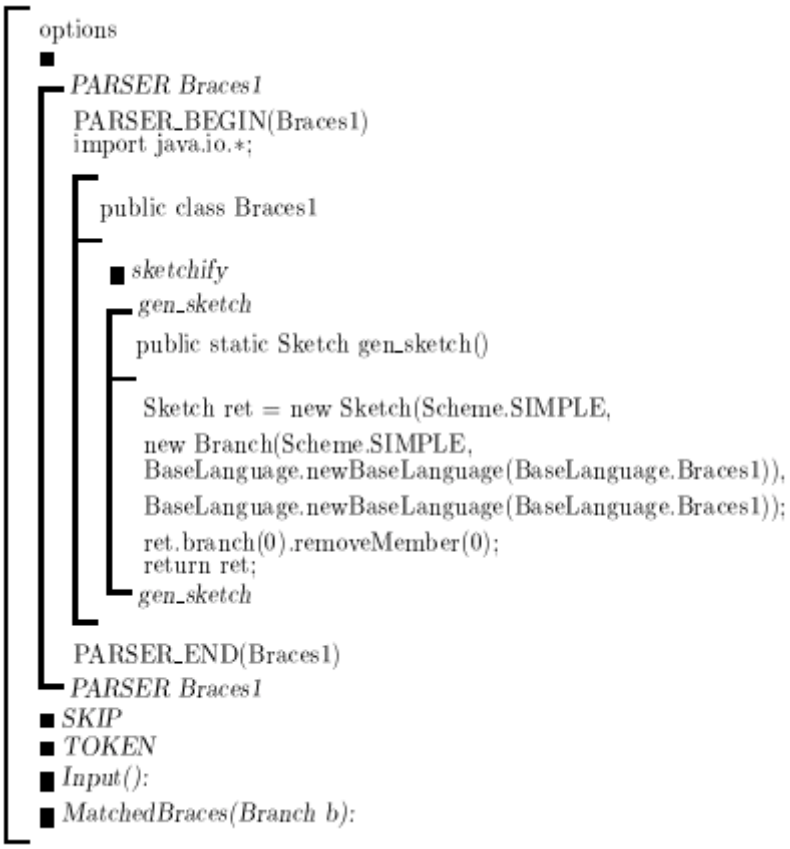
Nüüd defineerime juba tuttavad lekseemid (ka tühilekseemid), kasutades selleks SKIP ja TOKEN tüüpi regulaaravaldise produktsiooni:



Järgmisena lisame vajalikud looksulge parsivad BNF produktsioonid *Input* ja *MatchedBraces*:



Paneme tähele, et praeguseks pole veel ühtegi aktsiooni lisatud. Vastavalt jaotises 3.2.1 esitatud looksulgude keele skeem-mudelile konstrueerime abimeetodi *gen_sketch*, mis tagastaks tühja lihtskeemi (Amadeus-fRED klasstüübid *Sketch* ja *Branch* esindavad vastavalt mõisteid skeem ja haru):



Lõpuks lisame BNF produktsioonidesse ka vastavad aktsioonid, mis konstrueerivad soovitud skeemteksti:

Produktsiooni *Input* tagastusväärtus on nüüd *Sketch*, Java-blokis konstrueeritakse aga meetodiga *gen_sketch* tagastatav lihtskeem (*s_ret*). Produktsioonile *MatchedBraces* on

lisatud formaalne parameeter *Branch b* ning Java-blokis konstrueeritakse lihtskeem (*s*), mis lisatakse sisemise *MatchedBraces* ühtimisel ülemskeemi haru (*b*) järjekordseks liikmeks. Parseri meetodi *sketchify* tagastusväärtuseks on nüüd viit klassi *Sketch* isendile – konstrueeritud skeempuule. Seega on looksulgude keele skeemistusparseri grammatika valmis. Meetodi *sketchify* rakendamisel legaalsele sisendtekstile konstrueeritaksegi vastav skeempuu ja tagastatakse viit sellele.

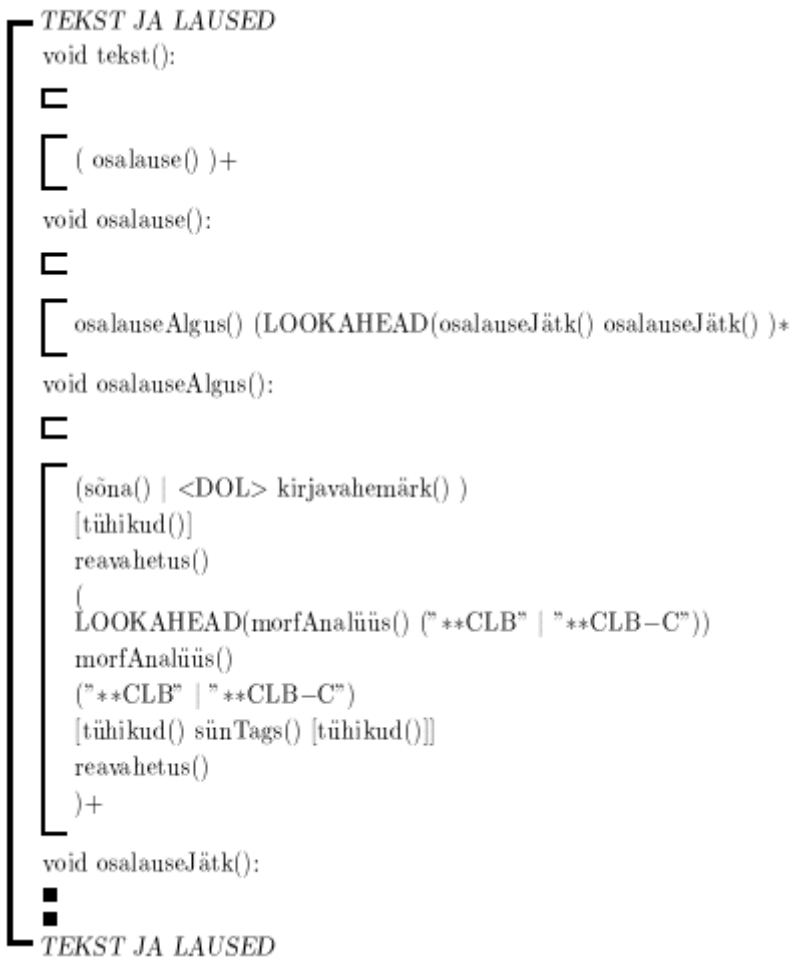
3.3.2 Keele EKFG skeemistusparseri loomine

Ka keele EKFG korral püüame esmalt koostada aktsioonivaba JavaCC grammatika, luues parseri klassi analoogselt eelmise jaotise näitega. Paneme kirja kõikvõimalikud lekseemid, mis sisendis on lubatud:

```
LEKSEEMIDE KIRJELDUSED
TOKEN:
  //Pole vaja eraldi % & ' > < *
  ■ SPACE ... MINUSMINUS
  A-Z a-z
    //A-Z a-z
    | < BASECHAR: ["\u0041"-"\u005A", "\u0061"-"\u007A", "\u00C0"-"\u00FF"] >
    | < DIGIT: ["\u0030"-"\u0039"] >
    | < BEGINNING: ["\u0000"-"\u0008", "\u000B", "\u000C", "\u000E"-"\u001f"] >
    | < ELSEFORCHAR: [ "\u0000"-"\u00FF" ] >
  A-Z a-z
  | < SYNTAG:
    <AT> (
    (<PLUS> | <MINUS>) ("FMV" | "FCV")
    | "NEG"
    | "SUBJ"
    | "OBJ"
    | "PRD"
    | "ADVL"
    | ("<" ("AN" | "AD" | "PN" | "NN" | "VN" | "INF_N" | "P" | "Q"))
    | (("AN" | "AD" | "PN" | "NN" | "VN" | "INF_N" | "P" | "Q") ">")
    | "J"
    | "I"
    )
  >
LEKSEEMIDE KIRJELDUSED
```

Lisaks kirjamärkidele (“,”, “(”, “)“, “@”, …), mis skeemil on ikoniseeritud, on defineeritud põhisümbolid (<BASECHAR>), numbrid (<DIGIT>), erisümbolid (<BEGINNING>), kõik ülejäänud sümbolid (<ELSEFORCHAR>) ning süntaktilised märgendid <SYNTAG>. Paneme tähele, et lekseemi <ELSEFORCHAR> defineerimise eesmärgiks on lekseemihalduri töövõime tagamine kõigi antud sisendsümbolite korral. Seega on parsimise seisukohalt võimalik saada vaid erindit *ParseException*.

Järgnevalt koostame aktsioonivaba grammatikafaili. Viimane on täielikult esitatud lisas. Siinkohal esitame vaid mõned produktsioonid: *tekst*, *osalause*, *osalauseAlgus*. Juhime tähelepanu sellele, et näiteks produktsioonis *osalause* on kasutatud süntaktilist eelvaatlust, sest *osalauseAlgus* ja *osalauseJätk* on süntaksilt küllaltki sarnased (vt jaotis 3.1.3), mis omakorda põhjustab parsimise valikpunktis segasust.



Kui vajalik aktsioonideta parser on koostatud ja ka testitud, siis tuleb vastava skeem-mudeli (vt 3.1.3, 3.4.2) baasil genereerida vajalikke skeemteksti konstruktsioone loovad meetodid ning nende abil luua vajalikud aktsioonid. See etapp on praeguseks juba vähesel määral automatiseeritud. Seetõttu vaatleme aktsioonide lisamist eraldi järgmises jaotises, kus näidatakse, kuidas vastav mudel otse skeemtoimetis grammatikafaili panna, et pärast tekstualiseerimist tekiksid skeemielemente genereerivad meetodid õigesse kohta.

3.4 Skeemistusparseri automaatne loomine

3.4.1 Idee

Nagu eespool selgitatud, on uue baaskeele *B* lisamisel esmasteks ülesanneteks keele *B* grammatika *G* ja skeem-mudeli *M* koostamine. Nende ülesannete lahendamisele järgneb skeemistusparseri loomine, mis koosneb järgmistest sammudest:

- vastavalt grammatikale *G* koostada ja siluda JavaCC grammatikafail;
- tulemusele lisada aktsioonid, mis realiseerivad mudeli *M*, st tegevused skeemteksti konstrueerimiseks.

Esimese sammu automatiseerimine on ilmselt väga keeruline (kui mitte võimatu), eeskätt just lekseemikirjelduste ja eelvaatluste (LOOKAHEAD) korraldamise osas. Heal juhul võib baaskeele jaoks leida vastava JavaCC grammatikafaili vastavast repositooriumist [JavaCC GR]. Vastasel korral tuleb see peatükis 2 kirjeldatud eeskirjade kohaselt koostada. Kui keele grammatika on juba kirjas BNF kujul, siis selle üleviimine “käsitsi” JavaCC XBNF kujule ei tohiks valmistada probleemi. Saadud parserit saab hõlpsasti testida ja siluda enne skeemteksti konstrueerimisaktsioonide lisamist, et veenduda selle korrektsuses.

Küll aga on lootust automatiseerida teist sammu – konstrueerimisaktsioonide lisamist. Nimelt on konstrueeritavate objektide (skeemkonstruktsioonide) mõiste hästi defineeritud. See viib mõttele, et juhul, kui õnnestub leida sobiv formalism ka skeem-

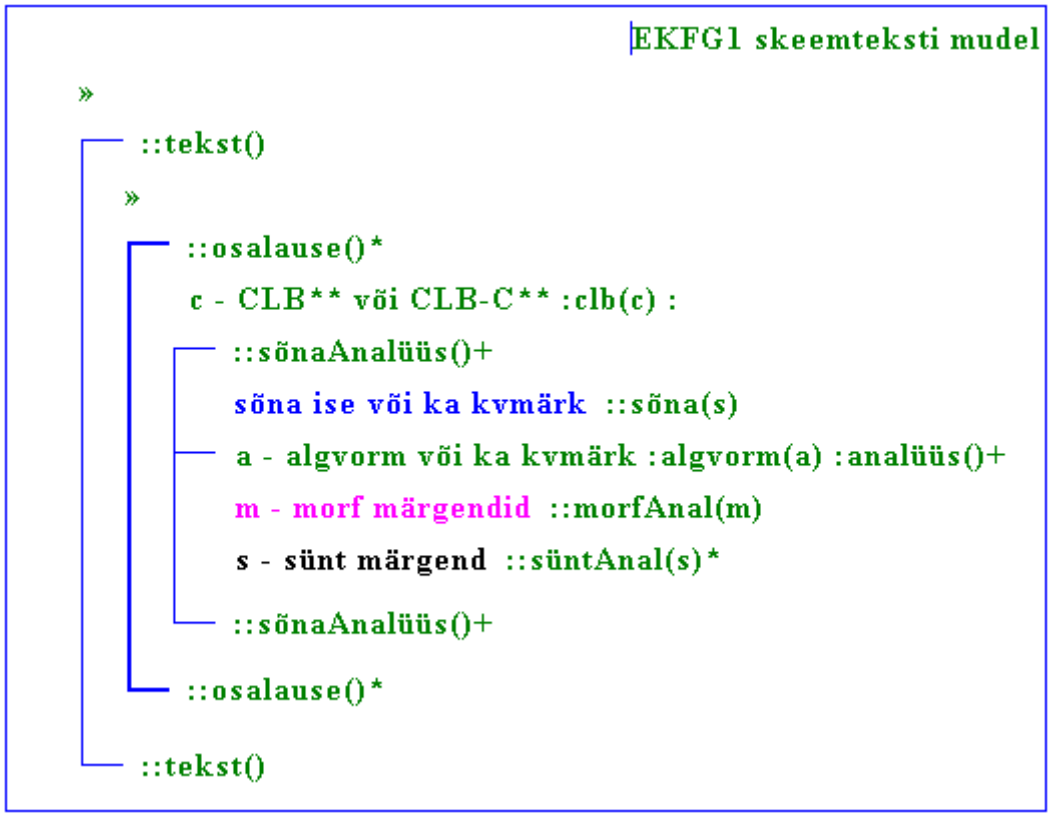
mudeli esitamiseks, siis saab välja töötada automaatse skeemistusparseri generaatori ASKP:

3.4.2 Lahenduse alged

Käsitletud näidete baasil paneme tähele, et skeemistusparseris

- on BNF produktsioonid üldiselt tagastusväärtuseta;
- produktsiooni Java-blokis genereeritakse vastav tühi konstruktsioon;
- produktsiooni laiendustes lisatakse (järk-järgult) leitavaid konstruktsioone.

Praeguseks on automatiseeritud mudeli pealt (tühjade) konstruktsioonide genereerimismeetodite koostamine. Aluseks on skeem-mudel skeemtekstina, mille eesmärgiks ongi ära näidata uue baaskeele grammatika produktsioonidele vastavad skeemteksti elemendid. Hiljem genereeritakse selle mudeli baasil vastavad skeemteksti tühje konstruktsioone genereerivad Java meetodid, mis tuleb integreerida antud grammatikasse. Mudel koostatakse vastavat reeglistikku järgides skeemtoimetiga. Enne reeglite täpsustamist illustreeriv näide keele EKFG mudelist (deikoniseerituna):



Sisulist mudelit ümbritseb lihtskeem, mille baaskeel peab olema määramata (*None*) ning vaade kastvaade. Sisuline mudel ise aga peab olema määratud õige baaskeelega (vastav baaskeele klass peab olema loodud, vt jaotis 3.2.2), kuid vaatest see enam ei sõltu, sest mainitud genereeritavad meetodid konstrueerivad skeemteksti komponente alati skeemvaates.

Iga mudelis esitatud konstruktsiooni kommentaari lõpuosas määratakse vajadusel ära seda osa genereeriva meetodi signatuur (meetodi nimi ja mittekohustuslik argumendi nimi). Vastav kommentaari lõpp peab olema järgmise erikujuga:

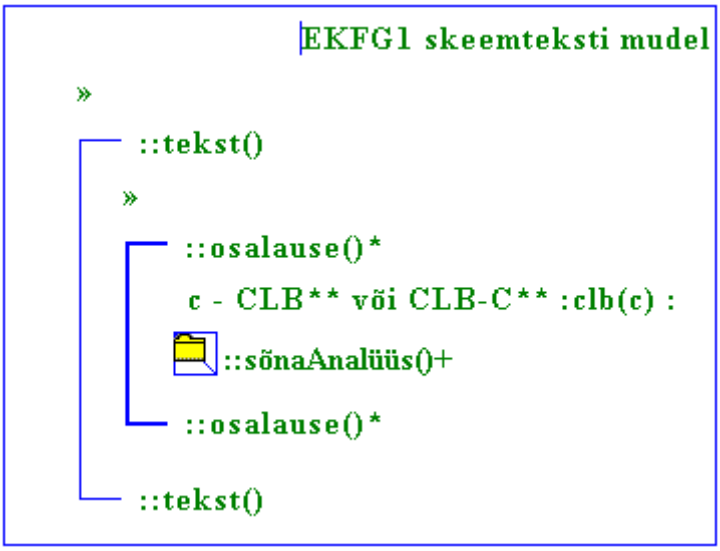
> ":" [konstruktsiooni_kommentaar "(" [argument] ")"] ":"
> [konstruktsioon "(" [argument]")"]

kus esimesele koolonile võib järgneda kommentaari meetodi signatuur ning teisele koolonile konstruktsiooni meetodi signatuur. Kommentaari algusosas ei ole koolonid lubatud. Kui argument on määratud, siis loodavasse meetodisse läheb see *String* tüüpi argumendina. Näiteks, kui soovitakse ainult teatud konstruktsiooni kommentaari

genereerivat meetodit (nimega “gen_skeemi_kommentaar”), siis tuleks kirjutada mudelis kommentaari kohale:

```
See on skeemi kommentaar :skeemi_kommentaar():
```

Kusjuures kommentaari algusosa “See on skeemi kommentaar” ei oma tähtsust. Skeem-mudeli tekstualiseerimise tulemuseks on vastavaid konstruktsioone genereerivad meetodid, mille nimed saadakse meetodite signatuuridest prefiksi “gen_” lisamisel. Mudelit illustreerivaid sümboleid “+”, “*” siinjuures ignoreeritakse. Tuleb rõhutada, et mudeli elemendid peavad olema enne tekstualiseerimist sobivas vormis, st peab olema määratud baaskeel, skeemide ja primitiivide tüübid ning ka ikoniseerimine. Näiteks sobivalt ikoniseeritud keele EKFG mudel:

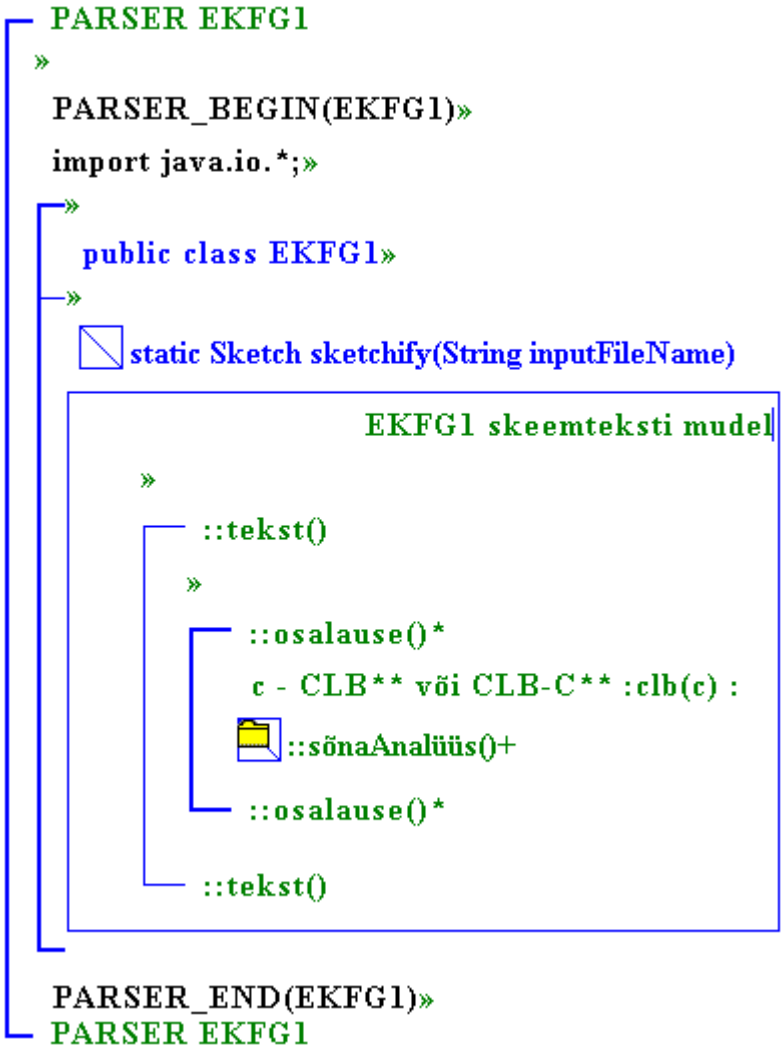


Pärast mudeli tekstualiseerimist tekib muuhulgas näiteks järgmine meetod:

```
public static Sketch gen_sõnaAnalüüs() {
      Sketch tulem = new Sketch(Scheme.SIMPLE,
            new Branch(Scheme.SIMPLE,
            BaseLanguage.newBaseLanguage(BaseLanguage.EKFG)),
            BaseLanguage.newBaseLanguage(BaseLanguage.EKFG));
      tulem.icon = true;
      tulem.branch(0).removeMember(0);
      return tulem;
}
```

See meetod tagastab tühja lihtharuga lihtskeemi, mis on ikoniseeritud ning mille baaskeeleks on EKFG.

Vastav mudel paigutatakse parseri klassimeetodi(te) järele:

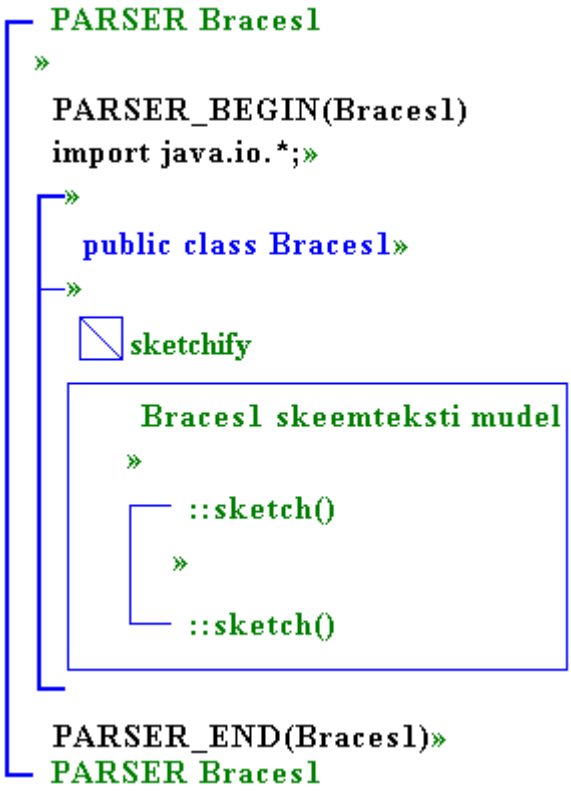


Mudeli asemele tekivad pärast kogu grammatika tekstualiseerimist skeemteksti elemente moodustavad meetodid parseri staatiliste meetoditena, mida saab edukalt kasutada aktsioonide koostamisel. Keele EKFG skeemistusparser on esitatud lisas.

Skeemistusparseris sisalduva mudeli tekstualiseerib meetod *textualize* klassis *BaseLanguageNone*.

Praeguseks on seega skeemistusparseri koostaja vabastatud konstruktsioonide genereerimismeetodite kirjutamisest. Eeldusel, et on välja töötatud uue baaskeele skeem-mudel skeemteksti kujul, on automaatselt loodav ka vastavate meetodite komplekt. Skeemistusparseri koostajal pole vaja enam teada suurt hulka Amadeus-fRED tehnilisi detaile.

Pöördudes tagasi keele Braces1 näite juurde (vt jaotis 3.3.1), näeme, et ära jääks meetodi *gen_sketch* "käsitsi" koostamine. Piisab, kui selle koha peal esitada Braces1 skeem-mudel:



Kokkuvõte

Eriotstarbeliste arvutikeelte toimetite loomine on enamasti kulukas. Alternatiivseks lahenduseks on skeem-modelleerimine, mille realiseerib skeemtoimeti Amadeus-fRED. Põhiliseks võtmeküsimuseks on sealjuures uue baaskeele integreerimine süsteemi. St skeemistamise ja tekstualiseerimise meetodite efektiivne realiseerimine.

Antud töös käsitletud skeemistamisülesande lahenduseks saab kasutada parserigeneraatori JavaCC abi. Baaskeele grammatika vastava kirjeldamise ja sellesse keele skeem-mudeli integreerimisega on võimalik luua skeemistusparser, mis suudab konstrueerida lähteteksti baasil skeempuu. Seda on aga juba võimalik Amadeus-fRED standardsete vahenditega töödelda. Autoril on pakutud lahendus skeemistusparseri aktsioonides rakendatavate skeemielemente konstrueerivate meetodite automaatseks genereerimiseks.

Täisautomaatset skeemistusparserit on väga raske luua. Edasiseks võiks kaaluda vastava formalismi täiendamist, mille alusel oleks võimalik puhta baaskeele grammatikale täiesti automaatselt skeemteksti konstrueerivaid aktsioone lisada.

The Parser Generator JavaCC and Sketchy Modeling

Kristo Heero

M.Sc. Thesis

Abstract

This paper deals with sketchy modeling system Amadeus-fRED. The main problem is to make effective the integration of new base languages in to this system. There are two main subtasks: sketchifying and textualizing. The aim of this thesis is to give methodical solution and overview, how to adjust the first goal. For this purpose, the Java parser generator JavaCC is considered. Once we can describe the new base language grammar in terms of JavaCC and add necessary actions, based on the sketchy model, it is possible to generate the needed parser. The latter can be used for building sketchy trees from input texts. The question is, how to make this process as automatic as possible. A preliminary concept for particle automation is elaborated and implemented. The further idea is to work out better mechanisms to describe relation between base language units and sketchy text constructs.

Kasutatud kirjandus

[A&A]

J. Kiho, “Algoritmid ja andmestruktuurid”, Tartu, 1997, 124 lk.

[EKFG]

T. Roosmaa, M. Koit, K. Muischnek, K. Müürisep, T. Puolakainen, H. Uibo, “Eesti keele formaalne grammatika”, Tartu, 2001, 158 lk.

[fRED]

J. Kiho, http://www.cs.ut.ee/~kiho/fRED/.

[JavaCC]

WebGain, http://www.webgain.com/products/java_cc/.

[JavaCC doc]

WebGain, http://www.webgain.com/products/java_cc/documentation.html.

[JavaCC GR]

JavaCC Grammar Repository, http://cobase-www.cs.ucla.edu/pub/javacc/.

[SKM]

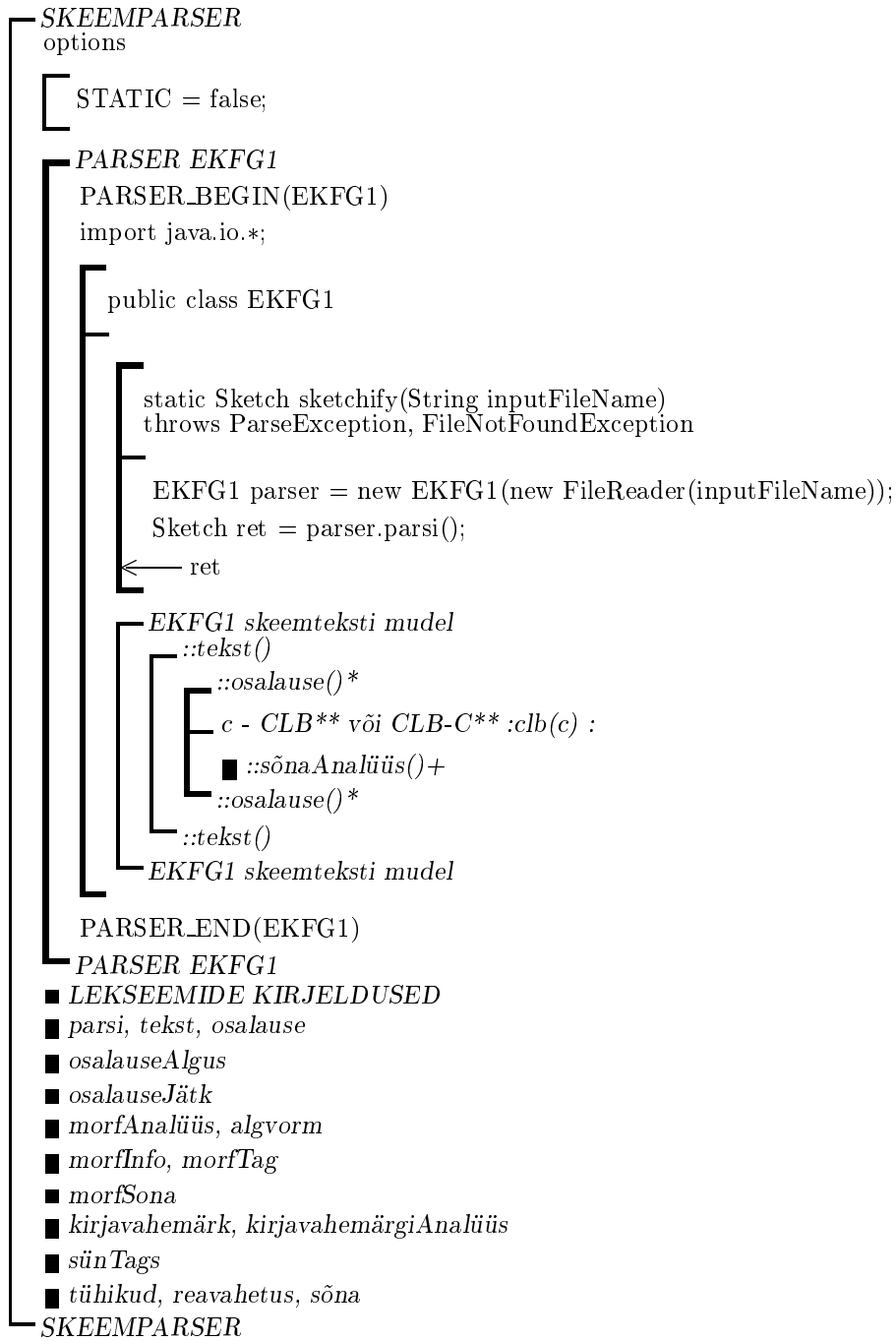
J. Kiho, “SKM. Sketchy Modeling of Computer Texts”, UT, Research Report, Jan. 2000, 64 lk.

[VJL]

J. Kiho, “Väike Java leksikon”, Tartu, 1997, 36 lk.

Lisa

Keele EKFG skeemistusparser


SKEEMPARSER
options
STATIC = false;
PARSER EKFG1
PARSER_BEGIN(EKFG1)
import java.io.*;
public class EKFG1
static Sketch sketchify(String inputFileName)
throws ParseException, FileNotFoundException
EKFG1 parser = new EKFG1(new FileReader(inputFileName));
Sketch ret = parser.parsi();
ret
EKFG1 skeemteksti mudel
::tekst()
::osalause()*
c - CLB** või CLB-C** :clb(c) :
::sõnaAnalüüs()+
::osalause()*
::tekst()
EKFG1 skeemteksti mudel
PARSER_END(EKFG1)
PARSER EKFG1
LEKSEEMIDE KIRJELDUSED
parsi, tekst, osalause
osalauseAlgus
osalauseJätk
morfAnalüüs, algvorm
morfInfo, morfTag
morfSona
kirjavahemärk, kirjavahemärgiAnalüüs
sünTags
tühikud, reavahetus, sõna
SKEEMPARSER

LEKSEEMIDE KIRJELDUSED

TOKEN:

- - - *Pole vaja eraldi % & ' > < **

■ *SPACE ... MINUSMINUS*

A-Z a-z

| < BASECHAR: ["\u0041"−"\u005A", "\u0061"−"\u007A", "\u00C0"−"\u00FF"] >

| < DIGIT: ["\u0030"−"\u0039"] >

| < BEGINNING: ["\u0000"−"\u0008", "\u000B", "\u000C", "\u000E"−"\u001f"] >

| < ELSEFORCHAR: ["\u0000"−"\u00FF"] >

A-Z a-z

| < SYNTAG:

<AT> (
(<PLUS> | <MINUS>) ("FMV" | "FCV")
| "NEG"
| "SUBJ"
| "OBJ"
| "PRD"
| "ADVL"
| ("<" ("AN" | "AD" | "PN" | "NN" | "VN" | "INF_N" | "P" | "Q"))
| (("AN" | "AD" | "PN" | "NN" | "VN" | "INF_N" | "P" | "Q") ">")
| "J"
| "I"
)

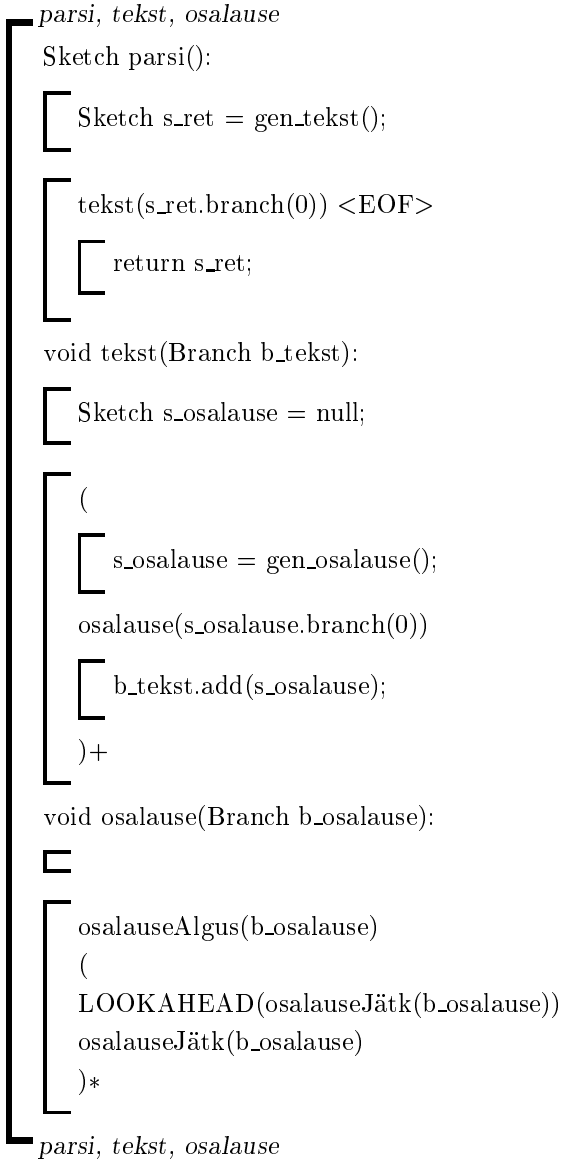
>

LEKSEEMIDE KIRJELDUSED

SPACE ... MINUSMINUS

```
< SPACE: " " >
| < TAB: "\t">
| < NEWLINE: "\n">
| < RETURN: "\r">
| < EXC: "!" > //"\u0021"
| < QUO: "\""> //"\u0022"
| < HASH: "#" > //"\u0023"
| < DOL: "$" > //"\u0024"
| < LPA: "(" > //"\u0028"
| < RPA: ")" > //"\u0029"
| < PLUS: "+" > //"\u002B"
| < COMMA: "," > //"\u002C"
| < MINUS: "−" > //"\u002D"
| < DOT: "." > //"\u002E"
| < SLASH: "/" > //"\u002F"
| < COLON: ":" > //"\u003A"
| < SEMICOLON:";" > //"\u003B"
| < EQ: "=" > //"\u003D"
| < QUE: "?" > //"\u003F"
| < AT: "@" > //"\u0040"
| < LSQ: "[" > //"\u005B"
| < RSQ: "]" > //"\u005D"
| < LL: "_" > //"\u005F"
| < LAN: "" > //"\u00AB"
| < RAN: "" > //"\u00BB"
| < DOTDOT: ".."">
| < DOTDOTDOT: "..."">
| < MINUSMINUS: "−−" >
```

SPACE ... MINUSMINUS


parsi, tekst, osalause
Sketch parsi():
Sketch s_ret = gen_tekst();
tekst(s_ret.branch(0)) <EOF>
return s_ret;
void tekst(Branch b_tekst):
Sketch s_osalause = null;
(
s_osalause = gen_osalause();
osalause(s_osalause.branch(0))
b_tekst.add(s_osalause);
)+
void osalause(Branch b_osalause):
osalauseAlgus(b_osalause)
(
LOOKAHEAD(osalauseJätk(b_osalause))
osalauseJätk(b_osalause)
)*
parsi, tekst, osalause

osalauseAlgus

```
void osalauseAlgus(Branch b):
{
    Sketch s_sona = gen_sõnaAnalüüs();
    Branch b_sona = null;
    String so;
    Token t, t2;
}
{
    (
    (so = sõna() | t2 = <DOL> so = kirjavahemärk()
    {
        so = t2.image + so;
    }
    )
    {
        s_sona.add(gen_sõna(so));
    }
    )
    [tühikud()]
    reavahetus()
    (
    LOOKAHEAD(morfAnalüüs(b_sona) ("**CLB" | "**CLB-C"))
    {
        b_sona = gen_analüüs();
    }
    morfAnalüüs(b_sona)
    (t = "**CLB" | t = "**CLB-C")
    {
        b.comment = gen_clb(t.image);
    }
    [tühikud() sünTags(b_sona) [tühikud()]]
    reavahetus()
    {
        {s_sona.add(b_sona);}
    }
    )+
    {
        b.add(s_sona);
    }
}
```

osalauseAlgus

osalauseJätk

void osalauseJätk(Branch b):

{
Sketch s_sona = gen_sõnaAnalüüs();
Branch b_sona = null;
String so;
Token t;
}

(
(so = sõna() | t = <DOL> so = kirjavahemärk()
{ so = t.image + so; }
)
{ s_sona.add(gen_sõna(so)); }
)
[tühikud()]
reavahetus()
(
LOOKAHEAD(morfAnalüüs(b_sona) [sünTags(b_sona) [tühikud()]] reavahetus())
{ b_sona = gen_analüüs(); }
morfAnalüüs(b_sona)
[sünTags(b_sona) [tühikud()]]
reavahetus()
{ s_sona.add(b_sona); }
)+
{ b.add(s_sona); }

osalauseJätk

morfAnalüüs, algvorm

```
void morfAnalüüs(Branch b):
    Token t;

    tühikud()
    (
    algvorm(b) tühikud() morfInfo(b)
    |
    t = <HASH> <HASH> <HASH> <HASH>
        b.add(gen_morfAnal(t.image + t.image + t.image + t.image));
    |
    kirjavahemärgiAnalüüs(b)
    )
    [tühikud()]
void algvorm(Branch b):
    String av, so;
    Token t;

    av = sõna()
    (
    (t = <EQ> | t = <LL> | t = <PLUS>) so = sõna()
        av += t.image + so;
    )*
        b.comment = gen_algvorm(av);
```

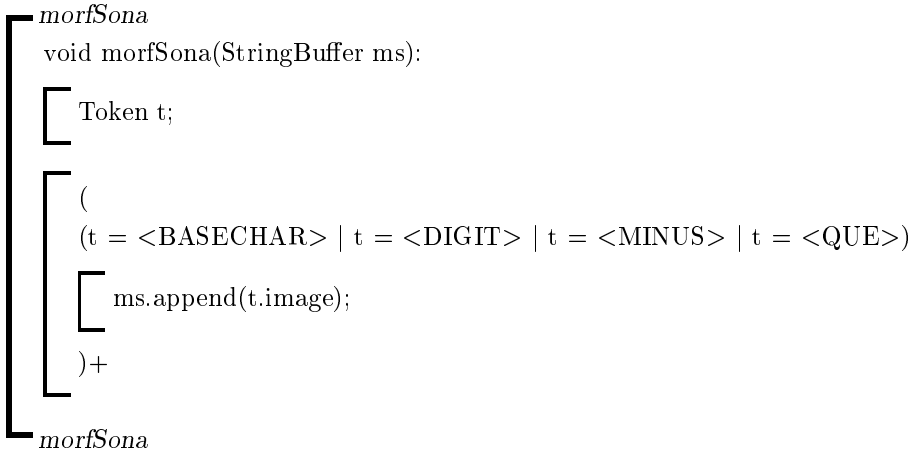
morfAnalüüs, algvorm

morfInfo, morfTag

```
void morfInfo(Branch b):
{
    StringBuffer mts = new StringBuffer();
}
{
    <SLASH> <SLASH>
    morfTag(mts)
    (
    LOOKAHEAD(tühikud() morfTag(mts)) tühikud() morfTag(mts)
    )*
    [tühikud()]
    <SLASH> <SLASH>
    {
        b.add(gen_morfAnal(mts.toString().trim()));
    }
}

void morfTag(StringBuffer mts):
{
    String so;
    StringBuffer mt = new StringBuffer();
    Token t1, t2;
}
{
    (
    t1 = <LL>
    {
        mt.append(t1.image);
    }
    morfSona(mt)
    t2 = <LL>
    {
        mt.append(t1.image);
    }
    | t1 = <HASH>
    {
        mt.append(t1.image);
    }
    morfSona(mt)
    | morfSona(mt)
    )
    {
        mts.append(" " + mt.toString());
    }
}
```

morfInfo, morfTag


morfSona
void morfSona(StringBuffer ms):
Token t;
(
(t = <BASECHAR> | t = <DIGIT> | t = <MINUS> | t = <QUE>)
ms.append(t.image);
)+
morfSona

kirjavahemärk, kirjavahemärgiAnalüüs

String kirjavahemärk():

Token t;

(
t = <EXC> | t = <LPA>
| t = <RPA> | t = <COMMA>
| t = <MINUS> | t = <MINUSMINUS>
| t = <DOT> | t = <DOTDOT>
| t = <DOTDOTDOT> | t = <SLASH>
| t = <COLON> | t = <SEMICOLON>
| t = <QUE> | t = <QUO>
| t = <AT> | t = <LSQ>
| t = <RSQ> | t = <LAN>
| t = <RAN> | t = "</s>"
| t = "<p>"
)

return t.image;

void kirjavahemärgiAnalüüs(Branch b):

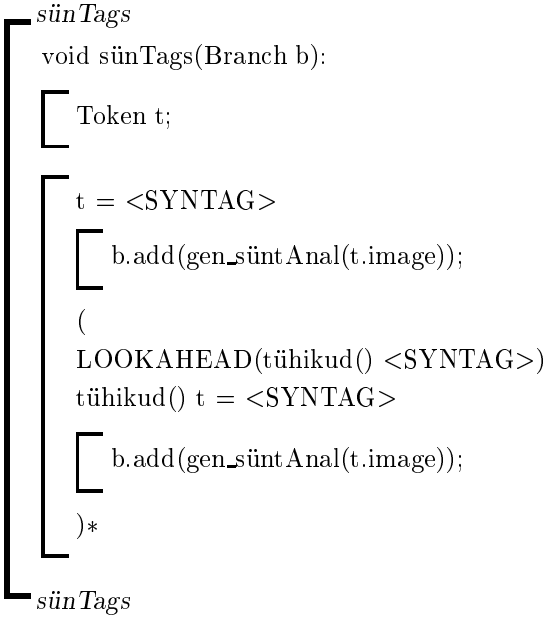
String so;
Token t;

t = <DOL> so = kirjavahemärk()

b.comment = gen_algvorm(t.image + so);

tühikud() morfInfo(b)

kirjavahemärk, kirjavahemärgiAnalüüs


sünTags
void sünTags(Branch b):
Token t;
t = <SYNTAG>
b.add(gen_süntAnal(t.image));
(
LOOKAHEAD(tühikud() <SYNTAG>)
tühikud() t = <SYNTAG>
b.add(gen_süntAnal(t.image));
)*
sünTags

tühikud, reavahetus, sõna

```
void tühikud():
{
}
{
  (<SPACE> | <TAB>)+
}
void reavahetus():
{
}
{
  (<NEWLINE> | <RETURN>)+
}
String sõna():
{
  Token t;
  String so = "";
}
{
  (
   - - - Sõnas ei tohi olla <EQ>, <LL>, <PLUS>, et oleks algvormi kirjeldus 1:1
   - - - Välja jätame ka <LAN>, <RAN>, <QUO> – igasugu jutumärgid
   - - - Sulud igasugused <LSQ>, <RSQ>, <LPA>, <RPA>
   - - - Ka <HASH> ja <DOL> on väljas, kuna #### on erimärk ja
   - - - $<kirjavahemärk> on süntaksiks
  (
  t = <BASECHAR> | t = <DIGIT> | t = <ELSEFORCHAR>
  | t = <MINUS> | t = <MINUSMINUS> | t = <DOT>
  | t = <DOTDOT> | t = <DOTDOTDOT> | t = <AT>
  | t = <COLON> | t = <SLASH> | t = <EXC>
  | t = <COMMA> | t = <SEMICOLON> | t = <QUE>
  | t = <SYNTAG>
  )
  {
    so += t.image;
  }
  )+
  {
    return so;
  }
}
```

tühikud, reavahetus, sõna