TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma

HARRI LUOMA

Tulkin toteutus ohjelmoinnin perusopetuksen tarpeisiin

Diplomityö

Tarkastajat: professori Hannu-Matti Järvinen
ja tutkija Essi Lahtinen

Tarkastajat ja aihe hyväksytty
tietotekniikan osaston
osastoneuvoston kokouksessa 16.4.2007

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
HARRI LUOMA: Tulkin toteutus ohjelmoinnin perusopetuksen tarpeisiin
Diplomityö, 56 sivua, 43 liitesivua
Lokakuu 2007
Pääaine: Ohjelmistotekniikka
Tarkastajat: Hannu-Matti Järvinen, Essi Lahtinen
Avainsanat: visualisointi, tulkki, ohjelmoinnin perusopetus, VIP, CLIP, C++, C--

Tietokoneohjelmien tekeminen ja ohjelmoinnissa alkuun pääseminen on hankalaa. Aloittelevien ohjelmoijien pitää usein kirjoittaa ohjelmakoodiinsa sellaisia rivejä, mitä he eivät ymmärrä. Tällaisia rivejä ovat esimerkiksi kirjastojen mukaan ottamiset. Ratkaisuna tähän on tulkkipohjainen ohjelmoinnin opetus. Ohjelmointikielen tulkki osaa suorittaa ohjelmakoodia suoraan komentoriviltä, eikä kirjastoja tai kääntämistä tarvita.

Tässä diplomityössä toteutettiin ohjelmointikielen tulkki C++:n osajoukolle. Tulkin tukema kieli nimettiin C--:ksi. Se sisältää kaikki C++:n perusrakenteet, mutta ei esimerkiksi luokkia eikä nimiavaruuksia. Tulkissa painotettiin selkeitä virheilmoituksia ja niiden lokalisointia. Tarkoituksena oli tehdä tulkki, jota käyttäen aloittelevat ohjelmoijat voivat päästä helposti alkuun ohjelmoinnissa.

Tulkki on kehitetty Antti Virtasen tekemästä VIP 1.0-järjestelmän tulkista, joten sen nimeksi annettiin VIP 2.0. VIP on javasovelmana toteutettu graafinen ympäristö, joka visualisoi ohjelmakoodin suoritusta ajoaikaisesti tulkin avulla. VIP 2.0:aan on tulossa uusi visualisoitu käyttöliittymä, mutta tämän diplomityön puitteissa tulkille toteutettiin vain komentorivikäyttöliittymä. Komentorivikäyttöliittymän nimeksi annettiin CLIP (Command Line InterPreter). Valmistuttuaan VIP 2.0 on tarkoitus ottaa käyttöön Tampereen teknillisellä yliopistolla ohjelmoinnin peruskursseilla. Opiskelijat voivat harjoitella ohjelmointitaitojaan VIPin avulla ja myös palauttaa osan ohjelmointitehtävistään VIPin kautta. VIP 2.0 tullaan julkaisemaan vapaana ohjelmistona GPL-lisenssillä. Tulkki toteutettiin Java 1.5:lla. Apuna käytettiin SableCC-jäsentäjägeneraattoria.

Tulkkia ei vielä ole otettu käyttöön, mutta alustava palaute on ollut positiivista. Oletuksena on, että kun tulkki otetaan käyttöön, opiskelijat oppivat C++:n perusrakenteet nopeammin kuin ennen.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
HARRI LUOMA : Implementation of Interpreter for Programming Education
Master of Science Thesis, 56 pages, 43 appendix pages
October 2007
Major: Software Engineering
Examiners: Hannu-Matti Järvinen, Essi Lahtinen
Keywords: visualization, interpreter, programming education, VIP, CLIP, C++, C--

Implementing computer programs and getting started with programming is often very hard. Novice programmers have to write program code parts which they do not understand. For example, they have to include libraries. The solution is to use interpreter-based programming education. An interpreter can execute the program code on command line without including any libraries or compiling the program.

In this thesis an interpreter for a subset of C++ was implemented. The language which the new interpreter supports was named as C--. It includes all the basic C++ structures but not all of the advanced features like classes and namespaces. Clear error messages and localization were emphasized in the implementation of the interpreter. The intention was to make an interpreter which can be used to get started in programming.

The interpreter was developed from VIP 1.0 (Visual InterPreter) [13] which was made by Antti Virtanen. For this reason the new system was named as VIP 2.0. VIP is a graphical user interface which visualizes the running of the code with interpreter at runtime. There is a new visualized user interface coming to VIP 2.0, but within the scope of this thesis only a command line interface was implemented. The command line interface was named as CLIP (Command Line InterPreter). When VIP 2.0 gets ready, it is going to be used on elementary programming courses at Tampere University of Technology. The students can practise their programming skills with VIP and return some of their programming assignments via VIP. VIP 2.0 is going to be published as free software under the GPL licence. The interpreter was implemented with Java 1.5. SableCC (Sable Compiler Compiler) was used when making the parser.

The interpreter has not yet been taken into use but the preliminary feedback has been positive. The hypothesis is that when the interpreter is going to be taken into use, the students will learn the basic structures of C++ faster than before.

ALKUSANAT

Tämä diplomityö on jatkoa teokselle “Minkälainen on kysymys?” [9], jonka kirjoitin 6-vuotiaana, ja joka käsitteli pääasiassa tähtitiedettä. Kyseisen teoksen kirjoitin lahjaksi paapalleni. Tämä diplomityö sen sijaan käsittelee enemmänkin erästä tulkkia, jonka tein käytettäväksi TTY:n perusohjelmointikursseilla.

Tämä diplomityö löytyy PDF-muodossa sivuiltani http://koti.mbnet.fi/hluoma/ tai http://www.students.tut.fi/∼luoma/. Sähköpostia minulle voi lähettää osoitteisiin harri.luoma@tut.fi ja harri.luoma@gmail.com.

Haluan kiittää Hannu-Matti Järvistä, Essi Lahtista ja Harri Järveä diplomityöni lukemisesta, virheiden löytämisestä ja parannusehdotusten tekemisestä.

Tampereella 24. lokakuuta 2007

Harri Luoma
harri.luoma@tut.fi / harri.luoma@gmail.com
050-3783457

SISÄLLYS

KUVAT

TAULUKOT

TERMIT JA SYMBOLIT

Lyhenne	Selitys ja mahdollinen viite
ANT	Apache Ant, työkalu ohjelmistojen kääntämisen automatisointiin, http://ant.apache.org/
ANTLR	ANother Tool for Language Recognition, http://www.antlr.org/
ASCII	American Standard Code for Information Interchange, yleisesti käytössä oleva merkkien koodaustapa
AST	Abstract Syntax Tree, abstrakti jäsennyspuu, rakenne, joka toimii eräänlaisena välikielenä tulkissa
BASH	Bourne Again SHell, komentotulkki, http://www.gnu.org/software/bash/
BNF	Backus-Naur Form, metasyntaksikieli, jolla lausekielien syntakseja voidaan määritellä, http://en.wikipedia.org/wiki/Backus-Naur_form
C++	C:stä kehitetty olio-ohjelmointikieli
C--	C++:n osajoukko, jota VIP ja CLIP tukevat.
CGI	Common Gateway Interface, tekniikka, jonka avulla selain voi lähettää dataa palvelimella suoritettavalle ohjelmalle, http://www.w3.org/CGI/
CLIP	Command Line InterPreter, komentorivikäyttöliittymä VIP 2.0:n tulkkiin.
CodeWitz	Tampereen ammattikorkeakoululla aloitettu kansainvälinen projekti, jonka tehtävänä on tuottaa visuaalisia työkaluja ohjelmoinnin opetukseen, http://www.codewitz.com/
Convit	CONcurrent programming VIsualisation Tool, apuväline rinnakkaisen ohjelmoinnin opetukseen, http://www.cs.tut.fi/~convit/
CUP	Constructor of Useful Parsers, jäsentäjägeneraattori Javalle, http://www2.cs.tum.edu/projects/cup/
EBNF	Extended BNF, BNF:n laajennus, http://en.wikipedia.org/wiki/Extended_Backus_Naur_Form
Eclipse	Ohjelmointiympäristö, joka tukee mm. Javaa, http://www.eclipse.org/
EDGE	Edge is a Development Group for programming Education, ohjelmoinnin perusopetukseen suuntautunut ryhmä TTY:n Ohjelmistotekniikan laitoksella, http://www.cs.tut.fi/~edge/
FIFO	First In First Out
g++	GNU C++ compiler, vapaa C++ -kääntäjä, http://gcc.gnu.org/
GPL	General Public Licence, vapaa ohjelmistolisenssi, http://www.gnu.org/copyleft/gpl.html

Lyhenne	Selitys ja mahdollinen viite
GUI	Graphical User Interface, graafinen käyttöliittymä
HTTP	HyperText Transfer Protocol, tiedonsiirtoprotokolla Internetissä tapahtuvaan tiedonsiirtoon, http://www.w3.org/Protocols/
IRC	Internet Relay Chat, Internetin reaaliaikainen keskusteluympäristö
ISO-8859-1	ASCII-merkistön länsieurooppalainen laajennus
Java	Sun Microsystemsin kehittämä laitteistoriippumaton olio-ohjelmointikieli, http://java.sun.com/
JavaCC	Java Compiler Compiler, LL(k)-jäsentäjägeneraattori Javalle, https://javacc.dev.java.net/
JFlex	Java Fast LEXer, selaajageneraattori Javalle, kehitetty JLex:stä, http://www.jflex.de/
JIT	Just In Time
JLex	a LEXical analyzer generator for Java, selaajageneraattori Javalle, http://www.cs.princeton.edu/∼appel/modern/java/JLex/
JLine	Javalle tehty kirjasto, joka käsittelee komentoriviltä annettua syötettä, http://jline.sourceforge.net/
LEX	a LEXical analyzer generator, selaajageneraattori C:lle, http://dinosaur.compilertools.net/lex/index.html
Lintula	TTY:n tietotekniikan osaston Unix/Linux-ympäristö
Perl	Practical Extraction and Report Language, skriptimäinen ohjelmointikieli, http://www.perl.com/
SableCC	Sable Compiler Compiler, jäsentäjägeneraattori Javalle, http://sablecc.org/
SVN	SubVersioN, versionhallintaohjelmisto, http://subversion.tigris.org/
TTY	Tampereen teknillinen yliopisto
Unicode	Merkistöstandardi, joka määrittelee suurimman osan kaikista kirjoitettujen kielten merkeistä
UTF-8	8-bit Unicode Transformation Format, Unicode-merkkien koodaustapa
VEPE	Visual, Educational Programming Environment, mahdollinen VIPin seuraaja
VIP	Visual InterPreter, visuaalinen tulkki, se järjestelmä, jota tämä diplomityö käsittelee
VIP 1.0	VIPin vanhempi versio, tekijänä Antti Virtanen, http://www.cs.tut.fi/∼vip/
VIP 2.0	VIPin uudempi versio, jota tämä diplomityö käsittelee

Lyhenne	**Selitys ja mahdollinen viite**
YACC	Yet Another Compiler-Compiler, jäsentäjägeneraattori C:lle, http://dinosaur.compilertools.net/yacc/index.html

1. JOHDANTO

Tietokoneohjelmien tekeminen ja ohjelmoinnin oppiminen on hankalaa [15]. Opiskelijoiden on usein hankala päästä ohjelmoinnissa alkuun, koska ohjelmointikielten kääntäjät ovat erittäin monimutkaisia. Yksinkertainenkin ohjelma vaatii yleensä toimiakseen esimerkiksi kirjastojen ottamista mukaan. Tämä on hankalaa aloitteleville opiskelijoille, jotka yrittävät vasta opetella kielen perusrakenteita.

Tässä avuksi tulee ohjelmointikielen tulkki. Tulkin avulla ohjelmoinnin opetus voidaan aloittaa juuri kielen perusrakenteista, eikä ylimääräisiä 'loitsuja' tarvita. Näillä 'loitsuilla' tarkoitetaan tässä esimerkiksi nimiavaruuksien ja kirjastojen käyttöön ottamista. Tässä diplomityössä toteutettiin ohjelmointikielen tulkki C++:n osajoukolle. Tämä tulkki toimii siten, että se jäsentää opiskelijan kirjoittaman ohjelmakoodin, tekee sille semanttiset tarkistukset ja jos ohjelmakoodista ei löytynyt virheitä, suorittaa ohjelmakoodin. Tulkkia ei lähdetty tekemään puhtaalta pöydältä, vaan pohjana käytettiin Antti Virtasen tekemää VIP 1.0 -järjestelmää (Visual InterPreter) [13]. Tästä johtuen uusi järjestelmä nimettiin VIP 2.0:ksi.

C++ ei ole paras mahdollinen kieli ohjelmoinnin opetukseen [8], mutta sen valintaan ei tässä diplomityössä pystytty vaikuttamaan. C++:n heikkoudet ohjelmoinnin opetuksessa tiedettiin, ja juuri tästä syystä alettiin kehittää tulkkia, joka rajaa hieman C++:n ominaisuuksia ja antaa opiskelijoille helpommin ymmärrettäviä virheilmoituksia kuin kääntäjät. Tulkista on tarkoituksella jätetty pois C++:n pidemmälle meneviä ominaisuuksia, kuten nimiavaruudet, luokat ja poikkeukset. Tällä tavoin kieltä on saatu yksinkertaistettua siten, että aloittelevan ohjelmoijan on helpompi oppia sitä.

Professori Hannu-Matti Järvinen esittää vielä julkaisemattomassa kirjassaan [6] tulkkipohjaisen lähestymistavan ohjelmoinnin opetukseen. Käytettynä ohjelmointikielenä on C++. Kirjassa käydään läpi kaikki ohjelmoinnin perusasiat lähtien liikkeelle tulostuksesta (cout-lause). Tulkin avulla tulostus voidaan opettaa ennen kuin opiskelijat edes tietävät main-funktion olemassaolosta tai kirjastoista, joita cout-lause vaatii. Kirjassa edetään tämän jälkeen lausekkeisiin, ehdolliseen suoritukseen ja pian myös funktioihin. Kaikki tämä tapahtuu siten, että opiskelijoiden ei tarvitse missään vaiheessa kirjoittaa 'loitsuja', joita he eivät ymmärrä (esimerkiksi "using namespace std;"). Tämän lähestymistavan taustalla on se havainto, että ohjelmointia opitaan yleensä vähän kerrallaan liittämällä uusi tieto vanhaan tietoon [15].

Opiskelijoiden on myös usein hankala ymmärtää, miksi heidän ohjelmansa toimii odottamattomasti ja miten sen saisi korjattua [10]. Monilla opiskelijoilla on myös

ongelmia ohjelmakoodin ja sen kuvaaman rakenteen välisen suhteen ymmärtämisessä [15]. Tähän auttaa ohjelmien visualisointi, jolloin opiskelija näkee, miten ohjelman suoritus etenee. Visualisointi auttaa aloittelevaa ohjelmoijaa näkemään, miten tulkki suorittaa ohjelmakoodia. Visualisoinnilla tarkoitetaan tässä ohjelmakoodin suorituksen etenemisen esittämistä graafisesti. Visualisointi on helpompi toteuttaa tulkin kuin kääntäjän avulla.

Yhdistämällä nämä kaksi vaatimusta, tulkkipohjainen lähestymistapa ohjelmoinnin opetukseen ja ohjelmien visualisointi, saatiin idea tälle tulkille. Hannu-Matti Järvinen tarvitsee kirjansa tueksi tulkkia, joka pystyisi tulkkaamaan kaikki ne ohjelmoinnin perusrakenteet, joita kirjassa käydään läpi. Toisaalta haluttiin kehittää VIPin tulkkia eteenpäin, koska siinä oli muutamia vakavia puutteita. Kahta erillistä tulkkia ei kuitenkaan haluttu lähteä kehittämään ja ylläpitämään, joten lähdettiin tekemään tätä diplomityötä ja tulkkia.

Alkuperäinen tarkoitus oli tehdä VIP 1.0:aan uusi tulkki ja pitää visualisoitu käyttöliittymä ennallaan. Vanhan käyttöliittymän integroinnista tulkkiin kuitenkin luovuttiin, ja annettiin uuden käyttöliittymän tekeminen Harri Järven vastuulle. Tämän diplomityön puitteissa tulkille toteutettiin komentorivikäyttöliittymä, johon viitataan tulevassa myös nimellä CLIP (Command Line InterPreter).

Kuten VIP 1.0, myös VIPin uusi versio on tarkoitus ottaa käyttöön Tampereen teknillisellä yliopistolla ohjelmoinnin peruskursseilla. Opiskelijat voivat harjoitella ohjelmointitaitojaan VIPin avulla ja palauttaa sen kautta myös osan ohjelmointitehtävistään. Ennenkuin VIPin graafinen käyttöliittymä valmistuu, tulkkia voidaan kuitenkin käyttää opetuksen tukena jo edellä mainitun komentorivikäyttöliittymän avulla.

Tulkin tukema kieli nimettiin C--:ksi. Se sisältää kaikki C++:n perusrakenteet, mutta ei esimerkiksi luokkia eikä nimiavaruuksia. Se riittää kuitenkin mainiosti perusohjelmoinnin opetukseen. Uusi tulkki tukee suurempaa osaa C++:n rakenteista kuin vanha tulkki. Näitä uusia ominaisuuksia ovat esimerkiksi luettelotyypit (enum) ja valintalause (switch-case). Uudesta tulkista pystyy myös ottamaan osan ominaisuuksista pois päältä valitsimien avulla.

Tulkin ja komentorivikäyttöliittymän toteutuksessa painotettiin selkeitä virheilmoituksia ja niiden lokalisointia. Virheilmoitukset on lokalisoitu jo suomeksi ja englanniksi.

Tulkki toteutettiin Java 1.5:lla käyttäen olio-ohjelmointia. Jäsentäjän tekemisessä käytettiin apuna SableCC:tä (Sable Compiler Compiler, jäsentäjägeneraattori Javalle). Suunnittelu tehtiin vanhan VIPin ratkaisujen ja SableCC:n generoimien luokkien pohjalta. Testauksessa käytettiin suoraan niitä koodiesimerkkejä ja harjoitustehtäviä, joita tulkilla tullaan ajamaan.

Luvussa 2 esitellään VIPin historiaa sekä vaatimuksia uudelle versiolle. Myös kaikki C--:n ominaisuudet käydään tarkemmin läpi.

Luvussa 3 käydään läpi tulkkien ja kääntäjien teoriaa. Tämän jälkeen esitellään jäsentäjägeneraattori SableCC:n toimintaa ja lopuksi käsitellään virheilmoituksia ja lokalisointia.

Luvussa 4 käydään läpi uuden tulkin toteutus. Toteutus kuvataan melko tarkasti ja erityisesti selitetään, miten komentorivikäyttöliittymä ja sen erikoisominaisuudet on tehty.

Luvussa 5 on yhteenveto koko diplomityöstä, omia mietteitäni sekä jatkokehitysajatuksia.

Liitteessä 1 on esitetty tulkin luokkarakenne ja liitteessä 2 tärkeimmät rajapinnat. Liitteenä 3 on SableCC:lle tehty kielioppitiedosto, liitteenä 4 suomeksi lokalisoidut tekstit ja virheilmoitukset ja liitteenä 5 esimerkkiajo komentorivikäyttöliittymää käyttäen.

2. VIPIN TAUSTAA

Tässä luvussa esitellään VIPin historiaa. Sen jälkeen esitetään muutamia syitä, miksi uutta versiota VIPistä ruvettiin kehittämään. Lopuksi käydään läpi tärkeimmät vaatimukset uudelle versiolle ja ne kielen ominaisuudet, joista tulkin pitää selviytyä.

2.1 VIPin vanhemmat versiot

VIPillä on takanaan jo pitkä kehityshistoria. Ensimmäisen VIPin version (VIP 1.0) teki Antti Virtanen [13, 14] vuosina 2004 - 2005 CodeWitz-projektin puitteissa. VIP 1.0:aan kopioitiin joitain osia vanhemmasta CONVIT-työkalusta [7], joka oli apuväline rinnakkaisen ohjelmoinnin opetukseen. Ensimmäisen version jälkeen VIP 1.0:aan on lisätty joitain ominaisuuksia ja korjailtu käyttöliittymää.

Myös professori Mikko Tiusanen kehitti kesällä 2005 C--:n tulkin pohjalta uutta versiota, jonka hän nimesi 0-kielen tulkiksi. Tähän tulkkiin ei suunniteltu visualisointeja, vaan se olisi ollut pelkästään komentorivikäyttöliittymän avulla käytettävä. Ideana oli tarjota opiskelijoille tulkki, jota olisi helpompi käyttää ja ymmärtää kuin kääntäjää. Tämä 0-kielen tulkki oli hieman rajoittuneempi kuin VIP 1.0:n tulkki. Siinä ei esimerkiksi ollut lainkaan osoittimia. Lisäksi siinä merkittiin muuttujia var-sanalla (esimerkiksi “var int i = 0;”) ja aliohjelmia func ja proc-sanoilla. Sana func kirjoitettiin sellaisten aliohjelmien eteen, jotka olivat funktioita, eli jotka eivät muuttaneet parametrejaan, vaan ainoastaan palauttivat jonkin arvon. Sana proc oli sellaisia aliohjelmia varten, jotka olivat proseduureja, eli jotka eivät palauttaneet mitään arvoa mutta saattoivat muuttaa parametrejaan. Tässä oli siis tehty selvä poikkeavuus C++:aan, eikä 0-kieli ollut enää C++:n osajoukko. Tämä sama idea oli käytössä myös professori Hannu-Matti Järvisen tulkkipohjaisessa ohjelmoinnin alkeiskirjassa [6]. Vaikka tämä ratkaisuvaihtoehto saattaisi olla pedagogisesti hyvä, eriävän syntaksin takia se hylättiin suunniteltaessa uutta versiota C--:sta.

2.2 Syitä VIPin uuden version kehittämiseen

Uutta versiota VIPistä tarvittiin, koska VIP 1.0:n tulkissa oli käytetty arkkitehtuuriratkaisuja, jotka eivät enää skaalautuneet uusiin vaatimuksiin. Tulkissa käytetyt selaajageneraattori JFlex ja jäsentäjägeneraattori CUP olivat hankalia sen takia, että osa Java-ohjelmakoodista kirjoitettiin JFlex- ja CUP-tiedostoihin. Tuloksena oli paljon sekavaa ohjelmakoodia, jonka uudelleenkäytettävyys oli huono ja johon oli vaikeaa lisätä enää uusia ominaisuuksia. Sama ongelma oli myös 0-kielen tulkin kanssa. Se oli kirjoitettu vanhoilla C-kielelle tarkoitetuilla YACC ja LEX-työkaluilla.

Myös niitä käytettäessä ohjelmakoodi kirjoitetaan syntaksitiedostojen sekaan, joten ohjelmakoodin ylläpidettävyys kärsii. C-kielinen ohjelmakoodi ei muutenkaan ole kaikkein ylläpidettävintä koodia. Lisäksi tämä 0-kielen tulkki ei koskaan ehtinyt valmiiksi asti. Näiden syiden takia tästä 0-kielen tulkista ei ollut juurikaan hyötyä uutta tulkkia suunniteltaessa. VIP 1.0:ssa oli myös niin paljon virheitä, että järkevin vaihtoehto oli aloittaa kokonaan uuden tulkin tekeminen.

Myös muita ohjelmointi- ja visualisointiympäristöjä kokeiltiin, mutta ne visualisoivat yleensä Javaa eikä C++:aa, joten ne eivät suoraan soveltuneet TTY:n käyttötarkoituksiin. Tällaisia ympäristöjä ovat ainakin BlueJ ja Jeliot. Niistä ja visualisoinnista yleensä on selitetty tarkemmin Virtasen Antin diplomityössä [13].

Yksi tärkeä syy tämän uuden tulkin kehittämiseen oli professori Hannu-Matti Järvisen vielä julkaisematon kirja, jossa esitellään tulkkipohjainen lähestymistapa ohjelmoinnin opetukseen [6]. Kirjan tueksi tarvittiin tulkki, jota pystyy käyttämään komentorivikäyttöliittymän avulla ja joka suoriutuu kaikista kirjassa esitetyistä ohjelmoinnin perusrakenteista. Olemassaolevia "C++"-tulkkejakin testattiin, mutta mikään niistä ei kelvannut, koska niissä oli mukana yleensä koko C++ ja ne olivat hankalia käyttää. Ohjelmoinnin opetuksessa laajan C++:n osajoukon sijaan tärkeintä on helppokäyttöisyys ja selkeät virheilmoitukset.

Alun perin tarkoituksena oli, että VIP 1.0:n käyttöliittymä olisi integroitu uuteen tulkkiin. Tulkkia tehtäessä kävi kuitenkin melko pian ilmi, että se olisi melko turhaa ja aikaavievää, koska vanha käyttöliittymä perustui vahvasti vanhan tulkin ominaisuuksiin. Niinpä integroinnista luovuttiin ja päätettiin tehdä kokonaan uusi visuaalinen käyttöliittymä uuden tulkin pohjalle. Uutta visuaalista käyttöliittymää ei kuitenkaan alettu tehdä tämän diplomityön puitteissa vaan sen tekeminen annettiin Harri Järven harteille.

2.3 Vaatimukset uudelle tulkille

Uuden version päävaatimukset ovat lähes samat kuin VIPin vanhallekin tulkille. Uuden version pitää pystyä tulkkaamaan ja kääntämään kaikkia niitä "C--"-ohjelmia, joita vanha tulkkikin pystyi. Uusia vaatimuksia asettaa se, että tulkkia pitää pystyä käyttämään komentorivikäyttöliittymän avulla ajatellen tulkkipohjaista lähestymistapaa ohjelmoinnin opetukseen. Näitä vaatimuksia on käsitelty seuraavissa aliluvuissa. Tarkemmin ominaisuudet ja niiden toteutus on esitelty luvussa 4.

2.3.1 C--:n ominaisuudet

Ohjelmointikieli, jota sekä vanha että uusi VIP osaa tulkata, on C++:n osajoukko, joten sen nimeksi valittiin C--. Tässä kielessä on pyritty siihen, että pahimmat C++:n sudenkuopat on peitetty. Tällä saavutetaan se, että opiskelijat, jotka tulkkia käyttävät, eivät putoa näihin sudenkuoppiin eivätkä opi käyttämään huonoa tyyliä ohjelmakoodissaan, vaan tulkki ilmoittaa siitä heille heti.

Koska C--:ta ja uutta tulkkia on tarkoitettu käytettäväksi vain perusohjelmointikursseilla, kaikki pidemmälle menevät C++:n ominaisuudet, kuten luokat ja mallit, on jätetty pois. Tämä vaikutti osaltaan siihen, että tulkin ylipäänsä pystyi tekemään diplomityönä, eikä se ollut liian laaja. Taulukossa 2.1 on esitetty kaikki C++:n ominaisuudet ja kerrottu, mitkä niistä on toteutettu C--:ssa.

Taulukko 2.1: C--:n ominaisuudet

Ominaisuus	**Mukana C--:ssa / kommentit**
Perustietotyypit: int, double, float, bool, char, void	Kyllä
Perustietotyypit: short int	Toteutetaan C++:n int:inä
Perustietotyypit: long int, long double	Toteutetaan C++:n int:inä ja doublena
Perustietotyypit: long int, long double	Toteutetaan C++:n int:inä ja doublena
Perustietotyypit: unsigned int	Kyllä
Perustietotyypit: unsigned char	Toteutetaan charina
Kontrollirakenteet: if, else, while, for, do/while	Kyllä
Kontrollirakenteet: switch, case, default	Kyllä
Kontrollirakenteet: break, continue	Kyllä
Kontrollirakenteet: goto	Ei
Operaattorit: () .	Kyllä
Operaattorit: + - * / %	Kyllä
Operaattorit: aritmeettiset << >> <<= >>=	Ei
Operaattorit: << >> virroille	Kyllä
Operaattorit: < <= > >= == !=	Kyllä
Operaattorit: & | ^ ~ &= |= ^=	Ei
Operaattorit: && || !	Kyllä
Operaattorit: = += -= *= /= %=	Kyllä
Operaattorit: ?: ,	Ei
Operaattorit: ::	Ei
Operaattorit: sizeof	Ei
Operaattorit: & * -> (osoittimet)	Kyllä
Operaattorit: ->* .*	Ei
Operaattorit: [ ]	Kyllä
Operaattorit: new, delete	Ei
Operaattorit: and, not, or	Kyllä

Ominaisuus	**Mukana C--:ssa / kommentit**
Operaattorit: bitand, bitor, xor, compl	Ei
Operaattorit: and_eq, not_eq, or_eq, xor_eq	Ei
Operaattorit: ++i, --i, i++, i--	Kyllä, suosittelee etuliiteoperaattoreita
Operaattorit: Tyyppimuunnokset (tyyppi)muuttuja, tyyppi(muuttuja)	Ei
Operaattorit: static_cast	Kyllä
Operaattorit: dynamic_cast, reinterpret_cast, const_cast	Ei
Automaattiset tyyppimuunnokset	Rajoitetummat kuin C++:ssa, katso aliluku 4.1.3
Viitteet, viiteparametrit	Kyllä
Osoittimet, osoitinaritmetiikka	Kyllä
Funktio-osoittimet	Ei
Taulukot ja niiden alustus aaltosulkeilla	Kyllä
Taulukoissa monta indeksiä	Ei
Funktiot	Kyllä
Funktiot: rekursio	Kyllä
Funktiot: kuormittaminen	Ei
Funktiot: oletusparametrit	Kyllä
Funktiot: tuntematon parametrien määrä (...)	Ei
Luettelotyypit (enum)	Kyllä
Tietueet (struct), alustus aaltosulkeilla	Kyllä
Tietueet: Rakentaja, metodit	Ei
Luokat	Ei
Varatut sanat: const	Kyllä
Varatut sanat: asm, auto, extern, inline, mutable, register, static, typedef, union, volatile, wchar_t	Ei
Standardikirjastot: EXIT_SUCCESS, EXIT_FAILURE	Kyllä
Standardikirjastot: RAND_MAX, srand, rand	Kyllä
Standardikirjastot: abs, pow, sqrt, exp, log, sin, cos, tan, ceil, floor	Kyllä

Ominaisuus	**Mukana C--:ssa / kommentit**
Standardikirjastot: isalpha, isdigit, isspace, islower, isupper, tolower, toupper, iscntrl, isprint, ispunct	Kyllä
Standardikirjastot: string: luonti, [], length, at, +, size, =, +=, empty, clear, substr, ==, !=, < >, <=, >=, katenointi chariin ja toiseen stringiin	Kyllä
Standardikirjastot: string: size_type, iterator, rakentajat, swap, getline, erase, append, insert, replace, c_str, find, rfind, begin, end	Ei
Standardikirjastot: iteraattorit	Ei
Standardikirjastot: cin, cout, endl, operaattorit << ja >> perustietotyypeille ja stringille	Kyllä
Standardikirjastot: cerr, flush, streampos, eof, beg, cur, end, getline, get, peek, ignore, tellg, seekg, put	Ei
Standardikirjastot: vector: tyhjän vektorin luonti, at, push_back, pop_back, [], size, empty, clear, =, ==, !=	Kyllä
Standardikirjastot: vector: vektorit vektoreiden alkioina (moniulotteisuus)	Kyllä
Standardikirjastot: vector: size_type, rakentajat, <, >, <=, >=, front, back, swap, resize, insert, erase	Ei
Standardikirjastot: muut kirjastot: limits, fstream, sstream, iomanip, deque, list, set, map, algorithm	Ei
Esikääntäjä: #-alkuiset rivit	Hyväksytään syntaksissa mutta ei käsitellä mitenkään
using namespace, using xxx::yyy	Hyväksytään syntaksissa mutta ei käsitellä mitenkään
Erikoismakrot: __LINE__, __FILE__ yms.	Ei
Tiedostovirrat	Ei
Mallit (templatet)	Vain vector
Poikkeukset	Ei
Nimiavaruudet	Ei

Ominaisuus	**Mukana C--:ssa / kommentit**
Syntaksi: Monen muuttujan esittely samalla rivillä (int x, y;)	Ei
Syntaksi: Monta sijoitusta samalla rivillä (x = y = z;)	Ei
Syntaksi: Muuttujan esitteleminen tietueen esittelyn perässä (struct A { int b; } a;)	Ei
Kommentit // /* */	Kyllä
Koodinvaihtomerkit: \n \t \"\a \ \ \' \b	Kyllä
Tulkki tunnistaa kaikki varatut sanat	Kyllä
Estetään kaikkien varattujen sanojen käyttö muuttujien nimissä	Kyllä
Funktioiden ennakkoesittelyt	Kyllä
Tietueiden ja luettelotyyppien ennakkoesittelyt	Ei
Vahvempi tyypitysjärjestelmä kuin C++:ssa	Kyllä
Ajonaikaiset tarkistukset: taulukon rajat	Kyllä
Ajonaikaiset tarkistukset: osoittimen läpi viittaaminen	Kyllä
Ajonaikaiset tarkistukset: alustamattomien muuttujien käytön estäminen	Kyllä
Ajonaikaiset tarkistukset: päättymätön silmukka	Kyllä

2.3.2 Muut ominaisuudet

Tulkkia toteutettaessa piti ottaa huomioon myös visuaalisen käyttöliittymän tarpeet. VIP 2.0:aa on tarkoitus käyttää ohjelmoinnin peruskursseilla siten, että opiskelijat voivat tehdä sillä esimerkiksi viikkoharjoituksiaan. VIPin pitää pystyä esittämään opiskelijoille tehtävä siten, että opiskelijat voivat ohjelmoida sen oleellisimmat kohdat itse ja tämän jälkeen tarkistaa, toimiiko heidän ohjelmansa siten kuin sen pitäisi. Tietyissä tehtävissä opiskelijoiden toimintaa pitäisi pystyä lisäksi ohjaamaan siten, että he käyttäisivät esimerkiksi if-lausetta switch-lauseen sijaan. Lisäksi VIPin pitäisi pystyisi näyttämään ohjelmakoodin visualisoinnin aikana ohjelmoijalle, mitä ohjelmassa tällöin tapahtuu. Tämän takia tulkkiin tarvittiin myös muita ominaisuuksia, joita ei ole normaaleissa C++-kääntäjissä. Näitä ominaisuuksia ovat:

Valitsimet

Tulkkiin tarvittiin ominaisuutta, jonka avulla tulkin ymmärtämää kieltä voidaan muokata tarpeen mukaan. Tässä apuun tulevat valitsimet, joiden avulla jonkin tietyn ominaisuuden saa kytkettyä pois. Tämä mahdollisuus haluttiin ainakin switch-case-lauseelle ja osoittimille. Perusohjelmointikurssien ensimmäisissä harjoituksissa, joissa opetellaan esimerkiksi if-rakenteen käyttöä, switch-case-lause on hyvä kytkeä pois päältä, jotteivät opiskelijat pystyisi käyttämään sitä. Osoittimet taas halutaan kytkeä pois esimerkiksi silloin, kun opetellaan viitteiden käyttöä. Tällöin halutaan, etteivät opiskelijat vahingossa käytä osoittimia viitteiden sijaan. Opiskelijoiden on nimittäin erittäin helppo sotkeutua C++:n monimutkaiselta näyttävään osoitinsyntaksiin, joka vilisee &- ja *-merkkejä. Valitsimia on käsitelty tarkemmin aliluvussa 4.3.1, josta löytyy myös luettelo ominaisuuksista, jotka on mahdollista kytkeä pois päältä.

Annotaatiot

Erityiskommentteja, joita voi lisätä ohjelmakoodin sekaan, kutsutaan tässä diplomityössä annotaatioiksi. Ne jakautuvat ohjeteksteihin ja ohjauskäskyihin. Kun visualisoinnissa tullaan sellaiselle riville, jolle on lisätty ohjetekstejä, VIP näyttää kommentit ikkunassa. Ohjetekstien avulla opiskelijalle pystytään hyvin havainnollisella tavalla opettamaan, mitä milläkin rivillä tapahtuu.

Harjoitustehtävien suunnittelussa tarvittiin myös ominaisuutta, jolla ohjelmakoodiin voidaan lisätä lukittuja ja piilotettuja koodirivejä. Lukittujen rivien avulla opiskelijan muokattavaksi voidaan antaa ainoastaan osa koodiriveistä. Piilotettuja riviä taas tarvitaan silloin, kun ohjelmakoodiin halutaan piilottaa tarkastuskoodia, jonka avulla voidaan tarkastaa, toimiiko opiskelijan ohjelma oikein. Piilotetut rivit eivät näy opiskelijalle mitenkään. Sekä lukitut että piilotetut koodirivit voidaan lisätä ohjelmakoodiin ohjauskäskyjen avulla. Annotaatiot ja niiden syntaksi on selitetty tarkemmin aliluvussa 4.3.2.

VIP-virrat

Opiskelijoiden tekemien ohjelmien tarkastamiseksi tulkkiin tarvittiin lisäominaisuuksia. Niinpä tulkkiin tehtiin VIP out- ja VIP in -virrat. Nämä virrat ovat käytettävissä ainoastaan graafisessa käyttöliittymässä, kun opiskelija tekee sillä sellaista harjoitustehtävää, jolle on määritelty tarkastuskoodi. Komentorivikäyttöliittymässä tällaista mahdollisuutta ei ole.

VIP out -virran avulla tekstiä voi tulostaa erilliseen ikkunaan. Tätä tarvitaan, kun halutaan näyttää ohjelman testiajon päätteeksi, miten opiskelijan ohjelma suoriutui testistä. Kun opiskelijan ohjelmakoodia ajetaan, voidaan tarkastaa, mikä on kunkin

muuttujan arvo testin päätteeksi. Tällöin opiskelijalle voidaan esimerkiksi tulostaa, mikä oli muuttujan arvo ja mikä sen olisi pitänyt olla.

VIP in -virran avulla opiskelijan ohjelmalle voidaan antaa syötteitä. VIP in -virtaan tulostettaessa kyseiset muuttujat laitetaan cin-virran puskuriin ja seuraavan kerran cin-virrasta luettaessa käyttäjälle palautetaan puskurissa oleva arvo. Puskuri toimii FIFO-periaatteella siten, että ensiksi puskuriin pistetty tulee sieltä myös ensimmäisenä pois. Tällä tavalla puskuriin voidaan laittaa mielivaltainen määrä muuttujia. Virtojen toteutus on kuvattu aliluvussa 4.3.3.

2.3.3 Tulkattavuus

Yksi tärkeimmistä VIPin ominaisuuksista on tulkattavuus. Se on erittäin tärkeää perusopetukseen suuntautuvalle ohjelmointiympäristölle, koska sen avulla saavutetaan mahdollisuus visualisoida ohjelmakoodia. Tulkattavuus tarkoittaa sitä, että sen sijaan, että ohjelma käännettäisiin ja ajettaisiin perinteisenä binääritiedostona, ohjelmaa ajetaan rivi riviltä ja lauseke lausekkeelta. Jos käytössä on visualisoitu käyttöliittymä, käyttäjälle voidaan lihavoida esimerkiksi rivi ja lauseke, jossa ohjelman suoritus on menossa. Niiden lisäksi käyttäjälle voidaan näyttää muuttujien arvot sekä laskulausekkeet, jotka on evaluoitu. Komentorivikäyttöliittymässä tulkista saadaan myös se hyöty, että se antaa käyttäjälle palautteen nopeammin kuin kääntäjä, eikä kääntämisen ja kirjastojen mukaan ottamiseen mene turhaan aikaa. Tulkkien ja kääntäjien eroista on kerrottu tarkemmin aliluvussa 3.1.1.

2.4 Komentorivikäyttöliittymä

Komentorivikäyttöliittymän kantavana ideana on se, että sitä voi käyttää ympäristössä, jossa visualisoitua käyttöliittymää ei ole saatavilla. Lisäksi komentorivikäyttöliittymä eroaa toimintaperiaatteeltaan merkittävästi visualisoidusta käyttöliittymästä. Komentorivikäyttöliittymässä komentoja annetaan komentokehotteesta yksi kerrallaan ja ne suoritetaan saman tien. Sen sijaan visualisoidussa käyttöliittymässä vaihtoehtoja on muitakin. Joissain tapauksissa visualisoitu käyttöliittymä voi toimia samaan tapaan kuin komentorivikäyttöliittymäkin, mutta esimerkiksi VIP 1.0:ssa ohjelmat suoritetaan aina kokonaisuudessaan.

Komentorivikäyttöliittymään voi kirjoittaa ohjelmointikielen lauseita kuin oltaisiin main()-funktion sisällä. Muuttujia ja vakioita voi luoda kuten normaalisti. Tämän lisäksi on oltava mahdollista määritellä funktioita, tietueita ja luettelotyyppejä, kuin oltaisiin main()-funktion ulkopuolella. Näiden lisäksi komentorivikäyttöliittymässä on erikoiskomentoja muun muassa symbolitaulun näyttämiseen ja tulkista poistumiseen.

Ajatellen tulkkipohjaista lähestymistapaa ohjelmoinnin opetukseen, komentorivikäyttöliittymä toimii myös opiskelijoille eräänlaisena hiekkalaatikkona, jossa he

voivat opetella ohjelmoinnin alkeita omalla äidinkielellään sen sijaan, että he tappelisivat kääntäjien hankalien virheilmoitusten kanssa. Opiskelijat voivat aloittaa suoraan kirjoittamaan ohjelmakoodia välittämättä esimerkiksi kirjastojen mukaanottamisesta.

2.5 Visualisoitu käyttöliittymä

Visualisoidun eli graafisen käyttöliittymän suurin etu komentorivikäyttöliittymään verrattuna on tietenkin ohjelmakoodin visualisointi. VIP 1.0:ssa ei komentorivikäyttöliittymää ollut lainkaan, vaan järjestelmä pohjautui vahvasti visuaaliseen käyttöliittymään. Se on ollut myös VIP 2.0:n painopisteenä komentorivikäyttöliittymän rinnalla. Sen toteuttaminen ei kuitenkaan kuulunut tämän diplomityön alueeseen.

Visualisoidussa käyttöliittymässä käyttäjän ohjelmakoodia suoritetaan rivi kerrallaan visualisoiden rivi ja kohta, jota kulloinkin tulkataan. Käyttöliittymässä näkyy myös annotaatiot, joita kyseiselle riville on lisätty, sekä cout-virta. Ikkunassa näkyy myös kaikki muuttujat ja niiden nykyiset arvot, sekä laskulausekkeet ja niiden tulokset.

Visuaalisessa käyttöliittymässä ohjelmakoodin suorituksen voi pysäyttää ja jatkaa pysäytetystä kohdasta, asettaa pysähdyskohtia (breakpoints) tai asettaa tulkin etenemään askeleittain haluamallaan nopeudella. Koodia voi muokata erillisessä ikkunassa lukuun ottamatta lukittuja ja piilotettuja rivejä. Tehtävästä riippuen ohjelmakoodin voi tarkastaa, jolloin käyttäjä näkee, suoriutuiko hänen ohjelmakoodinsa sille annetusta tehtävästä.

3. TULKKIEN TEORIAA

Tässä luvussa käydään läpi tulkkien ja kääntäjien teoriaa. Lähteenä on käytetty pääasiassa Appelin kirjaa "Modern Compiler Implementation in Java" [1]. Luvun alussa verrataan tulkkia ja kääntäjää, ja sen jälkeen käydään läpi kaikki lausekielen tulkkauksen vaiheet: leksikaalianalyysi, jäsennys, abstrakti jäsennyspuu ja semanttinen analyysi. Näiden jälkeen esitellään SableCC:n arkkitehtuuri ja syntaksi, jolla kieli määritellään SableCC:lle. Lopuksi käsitellään virheilmoituksia ja lokalisointia.

Antti Virtanen käy diplomityössään [13] läpi jo suuren osan tulkkien, kääntäjien ja lausekielien teoriasta. Tämän takia tässä käydään läpi näitä asioita hieman pintapuolisemmin keskittyen SableCC:n kuvaamiseen.

3.1 Tulkeista

3.1.1 Tulkki vs. kääntäjä

Ohjelmointikielet jaetaan perinteisesti tulkattaviin ja käännettäviin kieliin. Yleensä C++ ja muut vastaavat lausekielet luokitellaan käännettäviin kieliin ja skriptikielet, kuten Python, tulkattaviin kieliin. Jako ei kuitenkaan ole näin selvä, sillä monille kielille on olemassa sekä tulkki että kääntäjä.

Tulkkaaminen tarkoittaa sitä, että ohjelmakoodia suoritetaan rivi ja lauseke kerrallaan. Ohjelmakoodia ei käännetä suoraan konekielelle, vaan tulkki ja tulkkausympäristö huolehtivat koodin ajamisesta reaaliaikaisesti. Tämä lähestymistapa tarjoaa mahdollisuuden ohjelmakoodin reaaliaikaiseen seuraamiseen.

Kääntäminen taas tarkoittaa sitä, että ohjelmakoodi muutetaan kerralla konekieliseksi koodiksi. Kääntämisen jälkeen se linkitetään yhteen kirjastojen kanssa ja ajetaan konekielisenä koodina. Näin ollen käännettyä koodia on hankalaa seurata mitenkään ajonaikaisesti.

Tulkkien ja kääntämisen välimaastoon kuuluvat kääntäjät, jotka kääntävät ohjelmakoodia jollekin tavukielelle, joka on yleensä jo melko optimoitua ja tiivistettyä koodia. Se ei silti ole konekieltä, joten se on laitteistoriippumatonta. Tämän tavukoodin ajamiseen tarvitaan vielä erillinen kääntäjä, niin sanottu virtuaalikone. Tätä lähestymistapaa käytetään esimerkiksi Javassa.

JIT-tekniikassa (Just In Time) tulkki kääntää ohjelmakoodin konekielelle juuri ennen sen ajamista. Tämä antaa paremman suorituskyvyn itse ohjelmakoodin ajamiseen, mutta koodin kääntämiseen saattaa kulua enemmän aikaa.

Koska tulkkien ei tarvitse kääntää ohjelmakoodia konekielelle, ne voivat toimia korkeammalla abstraktiotasolla kuin kääntäjät. Ne voivat visualisoida ohjelmakoodia ja näyttää muuttujien arvot reaaliaikaisesti käyttäjälle. Samalla kuitenkin menetetään tehokkuus, eikä ohjelmakoodia pystytä optimoimaan. Visualisointikäytössä tulkin tehokkuus ei kuitenkaan yleensä ole ongelma.

3.1.2 Lausekielten tulkkaus

Riippumatta siitä, onko kyseessä lausekielen kääntäjä vai tulkki, sen on osattava kääntää sitä kieltä, jolle se tehdään. Tähän prosessiin kuuluu aina leksikaalianalyysi, jäsennys sekä semanttinen analyysi. Uudemmissa kääntäjissä on yleensä myös vaihe, jossa konkreettinen jäsennyspuu muunnetaan abstraktiksi. Seuraavassa kääntämistä käsitellään tulkin kannalta.

Jotta ohjelmasta ymmärrettäisiin jotain, se täytyy ensin analysoida. Kuten edellä jo mainittiin, analysointi jakautuu kolmeen eri osaan: ohjelmalle suoritetaan leksikaalinen, syntaktinen ja semanttinen analyysi.

Leksikaalianalyysi

Leksikaalianalyysissä lähdekoodi paloitellaan leksikaalialkoiksi eli tokeneiksi. Leksikaalianalyysiä kutsutaan myös selaamiseksi. Tällaisia leksikaalialkioita ovat esimerkiksi muuttujan nimi 'foo', numerot '23' ja '2.23', varatut sanat 'if' ja 'int' sekä ohjelmakoodissa olevat muita leksikaalialkioita ympäröivät merkit kuten ';' ja '{'. Nämä leksikaalialkiot annetaan sitten järjestyksessä jäsentäjälle, joka jäsentää ohjelmakoodin. On myös leksikaalialkioita, jotka hylätään heti leksikaalianalyysissä eikä niitä koskaan anneta jäsentäjälle. Tällaisia leksikaalialkioita ovat esimerkiksi kommentit ja välilyönnit.

Leksikaalialkoiden tunnistamiseen käytetään säännöllisiä lausekkeita. Kone muuttaa säännölliset lausekkeet äärellisiksi automaateiksi, joiden avulla merkkijonot pystytään tunnistamaan eri leksikaalialkioryhmiksi. Tällaisia leksikaalialkioryhmiä ovat esimerkiksi muuttujannimet ja numeraaliset vakiot. Leksikaalialkioiden tunnistuksessa käytetään seuraavia kahta sääntöä:

- Leksikaalialkioksi valitaan aina mahdollisimman pitkä pätkä tekstiä.
- Tietynpituiselle merkkijonolle valitaan aina ensimmäinen leksikaalialkiotyyppi, jonka säännölliseen lausekkeeseen se täsmää. On siis merkitystä sillä, mihin järjestykseen leksikaalialkioiden säännöt kirjoitetaan.

Jos ohjelmaa ei pystytä jakamaan leksikaalialkioksi, tapahtuu leksikaalivirhe ja leksikaalinen analyysi keskeytetään.

Jäsennys

Jäsennysvaiheessa ohjelman rakennetta analysoidaan ja kaikki sen lauseet jäsennetään. Jäsennykseen käytetään yhteysriippumattomia kielioppeja. Niitä kuvataan yleensä BNF-notaatiolla (Backus-Naur Form).

BNF-notaatio koostuu produktioista, joiden vasemmalla puolella on yksi välisymboli ja oikealla puolella 1..n kappaletta (yksi tai enemmän, n on tässä mielivaltainen positiivinen kokonaisluku) väli- tai loppusymboleja. Produktio voi olla esimerkiksi "summa ::= lauseke '+' lauseke". Tämä tarkoittaa, että summa on nyt rakenne, jossa on kaksi lauseketta ja niiden välissä '+'-merkki. Lauseke on vastaavasti määritelty jossain toisessa produktiossa. Produktioissa voidaan käyttää myös pystyviivaa '|'. Se tarkoittaa, että produktion vasemmalla puolella oleva välisymboli voi jäsentyä kummaksi tahansa pystyviivan vasemmalla tai oikealla puolella olevaksi lausekkeeksi.

BNF-notaatiota on myös laajennettu EBNF-notaatioksi (Extended BNF). Se tuo mukaan '?', '*' ja '+'-merkit. '?'-merkki symbolin perässä tarkoittaa, että symboli voi tulla mukaan tai jäädä pois. '*'-merkki tarkoittaa, että kyseinen symboli tulee mukaan 0..n kertaa. '+'-merkki on muuten vastaava, mutta siinä kyseinen symboli tulee mukaan 1..n kertaa. Kaikki EBNF:llä ilmaistavat kielet voidaan kuitenkin ilmaista myös BNF:llä, joten '?', '*' ja '+'-merkkien lisääminen BNF:ään ei lisää sen ilmaisuvoimaa, ainoastaan helpottaa merkintätapoja.

Ensimmäisen produktion vasemmalla puolella oleva välisymboli on alkusymboli, jonka pitäisi vastata ohjelmointikielissä koko ohjelmaa. Ensimmäinen produktio voi siis olla esimerkiksi "ohjelma ::= lause (';' lause)*", mikä tarkoittaa, että nyt ohjelma voi sisältää 1..n kappaletta lauseita eroteltuina puolipisteillä.

Jäsennysvaiheessa määritellään operaattoreiden sitomisjärjestys. Oletetaan, että kieliopissa olisi esimerkiksi seuraavat produktiot:

- lauseke ::= lauseke '+' termi | termi;
- termi ::= termi '*' tekijä | tekijä;
- tekijä ::= [0-9]+;

Tällöin lauseke 5 * 6 + 7 * 8 jäsennettäisiin (5 * 6) + (7 * 8). Lisäksi operaattorit olisivat vasemmalle sitovia, koska 5 * 6 * 7 jäsennettäisiin (5 * 6) * 7 ja 5 + 6 + 7 jäsennettäisiin (5 + 6) + 7.

Produktioiden laatimisessa pitää olla tarkkana, ettei kieliopista tule moniselitteinen. Jos kielioppi on moniselitteinen, sama lauseke voidaan jäsentää monella eri tavalla. Koska ohjelmointikielissä saman ohjelmakoodin täytyy jäsentyä aina samalla tavalla, moniselitteisellä kieliopilla ei voi määritellä ohjelmointikieltä. Joissain ohjelmointikielissä, kuten C++:ssa, on kuitenkin moniselitteisyyttä. Esimerkiksi C++:n esittely 'vector<int> i();' voisi tarkoittaa sekä muuttujan että funktion esittelyä. Tällaisissa tapauksissa moniselitteisyys pitää poistaa jollain keinolla. C++:ssa äskeinen esimerkki tulkitaan funktion esittelyksi (funktio, joka palauttaa vektorin).

Sen sijaan 'vector<int> i(5);' olisikin jo muuttujan esittely (vektori, jonka pituus on 5). Joillain työkaluilla, kuten CUPilla (Constructor of Useful Parsers, jäsentäjägeneraattori Javalle), voidaan moniselitteisyys poistaa joissain tapauksissa presedenssisäännöillä. SableCC:ssä (Sable Compiler Compiler, uudempi jäsentäjägeneraattori Javalle) tällaista mahdollisuutta ei ole.

Jäsennystavat

Jäsennystapoja on kaksi: kokoava jäsentäminen ja pelkistävä jäsentäminen. Kokoavat jäsentäjät (recursive descent) ovat yleensä LL(k)-jäsentäjiä (Left-to-right Leftmost-derivation, k-symbol lookahead). Ensimmäinen L (Left-to-right) tarkoittaa, että jäsentäjä lukee ohjelmakoodia vasemmalta oikealle. Toinen L (Leftmost-derivation) tarkoittaa, että jäsentäjä toimii ylhäältä alas -periaatteella siten, että se korvaa ohjelmakoodista löytyviä loppusymboleja välisymboleilla. Tullessaan jonkin symbolin kohdalle jäsentäjän on osattava seuraavan k:n merkin perusteella päätellä, mitä välisymbolia aletaan seuraavaksi koota. Kokoava jäsentäjä jäsentää siis produktioita vasemmalta oikealle.

Pelkistävät jäsentäjät (shift-reduce) ovat yleensä LR(k)-jäsentäjiä (Left-to-right Rightmost-derivation, k-symbol lookahead). Tässä ensimmäinen L tarkoittaa samaa kuin äskenkin. R (Rightmost-derivation) tarkoittaa, että jäsentäjä toimii alhaalta ylös -periaatteella, eli että jäsennys tapahtuu oikealta vasemmalle. Se toimii siten, että jäsentäjä kokoaa itselleen pinoa ohjelmakoodista löytyneistä symboleista. Aina uuden symbolin tullessa se tarkistaa pinon sekä k:n seuraavan symbolin avulla, pystyykö se redusoimaan symbolipinon jonkin produktion avulla. Suurin osa automaattisista jäsentäjägeneraattoreista generoi pelkistäviä jäsentäjiä. Käytännössä kaikki tällaiset jäsentäjät ovat LALR(1)-jäsentäjiä (Look-Ahead LR(1)), koska niille tunnetaan tehokkaat jäsennysalgoritmit. Vaikka LALR(1)-jäsentäjillä ei pystytä jäsentämään kaikkia niitä kieliä, mitä LR(1)- tai LR(k)-jäsentäjillä pystytään, niillä pystytään kuitenkin jäsentämään käytännössä kaikki ohjelmointikielet.

Kokoavissa ja pelkistävissä jäsentäjissä on myös eroja sen suhteen, minkälaisia rekursioita ne pystyvät käsittelemään syntaksissa. Kokoavat jäsentäjät eivät pysty käsittelemään kielioppeja, joissa esiintyy vasenta rekursiota (esimerkiksi "lauseke ::= lauseke '+' termi"). Vastaavasti pelkistävät jäsentäjät eivät pysty käsittelemään oikeaa rekursiota (esimerkiksi "lauseke ::= termi '+' lauseke"). Kokoavien jäsentäjien toimintaperiaate on ihmiselle yleensä helpompi ymmärtää kuin pelkistävien jäsentäjien periaate. Toisaalta jäsentäjägeneraattoreiden tekemät jäsentäjät ovat yleensä pelkistäviä, koska niitä on helpompi generoida kuin kokoavia jäsentäjiä.

Abstrakti jäsennyspuu

Abstraktia jäsennyspuuta tehtäessä konkreettisen syntaksin produktioista tehdään abstraktimpia produktioita. Tässä vaiheessa syntaksista jätetään pois kaikki sellaiset

produktiot, joita ei tarvita enää semanttisessa analyysissä. Produktioista jätetään pois myös kaikki sellaiset leksikaalialkiot, joilla ei enää tehdä mitään (esimerkiksi puolipisteet). Tällä tavalla konkreettisesta syntaksista, joka on usein tarpeettoman monimutkainen semanttisen analyysin kannalta, saadaan yksinkertaistettu versio, jossa on kuitenkin riittävästi informaatiota, jotta siitä voidaan tehdä vielä semanttinen analyysi.

Abstraktin syntaksin produktioista kootaan sitten abstrakti syntaksipuu (AST), joka toimii hyvin tulkkaamisen tai kääntämisen välikielenä. Abstraktin syntaksipuun lisäksi välikieliä saattaa olla muitakin, kääntäjästä riippuen, esimerkiksi Javan tapauksessa tavukoodi.

AST:n solmut kuvaavat produktioita. Solmujen lapsina ovat kyseisen produktion eri osat, jotka voivat taas vastaavasti olla produktioita. Lehtinä puussa ovat yksittäiset leksikaalialkiot, kuten esimerkiksi kokonaislukuvakiot.

Semanttinen analyysi

Semanttisessa analyysissä määritellään, mitä kukin ohjelman lause tai lauseke tarkoittaa. Muuttujat kiinnitetään niiden esittelyyn ja tehdään kaikki tyyppitarkastelut. Myös kaikki muut mahdolliset käännösvaiheessa tehtävät tarkastukset tehdään.

Semanttisessa analyysissä on aina mukana yksi tai useampi symbolitaulu. Niihin kerätään tiedot muuttujista, joita kullakin näkyvyysalueella on. Näiden tietojen avulla pystytään tekemään tyyppitarkastelut kaikissa niissä kohdissa, joissa kyseisiä muuttujia käytetään. Jos jokin virhe havaitaan, annetaan siitä virheilmoitus käyttäjälle. Virheen vakavuuden mukaan kääntämistä joko jatketaan tai se keskeytetään.

Seuraavassa luettelossa esitetään muita asioita, joita tyyppitarkastusten lisäksi semanttisessa analyysissä voidaan tarkistaa.

- Funktiolle annettavien parametrien määrän ja tyyppien pitää vastata funktion esittelyä.
- Havaitaan muuttujat, joita ei ole esitelty.
- Vakion (const) muuttamisen estäminen.
- break- ja continue-käskyjen pitää olla silmukan sisällä.

3.2 SableCC

3.2.1 SableCC yleisesti

SableCC (Sable Compiler Compiler) [3] on Javalle tehty LALR(1)-jäsentäjägeneraattori. Sille annetaan tiedosto, jossa määritellään kaikki leksikaalialkiot, produktiot sekä abstrakti syntaksipuu. Niiden perusteella se generoi automaattisesti abstraktin syntaksipuun luokat sekä niiden lisäksi Switch-suunnittelumallin mukaiset luokat, joilla abstraktia syntaksipuuta on näppärä käydä läpi.

SableCC helpottaa huomattavasti jäsentäjän tekemistä verrattuna siihen, että sitä yrittäisi tehdä käsin. Myös verrattuna muihin Javalle kehitettyihin selaaja- ja jäsentäjägeneraattoreihin, kuten JFlexiin ja CUPiin, SableCC on astetta helppokäyttöisempi:

- JFlex ja CUP ovat erillisiä työkaluja toisin kuin SableCC, jolla voi generoida samalla kertaa sekä selaajan että jäsentäjän.
- CUPissa on mahdollisuus asettaa komentoja, joiden avulla pystyy jäsentämään moniselitteisiä kielioppeja, mikä tekee kieliopista usein hyvin vaikeaselkoisen.
- CUPissa ei ole mahdollisuutta generoida AST:tä automaattisesti.
- Java-ohjelmakoodi on upotettu CUP-tiedostojen sekaan, mikä tekee niistä vaikeaselkoisia ja hyvin pitkiä.
- SableCC tukee Unicodea. Myös JFlex tukee sitä nykyään, mutta sen vanhempi versio JLex ei sitä tukenut.

Lisäksi SableCC:tä kehitetään edelleen. Tätä kirjoitettaessa (18. heinäkuuta 2007) SableCC:stä on juuri ilmestynyt versio 4-alpha.2, ja versio 4.0 on kehitteillä. Viimeisin vakaa versio on kuitenkin parin vuoden takainen 3.2, jolla VIP 2.0:n syntaksikin on käännetty, ja jota tässä diplomityössä käsitellään. Versio 3.2 käyttää Javan versiota 1.5, joten tuki Java 1.5:lle on oltava käännös- ja ajoaikana.

Javalle on olemassa myös LL(k)-jäsentäjägeneraattoreita, kuten ANTLR ja JavaCC. Kuten SableCC:ssäkin, myös ANTLR:ssä ja JavaCC:ssä on mahdollisuus generoida AST automaattisesti. SableCC:n generoima AST on kuitenkin vahvemmin tyypitetty ja tarkastettu, joten se ei voi korruptoitua yhtä helposti. Näiden LL(k)-jäsentäjägeneraattoreiden toimintatapa on kuitenkin ehkä hieman intuitiivisempi kuin SableCC:n, joten ohjelmointi ja virheiden metsästys syntaksista saattaa olla hieman helpompaa niiden avulla. SableCC:n virheilmoituksia on moitittu hankaliksi ymmärtää. Lisäksi, SableCC ei myöskään laita solmuluokkiin tietoa siitä, missä kohti lähdekoodia ne sijaitsevat. Tämä aiheuttaa melkoisesti lisätyötä, jos esimerkiksi AST:n solmujen rivinumerot halutaan tietää semanttisessa analyysissä. SableCC kuitenkin valittiin, koska sen syntaksi on helppolukuisempaa kuin JavaCC:n ja ANTLR:n ja sillä generoitu jäsentäjä on suhteellisen nopea. Lisäksi monet työkaverini suosittelivat sitä minulle.

3.2.2 SableCC:n syntaksi

SableCC:lle annetaan syötteenä kieliopin syntaksitiedosto, jossa on seuraavat osat:

Package Määritellään Java-pakkaus (eli hakemistopolku), johon SableCC generoi luokat.

Helpers Määritellään merkit, joita käytetään myöhemmin tokens-osassa.

States Määritellään tilat, joissa selaaja voi olla lukiessaan lähdekoodia.

Tokens Määritellään leksikaalialkiot.

Ignored Tokes Määritellään ne leksikaalialkiot, jotka ohitetaan jäsennysvaiheessa.

Productions Jäsennysvaiheessa käytettävät produktiot.

Abstract Syntax Tree Abstraktin syntaksipuun kuvaus.

Näistä osista suurimman osan voi jättää halutessaan pois. Järkevään syntaksitiedostoon tarvitaan kuitenkin ainakin Tokens ja Productions.

Seuraavassa käydään vielä tiedoston rakennetta tarkemmin läpi esimerkin avulla. Tarkoituksena on tehdä syntaksi kielelle, joka pystyisi jäsentämään muun muassa seuraavan ohjelmanpätkän:

```
a = 567 * 345 + 2;
b = 12 + 6 * 0;
summa = a + b;
print summa;
```

Sallimme myös normaalien "C++"-kommenttien kirjoittamisen ohjelmakoodin sekaan. Tiedosto aloitetaan kirjoittamalla pakkauksen nimi:

```
Package test;
```

Tämän jälkeen tulevat apumerkinnät, joilla määritellään kaikki merkit, joita leksikaalianalyysivaiheessa voi käyttää.

```
Helpers
  all = [0 .. 0xffff];
  cr = 13;
  lf = 10;
  tab = 9;
  eol = cr | lf | cr lf;
  not_eol = [all - [cr + lf]];
  not_star = [all - '*'];
  not_star_slash = [not_star - '/'];
  letter = [['a' .. 'z'] + ['A' .. 'Z']];
  digit = ['0' .. '9'];
  empty = eol | tab | ' ';
```

Helpers-osassa merkkejä voi luetella joko ASCII-muodossa heittomerkkien sisällä, desimaalilukuina tai heksalukuina. Desimaali- ja heksalukujen arvot tulkitaan merkkien ASCII-arvoiksi. Hakasulkeiden ja kahden pisteen avulla voi ilmaista merkkivälejä. Tokenit eli leksikaalialkiot eroavat edellisistä Helpers-säännöistä siten, että leksikaalialkioita voi käyttää produktioissa.

```
Tokens
  blank = empty+;
  comment = ('//' not_eol* eol)
    | ('/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/')
    ;
  comma = ',';
  semicolon = ';';
  star = '*';
  plus = '+';
  eq = '=';
  print = 'print';
  digit_sequence = digit+;
  identifier = letter (letter | digit)*;
```

Seuraavaksi määritellään leksikaalialkiot, jotka hylätään jäsennysvaiheessa. Yleensä tällaisiksi leksikaalialkioiksi määritellään kommentit ja välilyönnit yms.

```
Ignored Tokens
  comment, blank;
```

Produktioissa määritellään itse syntaksi. Tässä samalla määritellään myös muunnos AST-puuksi:

```
Productions
  program {-> program} =
    statement+ {-> New program([statement.statement])};
  statement {-> statement} =
    {assignment} assignment semicolon
      {-> New statement.assignment(assignment.identifier, assignment.expression)}
    | {print} print expression semicolon
      {-> New statement.print(expression.expression)};
  assignment {-> identifier expression} =
    {assignment} identifier eq expression
      {-> identifier expression.expression};
  expression {-> expression} =
    {multiply} expression star factor
      {-> New expression.multi(expression.expression, factor.expression)}
    | {factor} factor {-> factor.expression};
  factor {-> expression} =
    {addition} factor plus primary
      {-> New expression.plus(factor.expression, primary.expression)}
    | {primary} primary {-> primary.expression};
  primary {-> expression} =
    {identifier} identifier {-> New expression.identifier(identifier)}
    | {digit} digit_sequence {-> New expression.digit(digit_sequence)};
```

Tässä kaikki "{->}"-osat ovat muunnosta AST:hen. Esimerkiksi program -produktio muunnetaan AST:n samannimiseksi solmuksi. Kaikille eri vaihtoehdoille (pystyvii-

valla erotetut) pitää määritellä, miten ne muunnetaan AST:n solmuksi. Vaihtoehdot pitää muuntaa samantyyppisiksi AST-solmuiksi kuin mitä produktioiden vasemmalla puolella määritellään.

Katsotaan tarkemmin statement-produktiota. Se kuvataan samannimiseksi AST-solmuksi. Sillä on kaksi eri vaihtoehtoa, jotka on erotettu |-merkillä. Ensimmäinen on nimetty assignmentiksi. Nimellä on tässä vain se tarkoitus, että SableCC osaa erottaa vaihtoehdot assignment ja print toisistaan. Nimeä ei tarvita, jos vaihtoehtoja on vain yksi, kuten esimerkiksi program-produktiossa. Nimen jälkeen tulevat ne väli- ja loppusymbolit, jotka muodostavat produktion tämän vaihtoehdon. Nuolen jälkeen on esitetty kyseisen assignment-vaihtoehdon muuntaminen AST-solmuksi. New-operaattorilla luodaan uusi luokka, jonka nimi on AAssignmentStatement. Se ja APrintStatement periytyvät PStatement-luokasta ja muodostavat näin luokkahierarkian. Näin tulee ollakin, sillä statement-produktio kuvattiin statement-nimiseksi AST-solmuksi ja näin ollen SableCC olettaa, että kaikki vaihtoehdot on kuvattu PStatement-luokasta periytyviksi luokiksi. Kaikki SableCC:n generoimat luokat nimetään siten, että produktioiden eteen laitetaan P (esimerkiksi PStatement). Vaihtoehtojen eteen laitetaan A ja tämän jälkeen vaihtoehdon ja produktion nimet isolla alkukirjaimella (esim. AAssignmentStatement).

Joskus vaihtoehtoa AST-solmuksi kuvattaessa uutta luokkaa ei tarvitse luoda, vaan voidaan vain viitata toiseen produktioon, kuten on tehty esimerkiksi expression-produktion factor-vaihtoehdossa. Siinä factor.expression viittaa siihen, että factor-produktio kuvataan PExpression-luokan aliluokaksi. Tämä onnistuu, koska expressionkin on määritelty kuvattavaksi PExpression-luokan aliluokaksi. Tällä tavoin expression-hierarkia saadaan supistettua kasaan.

Jos AST aiotaan tehdä, kaikki produktiot täytyy kuvata AST-solmuiksi. Jos AST:tä ei tehdä, tällöin kuvauksia ei tarvita, ja SableCC generoi luokat suoraan produktioiden perusteella.

Lista voidaan määritellä hakasulkeiden avulla, kuten program-produktiossa on tehty. Tästä luodaan Javan LinkedList-tyypin olio. EBNF-syntaksin mukaisia +, * ja ? -operaattoreita voi käyttää kaikissa järkevissä paikoissa.

AST:tä käyttämällä produktioista tulee hieman monimutkaisia, mutta samalla itse AST ja siitä generoitu luokkarakenne yksinkertaistuu. AST täytyy vielä määritellä erikseen:

```
Abstract Syntax Tree
  program = statement*;
  statement =
    {assignment} identifier expression
    | {print} expression;
  expression =
    {multi} [exp1]:expression [exp2]:expression
    | {plus} [exp1]:expression [exp2]:expression
    | {identifier} identifier
    | {digit} digit_sequence;
```

Nyt AST näyttää yksinkertaiselta, ja siitä generoituukin vain kourallinen luokkia: AProgram, AAssignmentStatement, APrintStatement, AMultiExpression, APlusExpression, AIdentifierExpression ja ADigitExpression. Näille generoidaan vielä yliluokat PProgram, PStatement ja PExpression. Nämä kaikki periytyvät Node-luokasta. Myös kaikille leksikaalialkioille generoidaan luokat: TBlank, TComment, TComma jne. Kaikki nämä luokat sijoitetaan pakkaukseen test.node.

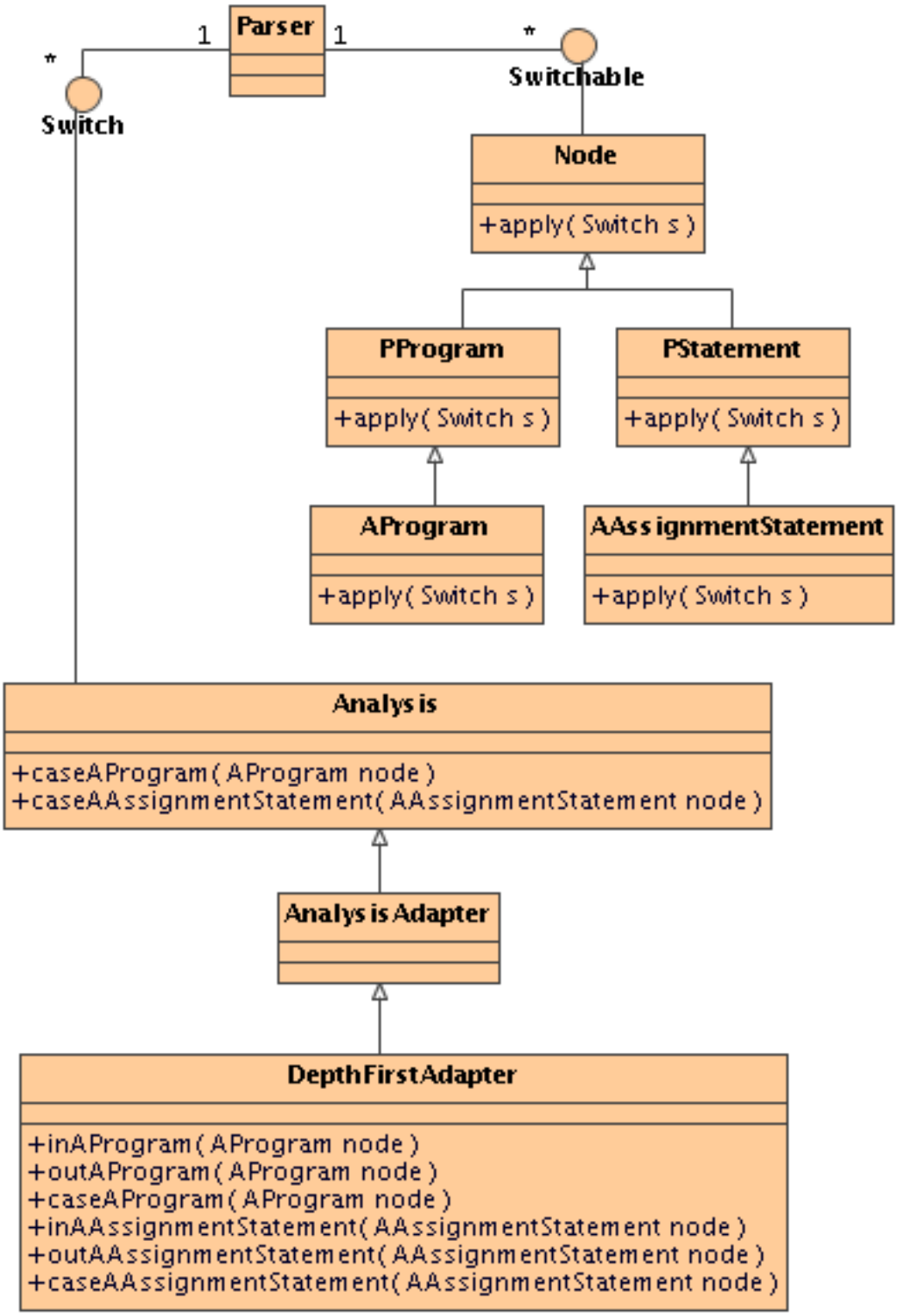
AProgram-luokalle generoidaan jäsenmuuttujaksi LinkedList<PStatement> statements, APrintStatementille jäsenmuuttujaksi PExpression expression ja AMultiExpressionille PExpressionit exp1 ja exp2. Tässä "[exp1]:"-syntaksi tarkoittaa uudelleennimeämistä. Uudelleennimeäminen täytyy tehdä, koska luokalla ei voi olla kahta samannimistä jäsenmuuttujaa.

Nyt syntaksi on valmis ja sen voi generoida SableCC:n avulla. SableCC generoi yllämainittujen luokkien lisäksi node-pakkauksen alle vielä pakkaukset parser ja lexer, joihin generoidaan jäsentäjä ja selaaja. Näiden lisäksi SableCC generoi analysis-pakkauksen, josta löytyy muun muassa DepthFirstAdapter-luokka, jonka avulla AST-puuta on helppo käydä läpi.

Tässä esimerkissä ei käyty läpi States-osiota. Sen avulla voidaan määrittää tiloja, joissa selaaja voi olla. Tilojen määrittelemisen jälkeen leksikaalialkioille voi määritellä, missä tiloissa ne tunnistetaan. Selaaja voi olla kerrallaan vain yhdessä tilassa, ja leksikaalialkion lukemisen jälkeen selaajalle voidaan määritellä uusi tila. Leksikaalialkion lukeminen voi siis aiheuttaa tilasiirtymän.

3.2.3 Switch-suunnittelumalli

SableCC käyttää AST-puun läpikäyntiin Switch-suunnittelumallia [3], joka on kehitetty Visitor-suunnittelumallista. Visitor-suunnittelumalli ei soveltunut SableCC:n tarpeisiin suoraan, koska se oli hieman liian rajoittunut. Niinpä sitä hieman laajennettiin ja nimettiin se uudelleen Switch-suunnittelumalliksi. Esimerkki Switch-suunnittelumallin käytöstä on esitetty kuvassa 3.1. Kuvaan on otettu mukaan Switch-suunnittelumallin kannalta tärkeimmät luokat ja metodit. Parser-luokka on kuvassa vain esimerkin vuoksi. Sen tilalla voisi olla mikä tahansa AST-puuta läpi käyvä luokka.



Kuva 3.1: Esimerkki Switch-suunnittelumallin käytöstä

Switch-suunnittelumallissa on kaksi rajapintaa, Switch ja Switchable. Switch-rajapinnasta on periytetty Analysis-luokka, jossa on metodit jokaista Switchable-luokasta perittyä luokkaa kohti. Nämä metodit nimetään laittamalla "case" luokan nimen eteen, esimerkiksi "caseAProgram". Switchable-rajapinnasta periytettyjen luokkien tulee toteuttaa apply(Switch s)-metodi, joka kutsuu oikeaa Analysis-luokan metodia. Esimerkiksi AProgram-luokan apply-metodi kutsuu Analysis-luokan caseAProgram-metodia. Tämä caseAProgram-metodi saa parametrikseen osoittimen kyseiseen AProgram-luokan olioon ja pystyy kutsumaan esimerkiksi sen lapsille apply-metodia. Tällä tavoin koko solmuhierarkia saadaan käytyä kätevästi läpi syvyyteen ensin haun periaatteella muokkaamalla ainoastaan yhtä Analysis-luokasta periytettyä luokkaa.

Analysisluokasta periytyy luokat AnalysisAdapter, DepthFirstAdapter sekä ReverseDepthFirstAdapter. AnalysisAdapter toteuttaa kaikille metodeille oletustoteutuksen, eli kaikkia metodeita ei välttämättä tarvitse enää toteuttaa aliluokissa. DepthFirstAdapter toteuttaa sitten itse puun läpikäynnin hierarkkisesti. Tämä näkyy seuraavasta esimerkistä.

```
public class DepthFirstAdapter extends AnalysisAdapter {
    public void inAProgram(AProgram node) {}
    public void outAProgram(AProgram node) {}
    public void caseAProgram(AProgram node) {
        inAProgram(node);
        for(PStatement e : node.getStatement()) {
            e.apply(this);
        }
        outAProgram(node);
    }
    public void inAAssignmentStatement(AAssignmentStatement node) {}
    public void outAAssignmentStatement(AAssignmentStatement node) {}
    public void caseAAssignmentStatement(AAssignmentStatement node) {
        inAAssignmentStatement(node);
        node.getIdentifier().apply(this);
        node.getExpression().apply(this);
        outAAssignmentStatement(node);
    }
    ...
}
```

Tässä nähdään, kuinka Program-solmun käsittelyfunktiossa (caseAProgram) ensin kutsutaan inAProgram-metodia, sitten käsitellään kaikki lapset (e.apply(this)) ja vielä viimeiseksi outAProgram-metodia. Näiden funktioiden avulla ohjelmalle saadaan helposti melko yksinkertainen rakenne. Suurimmassa osassa tilanteita vain outAXxx-tyyppiset metodit täytyy toteuttaa, ja muu hoituu automaattisesti. Funktiokutsu e.apply(this) osaa aina kutsua oikeaa caseAXxx-metodia, koska kaikis-

sa Switchable-luokan aliluokissa (eli AST-puun solmuluokissa) virtuaalinen apply-metodi korvataan toteutuksella, joka kutsuu kyseistä luokkaa vastaavaa caseAXxx-metodia:

```
public final class AAssignmentStatement extends PStatement
{
    private TIdentifier _identifier_;
    private PExpression _expression_;

    public void apply(Switch sw) {
        ((Analysis) sw).caseAAssignmentStatement(this);
    }
    ...
}
```

Tämän Switch-suunnittelumallin avulla on nyt helppo toteuttaa AST-puun läpikäyvät luokat. Ne periytetään DepthFirstAdapterista, ja niihin toteutetaan halutut metodit.

Huonona puolena tässä ratkaisussa on se, että luokkien toimintakuntoon saamiseksi ohjelmakoodia pitää kirjoittaa melko paljon. Onneksi SableCC auttaa tässä ja generoi suurimman osan ohjelmakoodista automaattisesti. Jäljelle jääkin ainoastaan omien DepthFirstAdapterista periytettyjen luokkien kirjoittaminen. Lisäksi, jos syntaksissa joudutaan muuttamaan jotain, täytyy vastaavat muutokset tehdä kaikkiin Analysis-luokasta perittyihin luokkiin. Näistä puutteista huolimatta Switch-suunnittelumalli on erittäin käyttökelpoinen, varsinkin juuri tähän tarkoitukseen.

3.3 Virheilmoitukset

Virheilmoituksia näytetään käyttäjälle silloin, jos ohjelman ajossa tapahtuu virhe, jota ohjelma ei itse pysty korjaamaan. Hyvän ohjelmiston tulisi pystyä mahdollisimman pitkälle korjaamaan virhetilanteet itse ja antaa käyttäjälle mahdollisimman vähän virheilmoituksia. Kääntäjissä asia on kuitenkin usein juuri päinvastoin. Mitä enemmän erilaisia virheilmoituksia ja varoituksia kääntäjä osaa näyttää käyttäjälle, sitä parempi, koska usein varoituksetkin ovat erittäin hyödyllisiä virheitä etsittäessä ja hyvään ohjelmointityyliin pyrittäessä.

Virheilmoitusta annettaessa tulisi varmistaa, että se täyttää seuraavat hyvän virheilmoituksen tunnusmerkit: [2]

- Virheilmoituksesta tulee käydä ilmi, mikä ohjelma tai ohjelmanosa antoi sen.
- Virheilmoitus tulee antaa oikeasta asiasta ja niin, että käyttäjä ymmärtää sen.
- Virheilmoituksen tulee olla täsmällinen niin, ettei sitä voi ymmärtää väärin.
- Virheilmoituksen tulee tarjota tietoa siitä, miten virheestä voi toipua ja mistä saa lisätietoa.

- Virheilmoituksen ei tule antaa käyttäjälle väärää, turhaa tai epätarkkaa tietoa.
- Virheilmoituksen ulkoasun ja tyylin tulee olla johdonmukainen muiden virheilmoitusten kanssa.
- Virheilmoitus tulisi voida näyttää käyttäjän haluamalla kielellä.
- Virheilmoituksen tulee erottua muista samankaltaisista virheilmoituksista esimerkiksi identifiointikoodin avulla. Toisaalta identifiointikoodin avulla virheilmoitukset saadaan tunnistettua vielä senkin jälkeen, jos ohjelmisto käännetään toiselle kielelle.
- Virheilmoitus tulisi kirjoittaa käyttäen kokonaisia virkkeitä.
- Virheilmoituksen tulisi antaa kansantajuisen tekstin lisäksi tarpeen mukaan ohjelmoijalle tarkempaa teknistä tietoa.
- Virheilmoituksen tulisi olla kohtelias eikä se saisi loukata käyttäjää.

Tyypillisimmät huonojen virheilmoitusten syyt:

- Ohjelmoija kirjoittaa virheilmoituksen lähinnä itselleen unohtaen, että käyttäjät, jotka virheilmoituksen saavat, eivät välttämättä ymmärrä ohjelmiston sisäistä toimintaa.
- Virheilmoitus on kirjoitettu kiireessä, eikä siinä ole tarpeeksi tietoa virheestä toipumiseksi.
- Virheilmoitus on jäänyt kirjoittamatta ja tarkan virheilmoituksen sijaan käyttäjälle näytetään jokin yleinen virheilmoitus.
- Kielen vaihtamisen jälkeen virheilmoituksesta on tullut käsittämätön.
- Tietokoneohjelmien tekeminen on vaikeaa, ja muiden ongelmien ohessa virheilmoitusten kirjoittaminen saattaa usein jäädä sivuseikaksi.

Ohjelmointikielen tulkin tai kääntäjän tapauksessa on tärkeää painottaa sitä, että virheilmoitus on riittävän täsmällinen ja tarkka. Jos virheilmoituksen pitää olla lyhyt, on parempi kertoa itse ongelma kuin yrittää selittää ratkaisua siihen. Usein ratkaisun löytäminen voikin olla huomattavasti vaikeampaa kuin pelkkä ongelman kuvaaminen. Ongelma pitäisi kuvata mahdollisimman tarkasti käyttämällä esimerkiksi muuttujien ja luokkien nimiä sen sijaan, että annettaisiin vain jokin yleinen ilmoitus. Myös rivinumero ja mahdollinen paikka rivillä ovat erittäin tärkeitä antaa, sillä jos kääntäjä tulkitsee ongelman väärin, rivinumero on ainut tieto, jonka avulla virheen voi löytää.

3.4 Lokalisointi

Lokalisointi eli paikallistaminen ja internationalisointi eli kansainvälistäminen tarkoittavat ohjelmiston kääntämistä ja sen sopeuttamista toiseen kieleen. Internationalisointi on ohjelmiston muokkaamista siten, että se on sen jälkeen pienellä vaivalla käännettävissä mille tahansa muulle kielelle. Lokalisointi on sitten kyseisen ohjelmiston kääntämistä tietylle kielelle.

Lokalisointi on aina eduksi käännettävälle ohjelmistolle, sillä ohjelmistojen käyttö on suurimmalle osalle käyttäjistä helpompaa omalla äidinkielellä. Lokalisointi alentaa sitä kynnystä, jonka uusien ohjelmien käytön oppiminen muodostaa. [4]

Lokalisointi ei ole vain pelkän kielen kääntämistä toiselle kielelle. Parhaimmillaan lokalisoinnissa mietitään kaikkia seuraavia asioita:

- kieli
- merkistöt, numerojärjestelmät, kirjoitusjärjestelmät, näppäimistöjen järjestykset
- päiväykset, kellonajat, aikavyöhykkeet
- paikalliset yhteystiedot, puhelinnumerot, osoitteet
- mittayksiköt, valuutat
- kuvalliset symbolit, kuten oikein/väärin merkit
- paikalliset säännöt ja tavat
- kulttuuriarvot ja sosiaalinen ympäristö.

Lokalisointi pitäisi ulottaa itse ohjelmiston lisäksi myös seuraaviin osa-alueisiin:

- käyttöliittymä
- ohjeet
- tavutus
- oikoluku
- WWW-sivut ja muut lisäpalvelut.

Ohjelmistoa lokalisoitaessa pitäisi pyrkiä antamaan yhdenmukaiset termit samoille asioille ohjelman eri osissa. Lisäksi alkuperäisen ja lokalisoidun tekstin pitäisi vastata toisiaan mahdollisimman tarkasti. Nämä kaikki asiat huomioon ottaen lokalisointi ei ole mikään läpihuutojuttu, vaan todella vie paljon aikaa.

Internationalisointi ja lokalisointi ovat hankalia tehdä ohjelmistoon jälkeenpäin. Ne onkin syytä ottaa huomioon jo ohjelmiston suunnittelussa. Yleensä tekstidata

ja muut ympäristöstä riippuvat resurssit erotetaan ohjelmakoodista. Tällöin lokalisointi onnistuu ihannetapauksessa erillisiä resursseja muokkaamalla ilman ohjelmakoodin muuttamista. Lokalisoidut versiot vaikeuttavat ylläpitoa. Jos esimerkiksi usealla kielellä käyttäjälle näytettävää viestiä halutaan muokata, pitää kaikki saman resurssin käännökset päivittää.

Tulkeissa ja kääntäjissä lokalisointi painottuu lähinnä kääntäjän antamien virheilmoitusten lokalisointiin. Kääntäjät eivät yleensä käsittele päiväyksiä eikä valuuttoja, joten niitä ei tarvitse lokalisoida. Jos kääntäjään liittyy graafinen käyttöliittymä, se täytyy tietenkin myös lokalisoida, ja siinä voikin olla jopa suurempi työ kuin pelkkien virheilmoitusten lokalisoinnissa. Mitä huolellisemmin ohjelma lokalisoidaan, sitä parempi on sen käytettävyys. [12]

4. TULKIN TOTEUTUS

Tässä luvussa kuvataan VIP 2.0:n tulkin rakenne pakkauksittain, kuvataan luokkien suhteita luokkakaavioiden avulla, esitetään tärkeimmät rajapinnat ja kuvataan, miten tulkin toteutus onnistui SableCC:n avulla. Lopuksi käydään tarkemmin läpi muutamia ominaisuuksia, kuten lokalisointi, annotaatiot ja valitsimet. Näistä ominaisuuksista kuvataan niiden toteutus ja käyttö.

4.1 Rakenne

4.1.1 Yleistä

VIP 2.0 on toteutettu Java 1.5:lla, joten versio 1.5 vaaditaan Javasta myös ohjelmaa ajettaessa. Javaa oli käytetty myös VIP 1.0:ssa ja sille on olemassa hyvät työkalut, joten se oli luonnollinen valinta.

VIP 2.0:n tulkki on toteutettu SableCC:n avulla. SableCC valittiin, koska se on tällä hetkellä Javalle saatavista jäsentäjägeneraattoreista monipuolisin. Se osaa muodostaa AST:n automaattisesti ja sen syntaksitiedosto on helppolukuista. Lisäksi SableCC on avoimen lähdekoodin ohjelmisto, joten se on täysin ilmainen ottaa käyttöön. SableCC on kuvattu tarkemmin aliluvussa 3.2.

Käyttöliittymä vanhaan VIP 1.0:aan oli toteutettu javasovelmana (Java applet). Harri Järvi on toteuttamassa uutta visuaalista käyttöliittymää, ja luultavasti sekin toteutetaan javasovelmana.

Javan luokat, SableCC:n syntaksitiedosto ja kommentit näihin on tehty englanniksi. Käyttöliittymä ja virheilmoitukset on lokalisoitu jo ainakin suomeksi ja englanniksi.

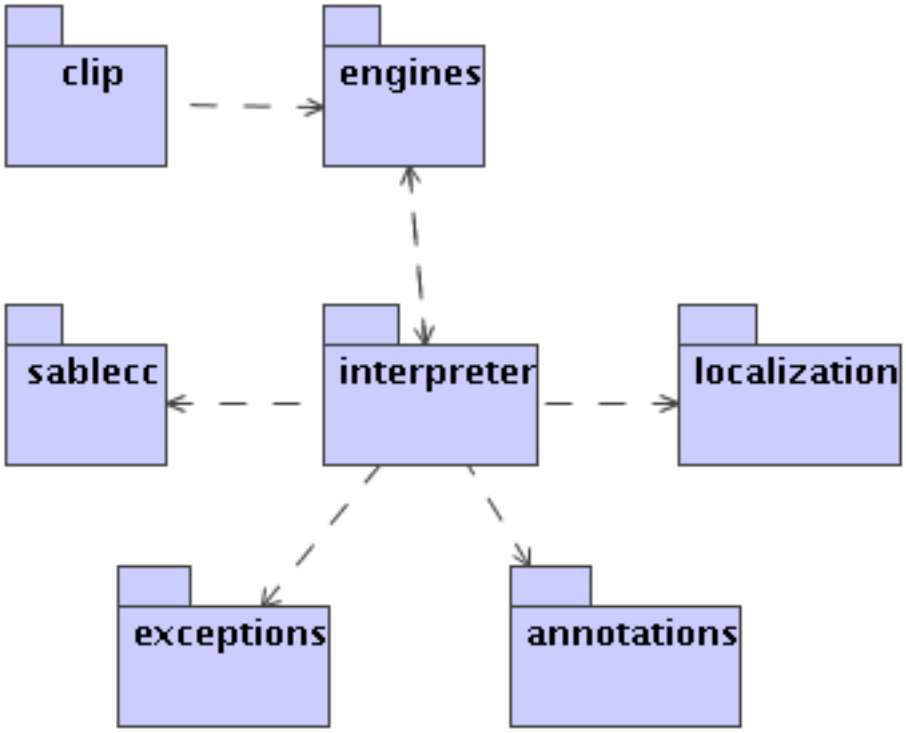
4.1.2 Luokkarakenne

Kuvassa 4.1 on esitelty tulkin rakenne yleisellä tasolla. Kuvassa näkyy kaikki tärkeimmät pakkaukset, joita VIP 2.0:n tulkissa on.

Tulkin luokkarakenne näkyy tarkemmin kuvasta 4.2, jossa on kuvattu kaikki keskeisimmät luokat komentorivikäyttöliittymän näkökulmasta. Keskeisimpiä luokkia kuvassa ovat Clip, ConsoleInterpreterEngine, SemanticAnalyzer sekä Interpreter. Kuvasta on jätetty pois joitain luokkia, jotka eivät ole kovin olennaisia. Tällaisia pois jätettyjä luokkia ovat ainakin Node-luokasta periytyneet luokat ja poikkeukset. Luokkakaavioista on jätetty pois myös joitain luokkien välisiä suhteita, jotka tekisivät kaaviosta vain hankalalukuisen. Tällaisia ovat ainakin luokkien Interpreter

ja SemanticAnalyzer suhteet. Nämä luokat ovat nimittäin yhteydessä miltei kaikkiin luokkakaavion luokkiin. Myös Variable-luokasta periytyneet luokat on jätetty kuvasta pois. Ne on kuvattu erillisessä luokkakaaviossaan (kuva 4.3).

Seuraavassa käydään vielä läpi kaikki pakkaukset. Pakkauksista on tässä esitetty vain yleiskuvaus. Tarkemmat tiedot löytyvät liitteestä 1.



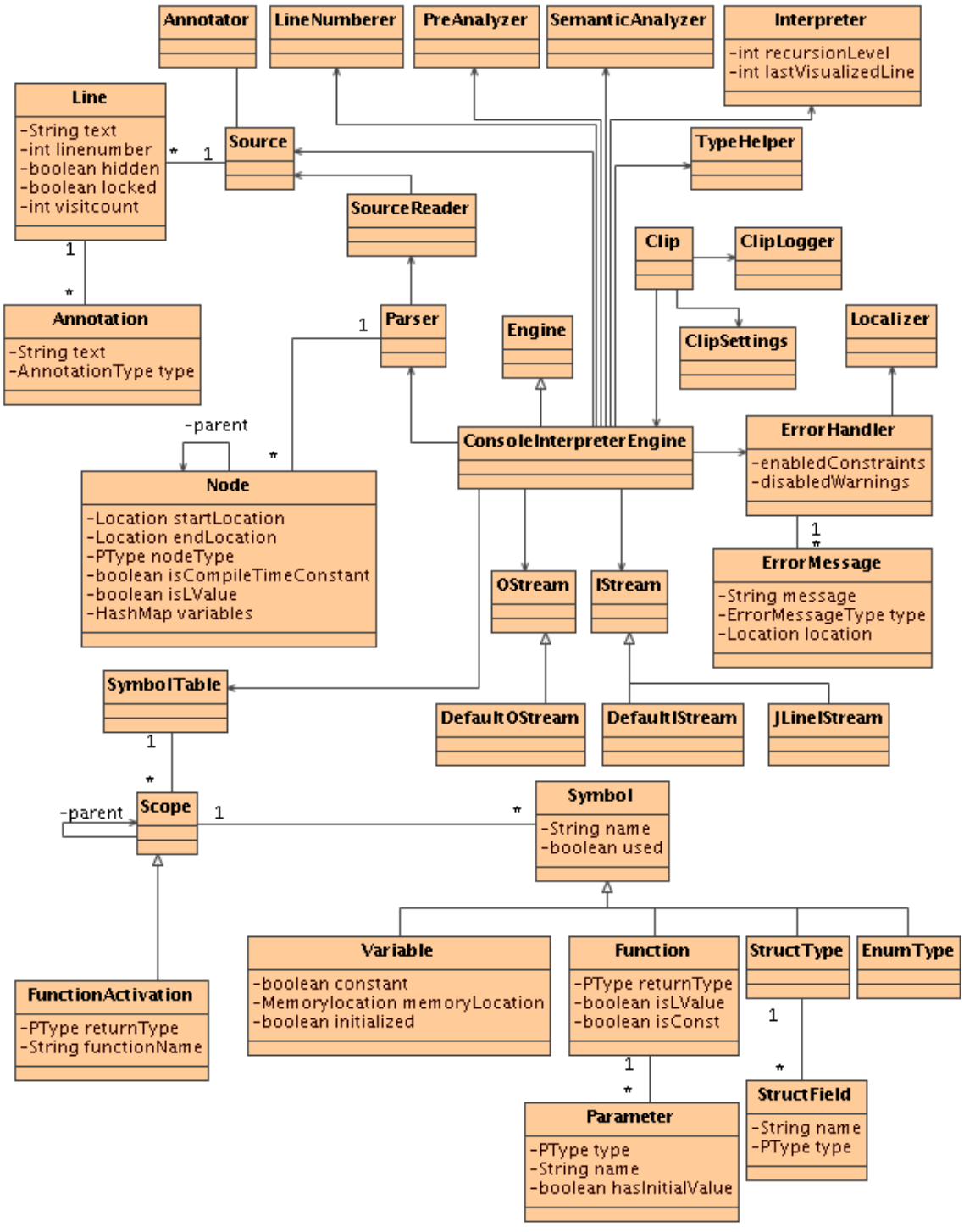
Kuva 4.1: VIP 2.0:n tulkin rakenne

Engines-pakkaus

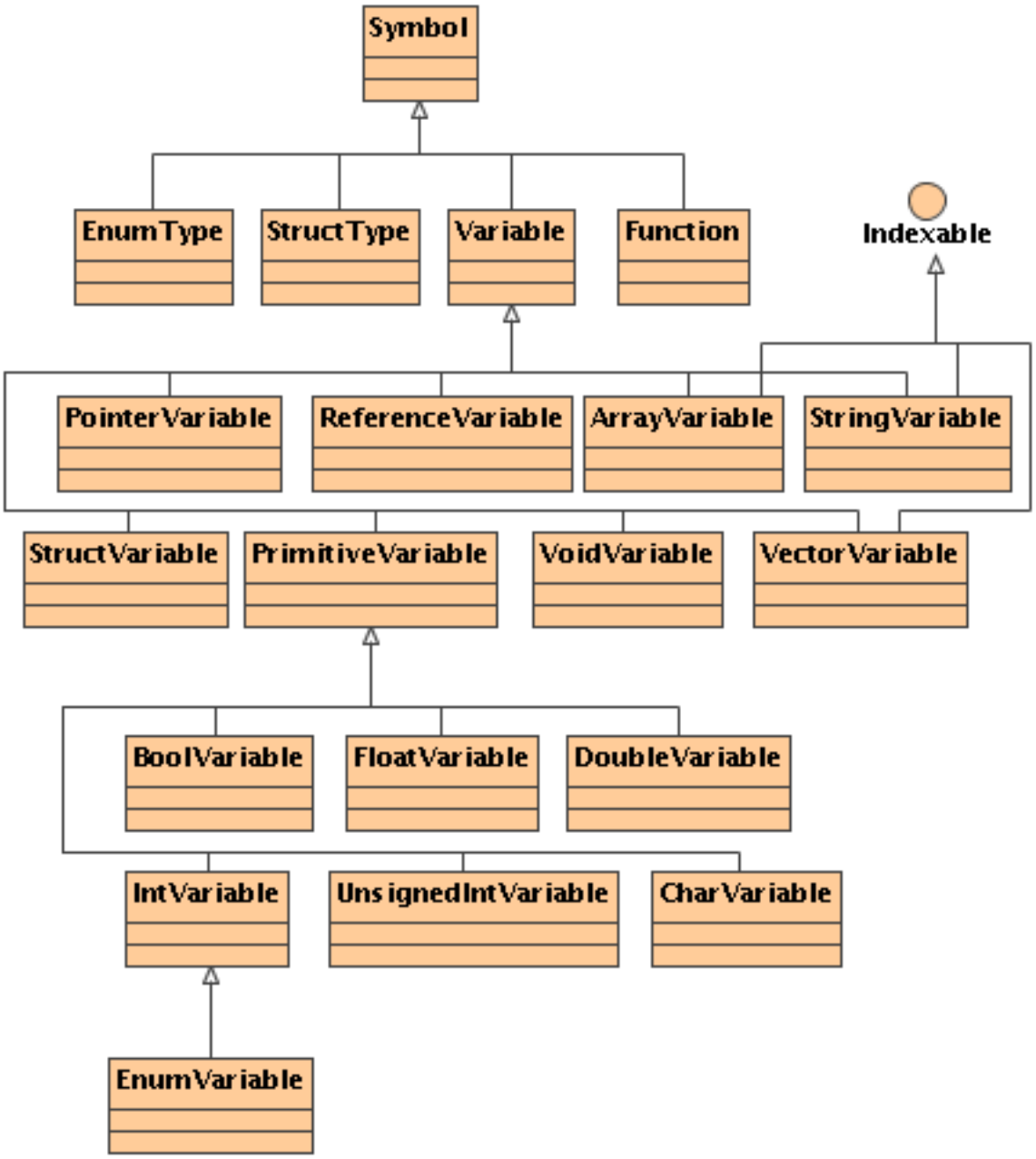
Engines-pakkaus sisältää moottoreita, joiden avulla tulkin voi käynnistää eri tiloihin. Valittavana on ainakin kaksi erilaista tilaa, joista toinen käynnistää tulkin komentorivikäyttöliittymään ja toinen suorittaa yhden tiedoston sisällön. Komentorivikäyttöliittymää on käsitelty tarkemmin aliluvussa 4.2.1.

Clip-pakkaus

Clip-pakkaus (Command Line InterPreter) sisältää komentorivikäyttöliittymään liittyviä luokkia.



Kuva 4.2: Tulkin keskeisimmät luokat



Kuva 4.3: Variable-luokkahierarkia

Interpreter-pakkaus

Interpreter-pakkaus sisältää itse tulkin ja kaikki sen sisäiset luokat, kuten symbolitaulun. Interpreter-pakkaus sisältää myös muutaman alipakkauksen. DFAs-alipakkaus sisältää SableCC:n DepthFirstAdapter-luokasta perittyjä luokkia. Nämä luokat sisältävät käsittelijät jokaiselle AST-puun solmulle. Streams-alipakkauksessa sisältää cin- ja cout-virrat ja niiden rajapinnat. Types-alipakkaus sisältää luokat luettelotyypeille ja tietuetyypeille. Variables-alipakkaukset sisältää luokat muuttujia varten.

Annotations-pakkaus

Annotations-pakkaus sisältää muutaman annotaatioihin liittyvän luokan. Annotaatioista lisää aliluvussa 4.3.2.

Exceptions-pakkaus

Exceptions-pakkaus sisältää pari poikkeusta, joita heitetään semanttisessa analyysissä ja ajon aikana. Poikkeukset tallennetaan ErrorHandleriin ja käsitellään kootusti Enginestä käsin. Kaikki nämä poikkeukset periytyvät RuntimeException-luokasta, jolloin niitä ei tarvitse erikseen esitellä kunkin metodin throws-osassa. Tämä on tärkeää, sillä poikkeuksia heitetään pääasiassa SableCC:n generoimista luokista periytetyistä luokista, joihin ei voi lisätä throws-osaa.

GUI-pakkaus

GUI (Graphical User Interface) -pakkaus tulee sisältämään graafisen käyttöliittymän. Graafinen käyttöliittymä ei kuulu tämän diplomityön aihealueeseen. Harri Järvi tulee käsittelemään sitä omassa diplomityössään.

Localization-pakkaus

Localization-pakkaus sisältää Localizer-luokan sekä properties-tiedostot, joiden avulla ohjelman lokalisointi on toteutettu. Nämä properties-tiedostot ovat resurssitiedostoja, joissa on kullekin eri kielelle käännetyt virheilmoitukset ja muut käyttöliittymän tekstit. Esimerkiksi messages_fi.properties-tiedosto sisältää suomeksi lokalisoidut tekstit. Localizer-luokka valitsee valitun luonnollisen kielen perusteella oikeankieliset virheilmoitukset näytettäväksi käyttäjälle. Lokalisointia käsitellään tarkemmin aliluvussa 4.2.4.

SableCC-pakkaus

SableCC-pakkaus sisältää kaikki SableCC:n generoimat luokat. Ne esiteltiin tarkemmin jo aliluvussa 3.2. SableCC:n generoimista luokista jouduttiin muuttamaan

ainoastaan Nodea. Siihen lisättiin kyseisen solmun alku- ja loppukohdat ohjelmakoodissa, solmun tyyppi, solmun arvo ohjelman ajon aikana sekä tietoa siitä, onko solmu vakio, l-arvo tai mahdollisesti for-silmukkamuuttuja. Näitä kaikkia arvoja tarvitaan semanttisessa analyysissä, kun vakiotarkistukset (const) tehdään kaikille solmuille, l-arvot tarkistetaan kaikissa sijoituslauseissa ja for-silmukkamuuttujien muuttaminen estetään for-silmukan sisällä.

Profiler- ja Analyzer-pakkaukset

VIP 1.0:ssa oli käytössä Profiler-pakkaus, joka seurasi, miten VIPiä käytettiin. Profiler lähetti lokiviestejä HTTP:n avulla CGI-ohjelmalle, joka tallensi viestit tiedostoon. Näitä tietoja voitiin myöhemmin analysoida Analyzer-paketissa olevien Java-luokkien ja Perl-skriptien avulla. VIP 2.0:aan näitä ei ole ainakaan vielä integroitu. CLIPissä on sen sijaan Logger-luokka, joka kirjaa käyttäjien syötteet lokitiedostoon (katso aliluku 4.2.1). Tämä ei tosin vielä toimi TTY:n tietotekniikan osaston Unix/Linux-ympäristön (Lintula) ulkopuolella.

4.1.3 Tyyppijärjestelmä

VIP 2.0:n tulkissa on kaksi eri tyyppijärjestelmää: käännösaikainen, joka generoidaan suoraan syntaksin avulla ja jonka perusteella tehdään semanttinen analyysi, sekä ajoaikainen, joka toimii kuvassa 4.3 sivulla 32 esitettyjen luokkien mukaan. Molemmat tyyppijärjestelmät on esitetty taulukossa 4.1.

Taulukko 4.1: Tyyppijärjestelmä

Tyyppi C++:ssa	**Käännösaikainen tyyppi**	**Ajonaikainen tyyppi**	**Ajonaikaisen tyypin sisäinen toteutus**
bool	ABoolType	BooleanVariable	Boolean
[unsigned|signed] char	ACharType	CharVariable	Character
[long] double	ADoubleType	DoubleVariable	Double
float	AFloatType	FloatVariable	Float
[signed] [short|long] [int]	AIntType	IntVariable	Integer
unsigned [short|long] [int]	AUnsignedIntType	UnsignedIntVariable	Long
string	AStringType	StringVariable	String
void	AVoidType	VoidVariable	-
<tyyppi>*	APointerType	PointerVariable	MemoryLocation
<tyyppi>&	AReferenceType	ReferenceVariable	Variable
const <tyyppi>	AConstType	-	-
vector< <tyyppi> >	AVectorType	VectorVariable	ArrayList<Variable>
<tyyppi>[]	AArrayType	ArrayVariable	ArrayList<Variable>
tunniste	AUserdefinedType	StructVariable tai EnumVariable	Hashmap<String, Variable> tai Integer, vastaavasti

Käännösaikainen tyyppijärjestelmä toimii siten, että SableCC generoi syntaksin perusteella PType-luokasta periytettyjä luokkia. Semanttinen analyysi tehdään sitten vertailemalla näitä tyyppejä. Kaikille AST-puun solmuille annetaan myös jokin näistä tyypeistä, jotta tyyppien selvittäminen olisi helpompaa. Taulukon 4.1 tyyppien lisäksi on olemassa tyyppi AFunctionType. Tämä tyyppi on siinä mielessä erilainen kuin muut, että SableCC ei generoi sitä, vaan se annetaan semanttisessa analyysissä niille AST-solmuille, jotka ovat funktioita.

Muunnos AUserdefinedType-tyypistä StructVariableksi tai EnumVariableksi tapahtuu symbolitaulun ja siellä olevien StructType- ja EnumType-tyyppien mukaan. Nämä tyypit, StructType ja EnumType, poikkeavat muista Type-loppuisista luokista siten, että ne ovat käyttäjän itsensä määrittelemiä tyyppejä, kun muut tyypit ovat olemassa jo ennen ohjelman kääntämistä.

AConstType on olemassa vain käännösaikana. Ajon aikana kukin Variablesta periytetty luokka sisältää tiedon siitä, onko kyseinen muuttuja vakio (const) vai ei. Tämä tieto on olemassa lähinnä visualisointia varten, koska kaikki semanttiset tyyppitarkastukset on tehty jo käännösaikana.

C--:ssa on pyritty tiukempaan tyyppijärjestelmään kuin mitä C++:ssa on. Esimerkiksi tyypin bool arvoa ei automaattisesti muunneta intiksi, vaan tyyppimuunnos on tehtävä static_castin avulla. Seuraavat tyyppimuunnokset tehdään automaattisesti:

- int ↔ unsigned int
- float ↔ double
- char → string
- [unsigned] int → float | double
- [unsigned] int → pointer
- string → char*
- enum → [unsigned] int
- <tyyppi> → const <tyyppi>
- <tyyppi>& → <tyyppi>
- <tyyppi>[ ] → <tyyppi>*

Esimerkiksi muunnos int ↔ unsigned int on sallittava, jotta esittelystä “unsigned int i = 5” ei tulisi tyyppimuunnosvirhettä.

4.1.4 Rajapinnat

VIP 2.0 on pyritty suunnittelemaan siten, että itse tulkkia pystyy käyttämään monen eri käyttöliittymän kautta. Tähän tarvitaan muutamia rajapintoja, jotka esitellään seuraavassa. Rajapinnat esitetään tässä vain pääpiirteittäin. Tarkemmat kuvaukset löytyvät liitteestä 2.

Engine-rajapinta

Engine-rajapinta toimii rajapintana tulkin ja käyttöliittymän välissä. Engine-rajapinnasta periytetyt luokat ovat moottoreita, joiden avulla tulkin voi käynnistää eri tiloihin. Engine-rajapinnan kautta käyttöliittymä ja muut ohjelman osat saavat osoittimet kaikkiin tärkeisiin tulkin osiin. Tällä hetkellä Engine-rajapinnan toteuttaa kaksi luokkaa, ConsoleFileEngine ja ConsoleInterpreterEngine. Ensimmäisen avulla voi ajaa tiedostoja kokonaisuudessaan kun taas jälkimmäinen toimii komentorivikäyttöliittymänä, joka on kuvattu aliluvussa 4.2.1.

IStream-rajapinta

IStream toimii rajapintana erityyppisille syötteenlukijoille. IStream-rajapinnan toteuttavien luokkien on toteutettava syötteen lukeminen annetusta virrasta sekä puskurointi. Puskurointia tarvitaan VIP in -virran toimimisen takia (katso aliluku 4.3.3).

IStream-rajapinnan toteuttaa kaksi luokkaa, DefaultIStream ja JLineIStream. DefaultIStream on oletustoteutus, joka lukee syötettä annetusta virrasta (esimerkiksi System.in:stä). JLineIStream on toteutus, joka käyttää JLine-kirjastoa syötteen lukemiseen. JLine tarjoaa sanojen ja tiedostonimien täydentämisen tabulaattorilla sekä komentohistorian. Nämä parantavat komentorivikäyttöliittymän käytettävyyttä. Tästä lisää aliluvussa 4.2.1.

OStream-rajapinta

OStream toimii rajapintana, kun tulkista tulostetaan eri kohteisiin. Sen toteuttaa kaksi luokkaa, DefaultOStream ja NullOStream. Näistä DefaultOStream on oletustoteutus, jonka avulla voi tulostaa kerralla moneen eri virtaan. NullOStream on versio, joka ei tulosta mitään mihinkään.

SymbolTableListener-rajapinta

VIP 1.0:sta peräisin oleva SymbolTableListener toimii rajapintana symbolitaulun ja visuaalisen käyttöliittymän välillä. Tällä hetkellä siinä on funktioita, joiden avulla symbolitaulu ilmoittaa käyttöliittymälle, kun esimerkiksi funktioita on kutsuttu, muuttuja esitelty tai muuttujan arvo muuttunut. Tämä rajapinta ei kuitenkaan vielä

ole käytössä ja saattaa muuttua, kun visuaalista käyttöliittymää ruvetaan tekemään. Myös muita rajapintoja saattaa vielä ilmaantua.

4.1.5 Tulkin toteutus SableCC:n avulla

Tulkki toteutettiin siis SableCC:n avulla. SableCC generoi ensin vip2.grammar-tiedostosta vastaavat luokat, kuten aliluvussa 3.2 kuvattiin. Näitä luokkia päästään käyttämään Parser- ja Lexer-luokkien avulla. Tulkki koostuu DepthFirstAdapter-luokasta periytetyistä luokista, jotka käyvät läpi AST-puuta solmu kerrallaan. Seuraavassa käydään läpi muutamia yksityiskohtia, joita kieliopin teossa tuli vastaan. Kokonaisuutena tiedosto vip2.grammar on esitetty liitteessä 3.

Tiedoston alussa on pitkä luettelo Unicode-merkkejä. Tässä luettelossa luetellaan kaikki ne Unicode-merkit, jotka ovat kirjaimia, eli joita voidaan käyttää ohjelmissa muuttujien nimissä. Toisessa luettelossa luetellaan vastaavasti kaikki numerot. Näiden luetteloiden avulla varmistetaan toisaalta, että Unicode-merkit toimivat, ja toisaalta, että kiellettyjä merkkejä ei muuttujien nimissä esiinny.

Annotaatioita ei käsitellä syntaksissa mitenkään. Koska annotaatiot alkavat merkeillä /*@ tai //@, ne ovat siis normaaleja "C++"-kommentteja tulkin mielestä. Annotaatiot jätettiin pois syntaksista, koska ne olisivat olleet hankalia jäsentää. Ne eivät nimittäin kuulu yhteen tiettyyn AST-puun solmuun, vaan koodiriviin. Annotaatiot, kuten muutkin kommentit, hylätään jäsennysvaiheessa eikä niitä oteta mukaan AST-puuhun.

Esikääntäjän komentoja ei eritellä, vaan kaikki #-alkuiset rivit tulkitaan komennoiksi, jotka voidaan suoraan ohittaa. Ne kuitenkin otetaan syntaksiin mukaan, jotta ne voidaan visualisoida ohjelmaa ajettaessa. Tällöin opiskelijoille voidaan näyttää annotaatioiden avulla, mitä kyseisillä riveillä tapahtuu, vaikka tulkin kannalta niillä ei tapahdukaan mitään.

Muutamia varattuja sanoja, kuten inline ja class, on esitelty ainoastaan siksi, että niiden käyttö voidaan estää muuttujien nimissä. Tämä on järkevää, jottei kukaan opi käyttämään varattuja sanoja sellaiseen tarkoitukseen, mihin niitä ei muulloin voi käyttää.

Syntaksiin jouduttiin lisäämään globaalille tasolle lauseet, jotka eivät ole "C++"-standardin mukaisia. Nämä lisättiin, jotta komentorivikäyttöliittymässä voitaisiin kirjoittaa lausekkeita suoraan komentokehotteeseen, kuten esimerkiksi "cout << i;". Näitä globaaleja lausekkeita voi kuitenkin käyttää ainoastaan komentokehotteessa. Osa laskulausekkeistakin toimii globaalilla tasolla. Esimerkiksi syöte "5;" komentokehotteessa tulostaa lausekkeen arvon, eli luvun 5. Kuitenkin esimerkiksi "5 * 6;" ei toimi. Tämä johtuu siitä, että syntaksiltaan samankaltainen lauseke "a * b;" tarkoittaa tässä tulkissa a-tyyppiin osoittavan osoittimen b esittelyä. Tällaiset laskulausekkeet voi tulostaa laittamalla ne sulkeisiin. Esimerkiksi "(5 * 6);" tulostaa 30.

Syntaksista on jätetty pois joitain rakenteita, joista ei ole aloittelevalle ohjelmoijalle kuin haittaa. Tällaisia rakenteita ovat esimerkiksi “a == b == c” ja “int a, b;”. Pilkku-operaattorin käyttö on estetty kaikkialla muualla paitsi for-lauseen kasvatusosassa.

Syntaksia tehdessä ongelmia aiheutti roikkuvan elsen ongelma (dangling else). Tämä ongelma näkyy seuraavassa esimerkissä.

```
if (i < 5)
  if (j < 10)
    cout << j;
 else
   cout << k;
```

Mistä nyt tiedetään, kuuluuko else-osa ensimmäiseen vai toiseen if-lauseeseen? Yleisesti on sovittu niin, että else-osa kuuluu jälkimmäiseen if-lauseeseen, joten tulkin pitäisi osata jäsentää se siten. Tämän takia syntaksiin oli määriteltävä rakenne, joka ei salli if-else-rakenteen ensimmäiseksi osaksi sellaista if-lausetta, jossa ei ole else-osaa. [5] Huomaa, että käyttämällä aaltosulkeita tällainen rakenne voidaan kuitenkin tehdä. Syntaksissa tämä rakenne näyttää yksinkertaistetusti seuraavalta:

```
statement =
    statement_without_trailing_substatement
    | if_then_statement | if_then_else_statement
    | while_statement | for_statement;

statement_no_short_if =
    statement_without_trailing_substatement
    | if_then_else_statement_no_short_if
    | while_statement_no_short_if
    | for_statement_no_short_if;

statement_without_trailing_substatement =
    statementseq | do_statement | ... (break, continue, return, cin, cout...);

if_then_statement = if_beginning statement;

if_then_else_statement = if_beginning statement_no_short_if kw_else statement;

if_then_else_statement_no_short_if = if_beginning statement_no_short_if
    kw_else statement_no_short_if;

if_beginning = kw_if tok_lpar conditional_expression tok_rpar;
```

Tästä havaitaan, että myös for- ja while-lauseille jouduttiin tekemään vastaavat versiot, jotka eivät salli lyhyttä if-lausetta. Muuten if-lauseen sisällä olevan for-lauseen sisällä oleva if-lause aiheuttaisi saman ongelman.

Syntaksi sisältää edelleen kommentoituna bittioperaattorit ^, |, &, ~, << ja >>. Tämä sen takia, että ne olisi tarpeen tullen helppo lisätä sinne. Kyseisten operaattoreiden käyttämisestä annetaan virheilmoitus, josta käy ilmi, että ne ovat kiellettyjä tässä tulkissa. Virhe annetaan, jos syntaksivirheen sisältävältä riviltä löytyy jokin näistä operaattoreista.

Tyyppien avainsanat (esimerkiksi int tai unsigned short int) on annettu selaajalle säännöllisinä lausekkeina sen sijaan, että ne olisi määritelty syntaksin puolella. Tämä sen takia, että tällä tavalla visualisoinnissa on helpompi laskea tyyppien avainsanojen alkamis- ja loppumiskohdat.

AST-puuhun jouduttiin lisäämään ylimääräisiä leksikaalialkioita, jotta jokaisen AST-puun solmun alkamiskohta ja loppukohta saataisin laskettua. Tämä jouduttiin tekemään, koska SableCC ei automaattisesti lisää solmuille alku- ja loppukohtia. Leksikaalialkioille nämä kuitenkin lasketaan. Alku- ja loppukohdat kaikille solmuille pystytään laskemaan, kunhan tiedetään solmun aloittavan leksikaalialkion alkukohta ja lopettavan leksikaalialkion loppukohta. Tämän laskemisen suorittaa LineNumberer-luokka.

AST-puun lisäksi tulkissa ei ole muuta välikieltä, vaan Interpreter-luokka tulkkaa suoraan AST-puuta. Yksi mahdollisuus olisi ollut toteuttaa linearisoitu välikieli, jonka tulkkaaminen olisi ollut hieman helpompaa kuin AST-puun tulkkaaminen. Tällöin vaikeus olisi kuitenkin siirtynyt siihen, miten AST-puu saadaan käännettyä linearisoiduksi välikieleksi. Kääntäjän tapauksessa tämä olisi ehkä ollut järkevää, mutta tulkissa, jossa AST-puun solmuja joudutaan visualisoimaan samalla kun niitä ajetaan, tämä olisi ollut huono ratkaisu.

4.1.6 Muutokset C--:een

Antti Virtanen esittää diplomityössään [13] vanhan VIP 1.0:n ja sen tukeman kielen, C--:n, ominaisuudet. Verrattuna VIP 1.0:aan syntaksi sisältää nyt paljon laajemman osan C++:aa kuin ennen. Seuraavassa luettelossa on lueteltu muutamia ominaisuuksia, joita “C--”-kieleen on lisätty.

- luettelotyypit (enum)
- taulukon alustus aaltosulkeilla
- tuki Unicodelle
- etuliiteoperaattorit ++i ja --i
- unsigned int sallittu
- switch-case-lause
- break- ja continue-avainsanat
- and-, or- ja not-avainsanojen käyttäminen mahdollista &&, || ja !:n lisäksi

- static_cast
- moniulotteiset vektorit.

Muita eroja ovat:

- Kielestä pystyy ottamaan joitain ominaisuuksia pois päältä (valitsimet, katso aliluku 4.3.1).
- Tietueet ja rekursio toimivat luotettavasti.
- Muutamia std:n funktioita on toteutettu, kuten abs, power ja isalpha.
- Tuki suuremmalle joukolle string- ja vector-luokkien metodeita.

4.2 Käyttöliittymä

4.2.1 Komentorivikäyttöliittymän toteutus

Komentorivikäyttöliittymä eli CLIP (Command Line InterPreter) on uutuus, jota VIP 1.0:ssa ei ollut. Sen avulla VIP 2.0:n tulkkia pystyy käyttämään suoraan komentoriviltä. CLIP on tehty ajatellen tulkkipohjaista lähestymistapaa ohjelmoinnin opetukseen [6]. CLIPissä ohjelmakoodia voi kirjoittaa suoraan komentoriville huolehtimatta kirjastojen mukaanottamisesta tai nimiavaruuksista.

Tulkkia voidaan käyttää komentoriviltä kahdessa eri tilassa. Jos tulkki käynnistetään suoraan komentoriviltä antamalla komento muodossa "clip <tiedosto>", tulkki käynnistyy tilaan, jossa se tekee semanttisen analyysin koodille ja sen onnistuessa ajaa tiedostosta löytyvän main-funktion. Tällöin interaktiivista komentorivitilaa ei käynnistetä ollenkaan. Jos tiedostoa ei anneta, eli jos tulkki käynnistetään pelkästään komennolla "clip", interaktiivinen tila käynnistyy. Seuraavassa tätä interaktiivista komentorivikäyttöliittymää kutsutaan CLIPiksi. Komento "clip" käynnistää itse asiassa skriptin, joka käynnistää CLIPin Javan avulla antaen sille tarvittavat komentoriviparametrit.

CLIPissä on käytössä JLine-kirjasto, joka tarjoaa käyttöliittymään sanojen täydentämisen ja komentohistorian. Komentohistoriassa voi liikkua eteen ja taaksepäin nuolinäppäimillä. Sanojen täydentäminen toimii siten, että tabulaattoria painamalla JLine näyttää kaikki mahdolliset tavat, miten sana voi jatkua. Jos sana voi jatkua vain yhdellä tavalla, sana jatketaan loppuun asti. Sanojen load, save ja saveall jälkeen CLIP siirtyy tilaan, jossa tiedostonimien ja hakemistojen täydentäminen on mahdollista. Tämä toiminnallisuus on samantapainen kuin esimerkiksi Bash-komentotulkin vastaava toiminnallisuus. Normaalissa tilassa CLIP osaa täydentää kaikki varatut sanat sekä symbolitaulussa olevien muuttujien ja funktioiden nimet.

Keskeisin luokka CLIPiä ajettaessa on ConsoleInterpreterEngine. Se ottaa syötteen vastaan, käsittelee erityiskomennot, suorittaa koodille semanttisen analyysin ja ajaa koodin. Jos syötetyssä koodissa on virheitä, CLIP näyttää ne käyttäjälle.

Keskeneräisen koodin tunnistaminen on toteutettu syntaksivirheen avulla. Jos annettu ohjelmakoodi on vajaa siten, että se ei muodosta vielä kelvollista "C++"-lausetta, CLIP osaa jäädä odottamaan lisää syötettä ennen kuin se yrittää ajaa koodin. ConsoleInterpreterEngine huomaa tämän siitä, että syntaksivirhe tulee koodia seuraavalta riviltä. Jos käyttäjä antaa tämän jälkeen tyhjän rivin, ohjelmakoodin lukeminen keskeytetään.

CLIPiin voi syöttää ohjelmakoodia kuin oltaisiin main-funktion sisällä. Käytännössä main-funktion sisällä ei olla, vaan kaikki syötetyt komennot tallennetaan globaalille näkyvyysalueelle. Tässä siis C-- on sallivampi kuin C++, jossa globaalit lauseet aiheuttaisivat jäsennysvirheen. Koska tulkki on koko ajan globaalilla näkyvyysalueella, funktioita ja tietueita voi esitellä ja käyttää aivan normaalisti.

CLIP tunnistaa seuraavat erikoiskomennot:

quit lopettaa tulkin käytön.

print tulostaa päätteelle tähän mennessä kirjoitetun ohjelmakoodin.

help tulostaa päätteelle ohjeita erikoiskomennoista ja tulkin käytöstä.

clear tyhjentää muistista kaikki muuttujat, funktiot ja tyypit.

show näyttää kaikki symbolitaulussa olevat muuttujat.

show <**regex**> näyttää kaikki ne muuttujat, joiden nimi sopii annettuun säännölliseen lausekkeeseen.

load <**tiedosto**> lukee tiedostossa olevan ohjelmakoodin.

save <**funktio**> <**tiedosto**> tallentaa funktion toteutuksen tiedostoon.

saveall <**tiedosto**> tallentaa kaiken syötetyn ohjelmakoodin tiedostoon.

run ajaa kaiken syötetyn ohjelmakoodin.

language näyttää kaikki käytettävissä olevat kielet.

language <**kieli**> vaihtaa käyttöliittymän kieltä.

Nämä erikoiskomennot voi kirjoittaa tulkkiin koska tahansa, kunhan ei kirjoita kyseiselle riville mitään muuta.

CLIPistä on poistettu muutamia varoituksia ja virheilmoituksia, jotka muuten olisivat häiritseviä tai joiden takia komentorivikäyttöliittymän käyttö ei onnistuisi lainkaan:

- Funktiota main ei tarvitse olla olemassa.
- Jos esiteltävä muuttuja tai funktio on jo olemassa, ei anneta virheilmoitusta vaan ainoastaan ilmoitus, jossa kerrotaan, että symbolin merkitys muuttui.

- Globaaleista muuttujista ei anneta varoitusta.
- Suoraan päätteelle kirjoitetut lauseet suoritetaan kuin ne olisivat pääohjelman sisällä.
- Jos muuttuja piilottaa aiemman muuttujan, varoitusta ei anneta.
- Tyhjästä lauseesta ei anneta varoitusta.
- Muuttujasta, jota ei ole käytetty, ei anneta varoitusta.
- Funktiosta, jota ei ole kutsuttu, ei anneta varoitusta.

Näistä kaikista annetaan siis virheilmoitus tai varoitus, jos tulkkia ajetaan visuaalisesta käyttöliittymästä tai suoraan komentoriviltä muodossa "clip <tiedosto>".

CLIP ei tulosta laskettujen lausekkeiden arvoja eikä näytä annotaatioita käyttäjälle mitenkään. Nämä ovat visuaalisen käyttöliittymän ominaisuuksia. Sen sijaan tulkki näyttää viimeisimmän syötetyn lausekkeen tyypin ja arvon. Esimerkiksi syöte "int i = -5; abs(i);" tulostaa funktion abs() paluuarvon ja sen tyypin.

CLIPiin on toteutettu UTF-8-koodaus (8-bit Unicode Transformation Format). Tämä tarkoittaa sitä, että myös käytettävän terminaalin merkistön koodaukseksi täytyy asettaa UTF-8. Muuten skandinaaviset merkit eivät toimi oikein. UTF-8:n ja Unicoden valinnalle oli monta syytä. Ensinnäkin SableCC:llä tehty syntaksitiedosto on määritelty tukemaan Unicodea. Toiseksi JLine-kirjasto käytännössä pakotti ottamaan käyttöön UTF-8:n. Kolmanneksi UTF-8 on hyvä valinta ajateltaessa järjestelmän lokalisoimista muille kielille. Näiden lisäksi Java tukee oletuksena Unicodea.

CLIP osaa rivittää tekstin useammalle riville siten, että mistään rivistä ei tule ylipitkä ja että teksti katkaistaan aina sanaväleistä. Lisäksi CLIPin käynnistysskripti selvittää käytetyn terminaalin leveyden, joten tekstin pitäisi rivittyä aina oikein. Tekstin rivitys tehdään DefaultOStream-luokassa.

CLIPissä on myös lokitoiminto, joka kirjaa kaiken ohjelmaan syötetyn tekstin lokitiedostoon. Lokitiedostoon kirjataan myös ohjelman käynnistys ja lopetus. Jokaisen lokiviestin yhteyteen laitetaan käyttäjän käyttäjätunnus sekä aikaleima. Poikkeuksen sattuessa myös poikkeuksen sisältö kirjataan. Tällä tavoin saadaan tietoa mahdollisista virheistä ja nähdään myös, mikä virheen aiheutti. Kirjaamisen suorittaa ClipLogger-luokka.

ClipSettings-luokka määrittää CLIPin asetukset. Se lukee asetuksia järjestyksessä seuraavista kohteista:

1. ympäristömuuttujat (voidaan asettaa komentoriviltä valitsimella -D)
2. clip.settings-tiedosto
3. oletusasetukset, jotka löytyvät ClipSettings-tiedostosta.

Määriteltäviä asetuksia ovat ainakin seuraavat:

- käytetäänkö lokitiedostoa
- lokitiedoston sijainti
- rivitys päälle ja pois sekä rivin maksimipituus
- käytettävät valitsimet
- varoitukset, jotka ohitetaan
- kieli
- rekursion maksimisyvyys
- käytetäänkö JLine-kirjastoa syötteen lukuun.

4.2.2 Visuaalisen käyttöliittymän toteutus

Visuaalinen käyttöliittymä ei varsinaisesti ollut tämän diplomityön aihealuetta. Sen toteuttamisesta vastaa Harri Järvi, ja se on vielä tällä hetkellä kesken. Seuraavassa on kuitenkin jo joitain asioita, joita visuaaliseen käyttöliittymään on tulossa, ja jotka siis piti ottaa huomioon jo tulkkia tehdessä.

Koodia pitää pystyä ajamaan rivi ja lauseke kerrallaan. Samalla pitää pystyä visualisoimaan käyttäjälle, missä mennään. Visualisoinnissa tulee olemaan toiminnot ohjelmakoodissa eteen- ja taaksepäin siirtymiseen, pysähtymiseen ja ohjelman ajon alusta aloittamiseen. Siirtyminen eteenpäin toteutetaan yksinkertaisesti antamalla tulkille lupa suorittaa seuraava askel. Taaksepäin siirtyminen toteutetaan luultavasti visualisoinnin puolelle siten, että tulkin kannalta taaksepäin ei siirrytä, ainoastaan selataan ohjelman tilaa edestakaisin.

Visualisoinnissa tulee olemaan ikkunat muuttujien arvojen näyttämiseen, laskulausekkeiden evaluoinnin ja tulosten näyttämiseen sekä ohje-ikkuna, jossa annotaatiot näytetään. Tulkissa on nyt jo tuki kaikille näille ominaisuuksille, vaikka mitään näistä ei näytetäkään käyttäjälle komentorivikäyttöliittymässä. Näiden toteutusta käydään tarkemmin läpi aliluvussa 4.3.

4.2.3 Virheilmoitukset VIP 2.0:n tulkissa

Uudessa tulkissa on 8 erilaista virheilmoitustyyppiä:

- leksikaalivirhe
- jäsennysvirhe
- semanttinen virhe
- varoitus
- huomautus

- rajoite
- ajonaikainen virhe
- muu virhe.

Kaikki nämä virheilmoitukset käsitellään ErrorHandlerissä. Näistä virheilmoituksista varoitus ja huomautus ovat siinä mielessä erilaisia, että niiden tapahduttua semanttista analyysiä vielä jatketaan. Muunlaisen virheen tapahduttua ohjelman ajo tai analysointi päätetään ja kaikki siihen mennessä huomatut virheet näytetään käyttäjälle.

SableCC antaa näistä virheilmoitustyypeistä leksikaali- ja jäsennysvirheitä. Semanttisia virheitä, varoituksia ja huomautuksia antaa SemanticAnalyzer. Se tarkistaa myös rajoitteet, jos niitä on asetettu. Ajonaikaiset virheet tunnistaa Interpreter-luokka. Muu virhe tarkoittaa tässä luettelossa esimerkiksi Javan NullPointerExceptionia. Se on luettelossa siksi, että sekin käsitellään ErrorHandlerissa ja kirjataan lokitiedostoon.

SableCC:n antamat virheilmoitukset

SableCC:n antamia virheilmoituksia muokataan ja lokalisoidaan. Esimerkiksi jäsennysvirheestä "[1,5] expecting: ';' " muokataan seuraavan näköinen virhe: "Jäsennysvirhe kohdassa [1, 5]: odotettiin jotain seuraavista: ';' ". Lisäksi virheilmoitukseen liitetään rivi, josta virhe tuli. Rivin alla näkyy osoitin, joka osoittaa, mistä kohtaa riviä virhe tuli. Tällä tavoin virheen löytäminen helpottuu.

Semanttiset virheet

Semanttisessa analyysissä annettavista virheistä on pyritty tekemään mahdollisimman helppolukuisia. Virheiden yksinkertaistamista auttaa rajoitetumpi syntaksi ja esimerkiksi mallien (template) poissaolo. Seuraavassa pari esimerkkiä, miten g++ (GNU C++ compiler) ja CLIP kääntävät saman ohjelman. Esimerkki 1:

```
struct A {
  int a;
}

int main() {
  return 0;
}
```

Tästä g++ antaa seuraavan virheilmoituksen:

```
a.cc:5: error: new types may not be defined in a return type
a.cc:5: note: (perhaps a semicolon is missing after the definition of 'A')
a.cc:5: error: two or more data types in declaration of 'main'
a.cc:5: error: '::main' must return 'int'
```

CLIP sen sijaan antaa seuraavan ilmoituksen:

```
Jäsennysvirhe kohdassa [5, 1]: odotettiin jotain seuraavista: ';'
int main() {
^
```

Esimerkki 2:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> i;
  cout << i;
  return 0;
}
```

Tästä ei g++ pidä, vaan antaa seuraavan ilmoituksen:

```
b.cc: In function 'int main()':
b.cc:8: error: no match for 'operator<<' in 'std::cout << i'
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ ostream.tcc:67: note: candidates are: std::basic_ostream<_CharT,
_Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(std::basic_ostream
<_CharT, _Traits>& (*) (std::basic_ostream<_CharT, _Traits>&))
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:78: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(std::basic_ios<_CharT,
_Traits>& (*)(std::basic_ios<_CharT, _Traits>&))
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:90: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(std::ios_base& (*)
(std::ios_base&)) [with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ ostream.tcc:241: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(long int) [with _CharT = char,
_Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:264: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(long unsigned int)
```

```
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:102: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(bool) [with _CharT = char,
_Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:125: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(short int) [with _CharT = char,
_Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/
ostream.tcc:157: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(short unsigned int)
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:183: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(int) [with _CharT = char,
_Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:215: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(unsigned int)
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:288: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(long long int)
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:311: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(long long unsigned int)
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:361: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(double) [with _CharT = char,
_Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:335: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(float) [with _CharT = char,
_Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:384: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(long double)
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
4.1.1/bits/ostream.tcc:407: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(const void*)
[with _CharT = char, _Traits = std::char_traits<char>]
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/../../../../include/c++/
```

```
4.1.1/bits/ostream.tcc:430: note: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(std::basic_streambuf<_CharT,
_Traits>*) [with _CharT = char, _Traits = std::char_traits<char>]
```

CLIP sen sijaan antaa hieman helpommin ymmärrettävän virheilmoituksen:

```
Semanttinen virhe rivillä 8: Tyypin "int-vektori" muuttujaa ei voi tulostaa
cout-virtaan.
  cout << i;
          ^
```

Ajonaikaiset virheet

CLIP huomaa myös sellaisia ajonaikaisia virheitä, jotka normaalissa kääntäjässä aiheuttaisivat ohjelman keskeyttämisen tai pysähtymisen. Tällaisia virheitä ovat:

- vektorin, taulukon tai stringin yli-indeksointi
- osoittimen siirtyminen taulukon rajojen yli
- liian syvä rekursio
- tyhjä osoitin
- alustamattoman muuttujan käyttö
- päättymätön silmukka
- funktiosta palautetaan viite tai osoitin funktion paikalliseen muuttujaan.

Kaikki tulkin antamat virheilmoitukset on lueteltu liitteessä 4. Lisäksi liitteessä 5 on CLIPin esimerkkiajo, josta näkyy, miten tulkki toimii käytännössä.

4.2.4 Lokalisointi VIP 2.0:ssa

Lokalisointi toteutettiin Javan luokkien java.util.ResourceBundle ja java.util.Locale avulla. Niitä käytetään siten, että ensin luodaan Locale-luokan olio, jolle annetaan parametreiksi kielikoodi, maakoodi ja kielivariantti, joista kaksi viimeisintä voidaan jättää pois. Tämän jälkeen luodaan ResourceBundle-olio, joka lataa lokalisoitavat merkkijonot tiedostosta, jonka nimi määräytyy käytettävästä localesta (kieliympäristö) esimerkiksi seuraavasti:

```
Locale fi = new Locale("fi", "FI", "windows");
ResourceBundle merkkijonot = ResourceBundle.getBundle("Tekstit", fi);
```

Yllä oleva ohjelmakoodi etsii nyt resursseja seuraavassa järjestyksessä olettaen, että järjestelmän oletuspaikka on en_US:

```
Tekstit_fi_FI_windows.java
Tekstit_fi_FI_windows.properties
Tekstit_fi_FI.java
Tekstit_fi_FI.properties
Tekstit_fi.java
Tekstit_fi.properties
Tekstit_en_US.java
Tekstit_en_US.properties
Tekstit_en.java
Tekstit_en.properties
Tekstit.java
Tekstit.properties
```

VIPissä lokalisoitavat merkkijonot sijaitsevat Localizer-luokan kanssa samassa hakemistossa, ja ovat nimeltään messages.properties, messages_fi.properties, messages_en.properties jne. Näissä resurssitiedostoissa on nyt lokalisoitavissa kaikki tulkin antamat virheilmoitukset sekä komentorivikäyttöliittymän tekstit. Tässä vaiheessa lokalisointi on toteutettu jo suomeksi ja englanniksi. Muita kielivaihtoehtoja ovat Etelä-Pohjanmaan murteelle käännetty versio sekä englanninkielinen IRC-simulaatio.

Resurssitiedoston muoto on yksinkertainen: <avain> = <arvo>. Tässä arvo voi sisältää myös muuttuvia osia, jotka merkitään numeroilla aaltosulkeissa, kuten seuraavassa esimerkissä on tehty:

```
error.invalidCast = Tyyppiä {0} ei voi muuttaa tyypiksi {1}.
```

Tämä virheilmoitus siis annetaan, kun yritetään luvatonta tyyppimuunnosta. Ilmoituksen lokalisoinnin yhteydessä ResourceBundle-luokalle annetaan parametreiksi nämä kaksi tyyppiä merkkijonoina ja ne liitetään virheilmoitukseen oikeille kohdilleen. Resurssitiedostot toimivat jopa siten, että jos jotain tiettyä virheilmoitusta ei löydy halutusta tiedostosta, sitä etsitään muista tiedostoista edellä olleen luettelon mukaisessa järjestyksessä. Suomenkielinen resurssitiedosto, joka sisältää kaikki suomeksi lokalisoidut tekstit ja virheilmoitukset, on liitteenä 4.

Liukulukutyyppien esitysmuoto lokalisoitiin ensin käytetyn kielen perusteella. Tämä havaittiin kuitenkin huonoksi ratkaisuksi, koska se vaihteli kieliasetusta muuttamalla eikä se ollut C++:n oletuksen mukainen. Niinpä liukulukutyypit tulostetaan nyt aina samalla tavoin, C++:n oletuksen mukaan.

Javan luokkien avulla saataisiin myös päivämäärät ja valuutat lokalisoitua käytetylle kielille, mutta koska tämä tulkki ei käsittele päivämääriä eikä valuuttoja, näitä ominaisuuksia ei tarvittu.

Käytettyä kieltä voi vaihtaa komentorivikäyttöliittymässä language-käskyllä. Käsky lukee tiedoston languages.txt, johon on kirjoitettu kaikki käytettävissä olevat kielet. Esimerkki languages.txt-tiedostosta:

```
english = en US
finnish = fi FI
etelapohjanmaa = fi FI etelapohjanmaa
irc = en US irc
```

Tässä esimerkiksi ensimmäinen rivi tarkoittaa, että komennolla "language english" luodaan uusi Locale-luokan olio parametreilla Locale("en", "US") ja etsitään sitä vastaava resurssitiedosto (tässä tapauksessa messages_en.properties). Jatkossa kaikki tekstit luetaan kyseisestä tiedostosta.

4.3 Muut ominaisuudet

4.3.1 Valitsimet

Valitsimien eli rajoitteiden avulla tulkin käyttöä pystytään rajoittamaan. Esimerkiksi osoittimet voidaan kytkeä pois päältä viitteiden käyttöä harjoiteltaessa, koska ne sekoittuvat helposti keskenään. Tällä hetkellä tulkissa on mahdollista kytkeä pois päältä seuraavat ominaisuudet:

- Valintalause (switch).
- Osoittimet.
- Silmukkamuuttujien muuttaminen for-silmukan sisällä
- Funktiot, joilla on sekä ei-vakioita parametreja että paluuarvo. Tällöin sallitaan vain sellaiset funktiot, jotka eivät muuta parametrejaan, sekä sellaiset proseduurit, jotka eivät palauta mitään arvoa, vaan ainoastaan muuttavat parametriensa arvoa.
- Globaalit muuttujat ja vakiot.
- Taulukot ja vektorit tietueiden kenttinä.

Tulkin kannalta rajoitteista tulevat virheilmoitukset ovat samalla tasolla kuin esimerkiksi tyyppimuunnoksesta tuleva semanttinen virhe, joten ohjelmakoodin semanttinen analysointi lopetetaan heti rajoitteen löydyttyä.

CLIPissä rajoitteita pystyy laittamaan päälle asettamalla clip.propertiesiin ominaisuuden clip.enabledConstraints arvoksi kyseisten rajoitteiden nimet pilkulla eroteltuna, esim:

```
clip.enabledConstraints = constraint.forVariables,constraint.pointers
```

Saman asian voi tehdä komentoriviltä antamalla Javalle valitsimen -D, esimerkiksi:

```
java -Dclip.enabledConstraints=constraint.pointers
     -classpath vip2.jar:lib/jline-0.9.91.jar
     fi.tut.cs.vip2.clip.Clip
```

Tällöin esimerkiksi komento “int* a = 0;” aiheuttaa virheen, joka kertoo, että osoittimet eivät ole sallittuja.

4.3.2 Annotaatiot

Annotaatiot jakautuvat ohjeteksteihin ja ohjauskäskyihin. Annotaatioiden tehtävää kuvattiin jo aliluvussa 2.3.2. Annotator-luokka lukee lähdekoodin läpi etsien sieltä annotaatioita monirivisessä (/*@ */) ja yksirivisessä (//@) muodossa. Annotator liittää ohjetekstit seuraavaan ei-tyhjään riviin, jolla on jokin ohjelman lause. Ohjetekstejä ei siis liitetä seuraavaan ohjelman lauseeseen, vaan riviin. Ohjauskäskyt sen sijaan vaikuttavat moneen riviin. Tulkin kannalta annotaatiot ovat kommentteja, eikä se käsittele niitä mitenkään erityisesti.

Ohjetekstit

Ohjetekstejä on monentyyppisiä:

/*@ ohje */ näytetään jokaisella kerralla tultaessa kyseiselle riville

/*@even ohje */ näytetään parillisilla kerroilla

/*@odd ohje */ näytetään parittomilla kerroilla

/*@6 ohje */ näytetään kuudennella kerralla (voi olla myös mikä tahansa muu numero)

/*@n ohje */ näytetään, jos mitään muuta ohjetta ei näytetä kyseisellä rivillä

/*@init ohje */ näytetään ennen ohjelman ajon alkamista.

Piilotettu ja lukittu ohjelmakoodi

Koodia voidaan piilottaa antamalla käsky /*@hide on*/, jonka jälkeen tuleva ohjelmakoodi on piilossa käyttäjältä. Tähän ohjelmakoodiin voidaan lisätä esimerkiksi tarkastuskoodia. Koodi saadaan taas näkyviin käskyllä /*@hide off*/. Lukitus toimii vastaavalla tavalla käskyjen /*@lock on*/ ja /*@lock off*/ avulla. Lukittua ohjelmakoodia ei voi muokata visuaalisessa käyttöliittymässä. Kaikki nämä voidaan antaa myös yksirivisen kommentin avulla, esimerkiksi “//@lock on” tai “//@even ohje”.

Seuraavassa esimerkki annotaatioiden käytöstä. Tämä esimerkki on harjoitus, jonka opiskelija voi tehdä ja tarkastaa visuaalisen käyttöliittymän avulla. Komentorivikäyttöliittymässä annotaatioita ei huomioida mitenkään, joten kyseisestä esimerkistä ei ole mitään iloa CLIPissä.

```
//@lock on
//@hide on
/*@init Tee ohjelma, joka laskee potenssilaskun silmukkarakenteen avulla.
 Muista tehdä ohjelmaan tarkistus, että syötetty eksponentti on
 positiivinen. */
//@hide off
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
  //@hide on
  //syötetään dataa cin-virran puskuriin
  __vip_in << 5 << 4;
  //@hide off
  //@ esitellään double-tyyppinen muuttuja, jonka nimi on kantaluku
  double kantaluku = 0;
  //@ esitellään kokonaisluku
  int eksponentti = 0;
  cout << "Syötä laskettavan potenssilaskun kantaluku ja eksponentti: ";
  //@ kysytään kantaluku
  cin >> kantaluku;
  //@ kysytään eksponentti
  cin >> eksponentti;
  double tulos = 0;
  //@lock off

  // Kirjoita ratkaisusi tähän, muista eksponentin tarkistus.

  //@lock on
  cout << kantaluku << "^" << eksponentti << " = " << tulos << endl;
  //@hide on
  if( tulos != 625.0 ) { // 5 ^ 4
     __vip << "Ohjelmasi ei tunnu laskevan tulosta oikein." << endl;
  }
  else {
     __vip << "Ohjelmasi vaikuttaa toimivan oikein, hyvä!" << endl;
  }
  //@hide off
  return EXIT_SUCCESS;
}
//@lock off
```

4.3.3 VIP-virrat

VIP-virtojen käytöstä tulikin jo esimerkki äskeisessä aliluvussa (aliluku 4.3.2). Siinä main-funktion kolmannelta riviltä näkyy, miten VIP in -virran avulla dataa voidaan syöttää cin-virran puskuriin. VIP out -virran avulla tarkastuksen tulos voidaan tulostaa opiskelijalle. VIP-virtojen toimintaa kuvattiin jo aliluvussa 2.3.2.

VIP-virrat on toteutettu IStream- ja OStream-rajapintojen ja niistä periytettyjen luokkien avulla. Jokainen IStream-luokasta peritty luokka toteuttaa puskurin, johon voidaan laittaa dataa esimerkiksi VIP in -virran avulla. VIP out -virta voi olla mikä tahansa OStream-rajapinnan toteuttava luokka, johon siis voidaan kirjoittaa mitä tahansa tekstiä, kuten muihinkin tulostusvirtoihin. Nämä rajapinnat on selitetty tarkemmin aliluvussa 4.1.4.

Erikoisfunktio __vip_in_ok() palauttaa arvon true, jos cin-virran puskuri on tyhjä, eli jos kaikki VIP in -virtaan lisätty data on luettu. Tämän avulla voi tarkistaa, että opiskelija käyttää cin-virtaa oletetusti.

4.4 Testaus

VIP 2.0:n tulkin testaamiseksi tehtiin luokka AutomaticTester, jonka toimintaperiaate on se, että se suorittaa tulkin avulla kaikki tests-hakemiston alihakemistoissa olevat .cc-päätteiset tiedostot. Jos tiedostoa ei ole ennestään ajettu, se kysyy käyttäjältä syötteen ja tallentaa sen .in-päätteiseen tiedostoon. Samalla se tallentaa ohjelman tulosteen .out-päätteiseen tiedostoon. Jatkossa tiedosto voidaan ajaa kysymättä käyttäjältä mitään. Tällöin tulostusta verrataan .out-tiedostoon ja ilmoitetaan, jos sisältö on muuttunut. Tällä tavoin sekä testien lisääminen että kaikkien testien ajaminen on helppoa.

4.5 Toteutustyökalut

VIP 2.0:n tulkki toteutettiin Javan versiolla 5.0. Ohjelmointiympäristönä käytettiin Eclipsen versiota 3.2. Versionhallintana käytettiin SVN:ää (Subversion), joten Eclipseen asennettiin Subclipse-plugin SVN:n käyttämiseksi. VIP 2.0:aa ja CLIPiä käännettäessä apuun otettiin Apache Ant:n versio 1.7.0. Sen avulla voidaan laittaa myös SableCC generoimaan AST:n luokat kielioppitiedoston perusteella. Näitä SableCC:llä generoituja luokkia ei ole nimittäin laitettu SVN:ään.

5. YHTEENVETO

Tässä diplomityössä toteutettiin tulkki ohjelmoinnin opetuksen tarpeisiin. Visualisoinnin ja tulkkauksen takia tulkki ei ole kovin tehokas, mutta se ei ollut tavoitteenakaan. Tulkki tulkkaa C++:n osajoukkoa, nimeltään C--, joka sisältää kaikki kielen perusrakenteet, mutta ei esimerkiksi luokkia eikä nimiavaruuksia. Tulkkia pystyy käyttämään komentorivikäyttöliittymän avulla. Tulkkiin on tulossa myös visuaalinen käyttöliittymä.

Tulkin toteutuksessa painotettiin lokalisointia ja hyviä virheilmoituksia. Lokalisointi uudelle kielelle onnistuu kääntämällä erilliseen resurssitiedostoon kootut virheilmoitukset uudelle kielelle. C++:aa yksinkertaisempi syntaksi mahdollistaa selkeämmät virheilmoitukset esimerkiksi käytettäessä vector- ja string-luokkia. Virheilmoituksista on pyritty tekemään sellaiset, että myös aloitteleva ohjelmoija ymmärtäisi ne. Lisäksi virheilmoitusten lokalisointi omalle äidinkielelle helpottaa niiden ymmärtämistä.

Tämä uusi tulkki on merkittävä uudistus verrattuna vanhaan VIP 1.0:n tulkkiin. Tulkki on myös välttämätön työkalu ajatellen tulkkipohjaista lähestymistapaa ohjelmoinnin opetukseen, jota professori Hannu-Matti Järvinen käsittelee vielä julkaisemattomassa kirjassaan [6]. Tulkki avaa uusia mahdollisuuksia C++:n käyttöön, sillä aiemmin ei ole ollut vapaasti saatavilla lokalisoitavaa “C++”-tulkkia, jossa olisi painotettu selkeitä virheilmoituksia.

Tässä diplomityössä käsiteltiin myös tulkkien ja kääntäjien teoriaa sekä esiteltiin käytetyn jäsentäjägeneraattorin, SableCC:n, toimintaa. Virheilmoituksia ja lokalisointia käsiteltiin myös yleiseltä kannalta.

Kun visuaalinenkin puoli valmistuu, linkki toimivaan versioon tulee löytymään ainakin EDGE-ryhmän sivuilta (http://www.cs.tut.fi/∼edge/). Komentorivikäyttöliittymää voi jo nyt käyttää TTY:llä Lintulan koneilta ja toivon mukaan pian sen ulkopuoleltakin komennolla “clip”. Komentorivikäyttöliittymää ei ole vielä otettu käyttöön, mutta oletuksena on, että kun se otetaan käyttöön, opiskelijat oppivat C++:n perusteet nopeammin kuin ennen. VIP 2.0 tullaan julkaisemaan vapaana ohjelmistona GPL-lisenssillä.

Metrics-työkalulla mitattuna lähdekoodin pituus on tällä hetkellä 38871 riviä. Luokkia on 341 ja pakkauksia 20. Luvuissa on mukana SableCC:llä generoidut luokat, joita on aika paljon. Itse kirjoitettua lähdekoodiakin on kuitenkin melko paljon.

5.1 Työn arviointi

Mielestäni CLIP onnistui hyvin, ja sitä mieltä ovat olleet myös sitä kaikki sitä kokeilleet henkilöt. Erityisesti sen selkeistä virheilmoituksista on pidetty. Virheilmoitukset ovatkin olleet suurin syy, miksi tätä tulkkia on ylipäätään ruvettu tekemään. Verrattuna VIP 1.0:aan ero on merkittävä. Vanhassa VIPissä monien esimerkkien ajamisessa oli ongelmia eivätkä ne aina toimineet niinkuin olisi pitänyt. Uudessa VIPissä vastaavia ongelmia ei pitäisi olla juuri lainkaan. Tosin sitä ei ole vielä testattu opiskelijakäytössä. Mahdollisiin ongelmiin on kuitenkin varauduttu ja niitä tullaan korjaamaan niiden ilmaantuessa. Uusi VIP on nyt paremman suunnittelun ja käytettyjen toteutustekniikoiden ansioista helpommin jatkokehitettävissä kuin vanha VIP.

Parannettavaa tulkissa voisi olla luokkien SemanticAnalyzer ja Interpreter toteutuksessa. Osan instanceof-kyselyistä olisi ehkä voinut poistaa käyttämällä Variable-luokkahierarkiassa enemmän polymorfismia. Näistä kahdesta luokasta, SemanticAnalyzer ja Interpreter, tuli myös erittäin pitkiä (yli 2000 riviä). Tähän tosin vaikuttaa suuresti SableCC:n käyttämä Switch-suunnittelumalli (aliluku 3.2.3) ja sen aiheuttamat suuret luokat. Joka tapauksessa nämä luokat ovat kuitenkin kohtuullisen helppolukuisia verrattuna esimerkiksi VIP 1.0:n vastaaviin rakenteisiin.

Tulkissa on vielä joitain virheitä. Esimerkiksi moniulotteisen vektorin ja lokiin kirjaamisen kanssa on vielä joitain ongelmia. Näitä virheitä tullaan korjaamaan tulevien viikkojen aikana.

Aikaa tähän diplomityöhön kului melkein vuosi (lokakuu 2006 - elokuu 2007). Tosin osan tästä ajasta olin töissä vain osa-aikaisesti. Työtunteja kertyi kuitenkin noin 1300. Se on hieman pitkä aika diplomityölle, mutta projektin pituus tiedettiin jo työhön ryhdyttäessä. Työ valmistui kuitenkin määräaikaansa mennessä (elokuu 2007).

Itse tulkin tekeminen sujui loppujen lopuksi melko hyvin, vaikka aluksi se vaikutti melko haastavalta. Virtasen Antin avustuksella pääsin kohtalaisen helposti alkuun, jonka jälkeen työ onkin ollut aika itsenäistä. Työtä tehdessäni opin paljon uusia asioita kääntäjien ja tulkkien suunnittelusta. Työ oli mielekästä, koska tiesin koko ajan, mihin tarkoitukseen ohjelmani tulee ja että sitä tullaan oikeasti käyttämäänkin.

5.2 Jatkokehitysajatuksia

Vaikka VIP 2.0:sta on tällä hetkellä valmiina vain komentorivikäyttöliittymä, esitetään tässä myös visuaaliseen puoleen ja koko järjestelmään liittyviä jatkokehitysajatuksia. Nämä graafiseen puoleen liittyvät ideat ovat jatkokehitysajatuksia ajatellen lähinnä VIP 1.0:n nykyistä toiminnallisuutta.

- Visuaaliseen käyttöliittymään voisi tehdä mahdolliseksi asettaa ohjelmakoodiin kehittyneempiä pysähdyskohtia (breakpoints). Näiden avulla ohjelmakoo-

dia pystyisi ajamaan uudestaan täsmälleen tiettyyn tilanteeseen huomioiden rekursiotasot ja silmukoiden läpikäynnit.

- Visualisoinnissa varatut sanat voisi lihavoida ja editorissa voisi muutenkin olla Eclipsestä tuttuja ominaisuuksia, kuten sanojen täydentäminen, ohjelmakoodin sisentäminen, funktioon siirtyminen sen nimeä painamalla yms.
- Komentorivikäyttöliittymässä UTF-8-koodauksesta pitäisi päästä eroon, tai ainakin koodaus pitäisi saada vaihtumaan automaattisesti ISO 8859-1:stä UTF-8:ksi tulkkia käynnistettäessä. Skandinaaviset merkit eivät nimittäin näy oletuksena oikein.
- Loppujen C++:n ominaisuuksien lisääminen tulkkiin.
- 'Näytönsäästäjän' toteuttaminen, joka alkaisi ajaa tulkissa satunnaista ohjelmakoodia ja näyttämään ASCII-grafiikkaa tulkin oltua käyttämättömänä esimerkiksi 10 minuuttia.
- Tulkin käyttöajan rajoittaminen esimerkiksi 5 tuntiin päivässä (liian innokkaille opiskelijoille, jotta heillä olisi muutakin elämää).
- Tulkkiin jotain taustamusiikkia, esimerkiksi Martti Servo ja Napander - Mikä on kun ei taidot riitä.

LÄHTEET

[1] Appel A. 2002: Modern Compiler Implementation in Java, 2nd edition, 501 s.

[2] Bolton, M.: Writing Error Messages, http://www.klariti.com/technical-writing/writing-error-messages.shtml (viitattu 10.7.2007)

[3] Gagnon, E. 1998: SableCC, an Object-Oriented Compiler Framework, http://sablecc.sourceforge.net/downloads/thesis.pdf, 107 s. (viitattu 22.10.2007)

[4] Grönroos, M. 2003: Avoimen lähdekoodin ohjelmistojen lokalisoinnin tila, http://www.lokalisointi.org/files/floss-lokalisoinnin-tila-2003.pdf (viitattu 16.7.2007)

[5] Iosif R., The Dangling Else problem, http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node101.html (viitattu 20.6.2007)

[6] Järvinen H.-M., Ohjelmoinnin alkeiden opetus tulkkia käyttäen, 2005 - 2007, julkaisematon.

[7] Järvinen H.-M., Tiusanen M. ja Virtanen A., 2003: Convit, a Tool for Learning Concurrent Programming, www.cs.tut.fi/∼edge/convit_paperi.pdf, 4s. (viitattu 19.9.2007)

[8] Kölling, M.: The problem of teaching object-oriented programming, part 1: Languages, *Journal of Object-Oriented Programming, 11(8):8-15*, January 1999

[9] Luoma, H. 1988: Minkälainen on kysymys?, n. 40 s.

[10] Robins A., Rountree J. ja Rountree N., Learning and teaching programming: A review and discussion, *Computer Science Education, 12(2):137-172)*, 2003.

[11] Stroustrup, B. 2000: C++ -ohjelmointi, 3. painos, 942s.

[12] Vesterholm, M. ja Kyppö, J. 2006: Java-ohjelmointi, 566 s.

[13] Virtanen, A., 2005: Visuaalinen tulkki ohjelmoinnin opetukseen, http://www.cs.tut.fi/∼vip/masters_thesis26042005.pdf, 51 s. (viitattu 7.6.2007)

[14] Virtanen, A., Lahtinen, E. ja Järvinen, H.-M. 2005: VIP, a Visual Interpreter for Learning Introductory Programming with C++, http://www.cs.tut.fi/∼codewitz/vip_paperi.pdf, 6s. (viitattu 19.9.2007)

[15] Winslow L., Programming pedagogy — a psychological overview, *SIGCSE Bulletin, 28(3)*, September 1996.

LIITE 1: LUOKKARAKENNE

Tämä liite sisältää kuvaukset kaikista tärkeimmistä tulkin luokista ryhmiteltynä pakkauksittain.

1.1 Engines-pakkaus

Engines-pakkaus sisältää moottoreita, joiden avulla tulkin voi käynnistää eri tiloihin.

ConsoleFileEngine tarjoaa keinon ajaa kokonaisia tiedostoja komentoriviltä.

ConsoleInterpreterEngine on komentorivipohjainen käyttöliittymä tulkkiin. Sitä on käsitelty tarkemmin aliluvussa 4.2.1.

Engine-rajapinta on puolestaan kuvattu tarkemmin aliluvussa 4.1.4 ja liitteessä 2.

1.2 Clip-pakkaus

Clip sisältää Main-funktion komentorivikäyttöliittymän käynnistämiseen.

ClipLogger on luokka, joka kirjoittaa tiedostoon kaiken komentorivikäyttöliittymässä syötetyn tekstin.

ClipSettings on luokka, joka hakee komentorivikäyttöliittymän asetukset tiedostosta clip.properties. Tämä luokka huomaa myös, jos asetuksia on annettu komentoriviparametreina ohjelmalle, jolloin niitä käytetään tiedostossa olevien asetusten sijaan.

1.3 Interpreter-pakkaus

ErrorHandler huolehtii tulkissa tapahtuvien virheiden käsittelystä. Kun ohjelmakoodissa havaitaan virhe (esimerkiksi muuttuja, jota ei ole esitelty), ErrorHandler tallentaa sen. Jos kyseessä oli vain varoitus, ohjelmakoodi käydään läpi loppuun asti ja etsitään muita virheitä. Tämän jälkeen ErrorHandlerilta kysytään löydetyt virheet ja varoitukset ja ne tulostetaan käyttäjälle. Jos löydetään virhe, semanttinen analyysi ja ohjelmakoodin suoritus keskeytetään välittömästi.

ErrorMessage on yksi virhe tai varoitus, joka ohjelmakoodista löytyi. ErrorMessage tietää, mistä kohdasta ohjelmakoodia virhe tuli ja mistä virheestä on kyse.

ErrorMessageType on ErrorMessagen tyyppi (enum). Mahdollisia vaihtoehtoja ovat selaaja-, jäsentäjä-, semanttinen, ajonaikainen ja muu virhe sekä varoitus, info ja rajoite.

Function kuvaa yhtä funktiota “C--”-koodissa.

FunctionActivation luodaan jokaista funktion kutsukertaa varten. Näistä muodostetaan aktivaatiopino symbolitauluun.

Line on yksi "C--"-koodin rivi.

Location on tietty paikka tietyllä rivillä.

Parameter on funktion parametri. Jokaisella funktiolla (Function) on tietty määrä parametreja (Parameter), jotka on lueteltu kyseisen funktion määrittelyssä. Parametrilla voi olla oletusarvo.

Scope vastaa yhtä näkyvyysaluetta ohjelmakoodissa. Näkyvyysalue on yleensä merkkien { ja } välinen alue. Poikkeus tähän on funktio, jonka parametrit oletetaan olevan samalla näkyvyysalueella kuin itse funktion toteutuskoodikin. Näkyvyysalueella määritellyt muuttujat peittävät aikaisemmin määritellyt samannimiset muuttujat, ja ne tuhotaan näkyvyysalueen lopussa. Samalla näkyvyysalueella ei voi olla kahta samannimistä muuttujaa, kuten normaalia.

Source sisältää "C--"-koodin.

SourceReader periytyy Javan Reader-luokasta. Sitä tarvitaan, kun luetaan lähdekoodia Source-luokasta ja syötetään se SableCC:lle.

STDFunctions sisältää kasan C++:n kirjastofunktioita std-nimiavaruudesta, muun muassa funktiot toupper, tolower, abs, sqrt ja isalpha.

Symbol on yliluokka kaikille olioille, joita symbolitaulu pitää sisällään. Symbolista periytyvät luokat on esitetty kuvassa 4.3.

SymbolTable pitää sisällään kaikki näkyvyysalueet ja niillä määritellyt muuttujat mukaanlukien globaalit vakiot, muuttujat, funktiot ja tyyppimäärittelyt. Sisältää myös näkyvyysalueiden avulla toteutetun aktivaatiopinon.

Näiden luokkien lisäksi Interpreter-pakkaus sisältää seuraavat alipakkaukset:

DFAs-pakkaus sisältää SableCC:n DepthFirstAdapter-luokasta perittyjä luokkia. Nämä luokat sisältävät käsittelijät jokaiselle AST-puun solmulle. Tämä pakkaus sisältää seuraavat luokat:

Line numberer laskee kaikille solmuille niiden alku- ja loppukohdat ohjelmakoodissa. Tämä täytyi tehdä, koska SableCC ei tarjonnut suoraan tukea solmujen alku- ja loppukohdille. Tämä aiheutti myös sen, että AST-puun produktioihin jouduttiin lisäämään joitakin sellaisia leksikaalialkioita, joita ei olisi muuten tarvittu, kuten esimerkiksi sulkeita tai puolipisteitä.

PreAnalyzer käsittelee tietueet ja funktiot ja lisää ne symbolitauluun ennen semanttista analyysiä. Tätä ominaisuutta tarvittiin, jotta funktioita ja tietueita voisi käyttää ohjelmakoodissa ennen kuin ne on määritelty. Tässä

siis C-- on sallivampi kuin C++. PreAnalyzer ei ole varsinainen ohjelmakoodin esikäsittelijä. Koodin esikäsittelyä (makroja yms.) tässä tulkissa ei ole.

SemanticAnalyzer tekee ohjelmakoodille varsinaisen semanttisen analyysin, johon kuuluu muun muassa tyyppitarkastelut ja muutenkin komentojen järkevyys. Semanttista analyysiä käsiteltiin tarkemmin aliluvussa 3.1.2.

Interpreter ajaa ohjelmakoodin olettaen, että sille on jo tehty semanttinen analyysi. Tämä on tärkein luokka ohjelmakoodia ajettaessa.

Streams-pakkaus sisältää cin- ja cout-virrat ja niiden rajapinnat.

IStream on rajapinta virroille, joista luetaan. Tästä periytyy tällä hetkellä ainakin DefaultIStream, joka tarjoaa mahdollisuuden lukea käyttäjän syötteitä päätteeltä, sekä JLineIStream, joka käyttää JLine-kirjastoa syötteen lukemisessa. J-Line-kirjasto mahdollistaa sanojen täydentämisen tabulaattorista ja komentohistorian.

OStream on vastaavasti rajapinta virroille, joihin kirjoitetaan. Tästä periytyy DefaultOStream, jonka avulla pystyy tulostamaan päätteelle sekä myös esimerkiksi tiedostoon. NullOStream on OStreamin toteutus, joka ei tulosta mitään mihinkään.

Types-pakkaus sisältää luokat luettelotyypeille (enum) ja tietuetyypeille (struct). Näitä tyyppejä ei pidä sekoittaa sablecc.node-pakkauksessa oleviin tyyppeihin, sillä ne kaikki tyypit ovat jo tiedossa ennen ohjelmaa. Nämä luettelotyypit ja tietueet ovat olemassa vasta kun ne määritellään. SableCC antaa näiden tyyppisille muuttujille tyypiksi AUserdefinedType. Tulkissa tyyppi muutetaan sitten tietyn tyyppiseksi EnumTypeksi tai StructTypeksi.

EnumType on luokka luettelotyyppejä varten.

StructType on luokka tietuetyyppejä varten.

StructField on luokka tietueen kenttiä varten. Jokainen tietuetyyppi pitää sisällään ne StructField-kentät, jotka tietueen alustuksessa on sille määritelty.

TypeHelper avustaa tyyppien muuntamiseksi toisikseen. Luokassa on funktioita automaattisia tyyppimuunnoksia varten sekä apufunktioita solmujen tyyppien vertaamiseksi ja selvittämiseksi.

Variables-pakkaus sisältää luokat muuttujia varten. Variables-luokkien luokkahierarkia on esitetty kuvassa 4.3.

ArrayVariable on taulukko, jonka alkioina voi olla mitä tahansa tyyppejä. Taulukko on siinä mielessä hieman keinotekoinen tyyppi, että oikeasti C++:ssa taulukot ovat vain osoittimia muistialueisiin. Tässä tulkissa

kuitenkin halutaan vahvempi tyyppijärjestelmä, joten taulukoita varten tarvitaan oma luokka. Taulukko tietää myös oman kokonsa, joten yli-indeksoinnit voidaan tarkistaa ajon aikana.

BoolVariable on bool-tyyppinen muuttuja "C--"-koodissa.

CharVariable on char-tyyppinen muuttuja. VIP 2.0:n tulkki tukee Unicodea ja merkkejä voi esitellä myös laittamalla niiden Unicode-heksakoodin \u:n jälkeen, esimerkiksi \u0040. Kaikki merkit, kuten skandinaaviset merkit (ä, ö), toimivat myös funktioiden ja muuttujien nimissä.

DoubleVariable on double-tyypin muuttuja (kaksoistarkkuuden liukuluku). Liukulukutyyppien tulostusulkoasu on C++:n standardin mukainen.

FloatVariable on float-tyypin muuttuja (liukuluku).

Indexable on rajapinta, jonka StringVariable, ArrayVariable ja VectorVariable toteuttavat. Se tarjoaa funktiot tietyllä indeksillä olevan muuttujan hakemiseksi ja muuttamiseksi.

IntVariable vastaa C-- -koodissa int-tyyppiä (kokonaislukua).

EnumVariable on jonkin EnumType-olion tyyppinen muuttuja. Se periytyy IntVariablesta, koska sillä on samat ominaisuudet kuin IntVariablella. EnumVariable-tyyppinen muuttuja luodaan "C--"-koodissa näin:

```
enum Kuukausi { TAMMIKUU, HELMIKUU };
Kuukausi k = TAMMIKUU;
```

Tässä luodaan EnumType-muuttuja, jonka nimeksi annetaan Kuukausi. Sen jälkeen luodaan kaksi EnumVariable-muuttujaa, joiden nimeksi annetaan TAMMIKUU ja HELMIKUU ja jotka ovat vakioita, arvoiltaan 0 ja 1. Tämän jälkeen luodaan vielä yksi EnumVariable-muuttuja nimeltään k, ja annetaan sille alkuarvo 0 (TAMMIKUU).

MemoryLocation kuvaa yhtä osoitetta muistissa. Kullakin muuttujalla on yksi tällainen MemoryLocation.

PointerVariable kuvaa osoitinta ja sen arvona on aina jokin MemoryLocation. Osoitinta voidaan siirtää "++"- ja "--"-operaattoreilla, jolloin se tarkistaa, ettei osoiteta taulukon rajojen ulkopuolelle.

PrimitiveVariable on kantaluokka kaikille C--:n perustietotyypeille, joita ovat bool, char, double, float, int ja unsigned int. Niillä voidaan suorittaa aritmeettisia operaatioita ja ne voidaan muuttaa toisikseen. PrimitiveVariable-luokasta periytyvät luokat näkyvät kuvasta 4.3.

ReferenceVariable on viite johonkin toiseen muuttujaan.

StringVariable on C++:n string.

StructVariable on jonkin StructType-olion tyyppinen muuttuja.

UnsignedIntVariable on unsigned int -tyypin muuttuja. Se on toteutettu Long-tyyppisen muuttujan avulla, koska Javassa ei ole unsigned-muuttujia. Sen arvoalue palautetaan aina jokaisen siihen kohdistuvan laskuoperaation jälkeen välille $[0..2^{32} - 1]$.

Variable on kaikkien muuttujien kantaluokka. Se tarjoaa myös muutamia apufunktioita muun muassa muuttujien tyyppien selvittämiseen.

VariableFactory tarjoaa funktioita muuttujien luomiseen.

VectorVariable on vektorityyppi (vector<type>). Sisätyyppinä voi olla mikä tahansa tyyppi, jopa toinen vektori. Tällä tavalla voi luoda moniulotteisia vektoreita. Moniulotteiset vektorit ovat sallittuja C--:ssa.

VoidVariable on tyyppiä void. Mukana lähinnä sen takia, että esimerkiksi funktioista voidaan palauttaa void-tyyppinen arvo. Tämä tarkoittaa C++:ssa sitä, että tällöin funktio ei palauta arvoa lainkaan. Tulkin toteutuksen kannalta funktio kuitenkin palauttaa muuttujan, jonka tyyppi on void.

1.4 Annotations-pakkaus

Annotations-pakkaus sisältää muutaman annotaatioihin (aliluku 4.3.2) liittyvän luokan:

Annotation sisältää yhden kommentin, joka voi olla yksi- tai monirivinen. Tällainen kommentti on lähdekoodissa määritelty merkkien /*@ ja */ välissä tai merkin //@ jälkeen. Ohjauskäskyjä (hide on, lock on) ei kuitenkaan tallenneta Annotationeiksi, vaan ne käsitellään eri tavalla. Annotaatio näytetään visualisoinnissa, kun ollaan ensimmäisellä annotaatiota seuraavalla ei-tyhjällä rivillä.

AnnotationType on enum, joka sisältää kaikki mahdolliset ohjetekstityypit, joita ovat Always, Init, Even, Odd, Nth ja Other.

Annotator lukee lähdekoodin läpi ja etsii sieltä kaikki annotaatiot liittäen ne oikeisiin riveihin. Annotaatiot liitetään siis aina riveihin, ei AST-solmuihin.

1.5 Exceptions-pakkaus

Exceptions-pakkaus sisältää seuraavat RuntimeException-luokasta periytyvät luokat:

VIPException on kantaluokka muille VIPin poikkeuksille.

SemanticException heitetään, kun ohjelmakoodista löydetään semanttinen virhe.

InterpretException heitetään, kun ohjelmakoodia suoritettaessa tapahtuu virhe.

LIITE 2: RAJAPINNAT

2.1 Engine-rajapinta

public Source getSource() palauttaa lähdekoodin, joka sisältää koodin riveittäin.

public Interpreter getInterpreter() palauttaa viitteen Interpreter-luokan olioon eli tulkkiin.

public SemanticAnalyzer getSemanticAnalyzer() palauttaa viitteen semanttiseen analysaattoriin.

public SymbolTable getSymbolTable() palauttaa viitteen symbolitauluun.

public ErrorHandler getErrorHandler() palauttaa viitteen virhekäsittelijään.

public TypeHelper getTypeHelper() palauttaa viitteen luokkaan, joka tarjoaa apufunktioita tyyppien selvittämiseen ja tarkistamiseen.

public IStream getCin() palauttaa viitteen virtaan, josta syöte luetaan.

public OStream getCout() palauttaa viitteen virtaan, jonne tulostus ohjataan.

public OStream getVipOut() palauttaa viitteen virtaan, jonne VIP out -virtaan tulostetut tekstit ohjataan.

public OStream getEvalOut() palauttaa viitteen virtaan, jonne kirjoitetaan laskulausekkeet ja niiden tulokset.

public OStream getAnnotationOut() palauttaa viitteen virtaan, jonne annotaatiot kirjoitetaan.

public int getMaxLineLength() palauttaa rivin maksimipituuden, tai arvon 0, jos rivitys ei ole käytössä.

2.2 IStream-rajapinta

public String readWord() lukee yhden sanan virrasta.

public String readLine() lukee yhden rivin virrasta.

public char readChar() lukee yhden merkin virrasta.

public long readLong() lukee yhden kokonaisluvun virrasta.

public double readDouble() lukee yhden liukuluvun virrasta.

public void bufferPrint(String msg) kirjoittaa parametrina annetun merkkijonon puskuriin, josta se luetaan seuraavalla read-käskyllä.

public void bufferPrint(char c) kirjoittaa merkin puskuriin.

public void bufferPrint(long l) kirjoittaa kokonaisluvun puskuriin.

public void bufferPrint(double d) kirjoittaa liukuluvun puskuriin.

public void setPrompt(String s) asettaa kehotteen. Tämä toimii JLineIStream-luokalle, joka osaa tulostaa kehotteen uudelleen esimerkiksi tabulaattorin painamisen jälkeen.

public void addCompletionString(String s) lisää täydennettävän merkkijonon. Lisäyksen jälkeen merkkijonon voi täydentää painamalla tabulaattoria. Toimii JLineIStream-luokalle.

public boolean isBufferEmpty() palauttaa arvon true, jos puskuri on tyhjä.

2.3 OStream-rajapinta

public void print(String msg) tulostaa annetun merkkijonon virtaan.

public void print(long l) tulostaa kokonaisluvun virtaan.

public void print(char c) tulostaa merkin virtaan.

public void print(double f) tulostaa liukuluvun virtaan.

public void println(String msg) tulostaa annetun merkkijonon sekä rivinlopetusmerkin virtaan. Vastaavasti on olemassa println(long l), println(char c) sekä println(double f).

public void flush() tyhjentää virran.

public void close() sulkee virran.

public void printNewLineIfNotEmpty() tulostaa virtaan rivinvaihdon, jos tällä hetkellä rivi ei ole tyhjä, eli jos rivinvaihtoa ei ole juuri tulostettu.

public void setMark() asettaa merkin. Liittyy seuraavaan funktioon.

public boolean somethingPrintedAfterMark() palauttaa arvon true, jos virtaan on tulostettu jotain merkin asettamisen jälkeen.

public void setMaxLineLength(int amount) asettaa rivityksen päälle. Tämän jälkeen kaikki virtaan tulostetut merkkijonot rivitetään siten, että niiden pituus ei ylitä parametrina annettua rivin maksimipituutta. Oletustoteutus rivittää tekstin siten, että se pyrkii katkaisemaan rivit sanaväleistä.

LIITE 3: C-- SYNTAKSI

```
/*
 * Visual Interpreter 2 syntax
 * This interpreter aims to
 *  - be subset of C++
 *  - perform the program like C++ would
 *  - give clear, informative, simple and useful error messages for C++ beginners
 *  - be more strongly typed than C++
 *
 * Interpreter aims not to be very efficient.
 *
 * Differences to C++:
 *  - Operators ?: and , are missing
 *  - Bitwise operators << >> | ^ & |= ^= and != are missing
 *  - No preprocessor
 *  - Casting possible only with static_cast (no dynamic_cast,
 *    reinterpret_cast, const_cast, (type)variable or type(variable))
 *  - No member pointer operators ->* and .*
 *  - No new/delete
 *  - No function pointers
 *  - No multidimensional arrays (possible with vectors)
 *  - No overloading of functions
 *  - No classes, and structs can't have methods
 *  - No templates, file streams, exceptions or namespaces
 *  - The following keywords are not implemented: goto, sizeof,
 *    typedef, union, inline, auto, extern, register, static, volatile,
 *    asm, wchar_t, mutable
 *
 *  + Global statements allowed for interpreter functionality
 *  + Supports unicode
 */

Package fi.tut.cs.vip2.sablecc;

/*
 * Helpers
 */
Helpers

    all = [0 .. 0xffff];
    cr = 13;
    lf = 10;
    tab = 9;
    eol = cr | lf | cr lf;
    not_eol = [all - [cr + lf]];
    not_star = [all - '*'];
    not_star_slash = [not_star - '/'];
    empty = eol | tab | ' ';

    // all unicode characters which are letters
    unicode_letter =
```

```
[0x0041..0x005a] | [0x0061..0x007a] | [0x00aa..0x00aa] | [0x00b5..0x00b5] |
[0x00ba..0x00ba] | [0x00c0..0x00d6] | [0x00d8..0x00f6] | [0x00f8..0x01f5] |
[0x01fa..0x0217] | [0x0250..0x02a8] | [0x02b0..0x02b8] | [0x02bb..0x02c1] |
[0x02d0..0x02d1] | [0x02e0..0x02e4] | [0x037a..0x037a] | [0x0386..0x0386] |
[0x0388..0x038a] | [0x038c..0x038c] | [0x038e..0x03a1] | [0x03a3..0x03ce] |
[0x03d0..0x03d6] | [0x03da..0x03da] | [0x03dc..0x03dc] | [0x03de..0x03de] |
[0x03e0..0x03e0] | [0x03e2..0x03f3] | [0x0401..0x040c] | [0x040e..0x044f] |
[0x0451..0x045c] | [0x045e..0x0481] | [0x0490..0x04c4] | [0x04c7..0x04c8] |
[0x04cb..0x04cc] | [0x04d0..0x04eb] | [0x04ee..0x04f5] | [0x04f8..0x04f9] |
[0x0531..0x0556] | [0x0559..0x0559] | [0x0561..0x0587] | [0x05d0..0x05ea] |
[0x05f0..0x05f2] | [0x0621..0x063a] | [0x0640..0x064a] | [0x0671..0x06b7] |
[0x06ba..0x06be] | [0x06c0..0x06ce] | [0x06d0..0x06d3] | [0x06d5..0x06d5] |
[0x06e5..0x06e6] | [0x0905..0x0939] | [0x093d..0x093d] | [0x0958..0x0961] |
[0x0985..0x098c] | [0x098f..0x0990] | [0x0993..0x09a8] | [0x09aa..0x09b0] |
[0x09b2..0x09b2] | [0x09b6..0x09b9] | [0x09dc..0x09dd] | [0x09df..0x09e1] |
[0x09f0..0x09f1] | [0x0a05..0x0a0a] | [0x0a0f..0x0a10] | [0x0a13..0x0a28] |
[0x0a2a..0x0a30] | [0x0a32..0x0a33] | [0x0a35..0x0a36] | [0x0a38..0x0a39] |
[0x0a59..0x0a5c] | [0x0a5e..0x0a5e] | [0x0a72..0x0a74] | [0x0a85..0x0a8b] |
[0x0a8d..0x0a8d] | [0x0a8f..0x0a91] | [0x0a93..0x0aa8] | [0x0aaa..0x0ab0] |
[0x0ab2..0x0ab3] | [0x0ab5..0x0ab9] | [0x0abd..0x0abd] | [0x0ae0..0x0ae0] |
[0x0b05..0x0b0c] | [0x0b0f..0x0b10] | [0x0b13..0x0b28] | [0x0b2a..0x0b30] |
[0x0b32..0x0b33] | [0x0b36..0x0b39] | [0x0b3d..0x0b3d] | [0x0b5c..0x0b5d] |
[0x0b5f..0x0b61] | [0x0b85..0x0b8a] | [0x0b8e..0x0b90] | [0x0b92..0x0b95] |
[0x0b99..0x0b9a] | [0x0b9c..0x0b9c] | [0x0b9e..0x0b9f] | [0x0ba3..0x0ba4] |
[0x0ba8..0x0baa] | [0x0bae..0x0bb5] | [0x0bb7..0x0bb9] | [0x0c05..0x0c0c] |
[0x0c0e..0x0c10] | [0x0c12..0x0c28] | [0x0c2a..0x0c33] | [0x0c35..0x0c39] |
[0x0c60..0x0c61] | [0x0c85..0x0c8c] | [0x0c8e..0x0c90] | [0x0c92..0x0ca8] |
[0x0caa..0x0cb3] | [0x0cb5..0x0cb9] | [0x0cde..0x0cde] | [0x0ce0..0x0ce1] |
[0x0d05..0x0d0c] | [0x0d0e..0x0d10] | [0x0d12..0x0d28] | [0x0d2a..0x0d39] |
[0x0d60..0x0d61] | [0x0e01..0x0e2e] | [0x0e30..0x0e30] | [0x0e32..0x0e33] |
[0x0e40..0x0e46] | [0x0e81..0x0e82] | [0x0e84..0x0e84] | [0x0e87..0x0e88] |
[0x0e8a..0x0e8a] | [0x0e8d..0x0e8d] | [0x0e94..0x0e97] | [0x0e99..0x0e9f] |
[0x0ea1..0x0ea3] | [0x0ea5..0x0ea5] | [0x0ea7..0x0ea7] | [0x0eaa..0x0eab] |
[0x0ead..0x0eae] | [0x0eb0..0x0eb0] | [0x0eb2..0x0eb3] | [0x0ebd..0x0ebd] |
[0x0ec0..0x0ec4] | [0x0ec6..0x0ec6] | [0x0edc..0x0edd] | [0x0f40..0x0f47] |
[0x0f49..0x0f69] | [0x10a0..0x10c5] | [0x10d0..0x10f6] | [0x1100..0x1159] |
[0x115f..0x11a2] | [0x11a8..0x11f9] | [0x1e00..0x1e9b] | [0x1ea0..0x1ef9] |
[0x1f00..0x1f15] | [0x1f18..0x1f1d] | [0x1f20..0x1f45] | [0x1f48..0x1f4d] |
[0x1f50..0x1f57] | [0x1f59..0x1f59] | [0x1f5b..0x1f5b] | [0x1f5d..0x1f5d] |
[0x1f5f..0x1f7d] | [0x1f80..0x1fb4] | [0x1fb6..0x1fbc] | [0x1fbe..0x1fbe] |
[0x1fc2..0x1fc4] | [0x1fc6..0x1fcc] | [0x1fd0..0x1fd3] | [0x1fd6..0x1fdb] |
[0x1fe0..0x1fec] | [0x1ff2..0x1ff4] | [0x1ff6..0x1ffc] | [0x207f..0x207f] |
[0x2102..0x2102] | [0x2107..0x2107] | [0x210a..0x2113] | [0x2115..0x2115] |
[0x2118..0x211d] | [0x2124..0x2124] | [0x2126..0x2126] | [0x2128..0x2128] |
[0x212a..0x2131] | [0x2133..0x2138] | [0x3005..0x3005] | [0x3031..0x3035] |
[0x3041..0x3094] | [0x309b..0x309e] | [0x30a1..0x30fa] | [0x30fc..0x30fe] |
[0x3105..0x312c] | [0x3131..0x318e] | [0x4e00..0x9fa5] | [0xac00..0xd7a3] |
[0xf900..0xfa2d] | [0xfb00..0xfb06] | [0xfb13..0xfb17] | [0xfb1f..0xfb28] |
[0xfb2a..0xfb36] | [0xfb38..0xfb3c] | [0xfb3e..0xfb3e] | [0xfb40..0xfb41] |
[0xfb43..0xfb44] | [0xfb46..0xfbb1] | [0xfbd3..0xfd3d] | [0xfd50..0xfd8f] |
[0xfd92..0xfdc7] | [0xfdf0..0xfdfb] | [0xfe70..0xfe72] | [0xfe74..0xfe74] |
```

```
    [0xfe76..0xfefc] | [0xff21..0xff3a] | [0xff41..0xff5a] | [0xff66..0xffbe] |
    [0xffc2..0xffc7] | [0xffca..0xffcf] | [0xffd2..0xffd7] | [0xffda..0xffdc];

// all unicode characters which are numbers
unicode_digit =
    [0x0030..0x0039] | [0x0660..0x0669] | [0x06f0..0x06f9] | [0x0966..0x096f] |
    [0x09e6..0x09ef] | [0x0a66..0x0a6f] | [0x0ae6..0x0aef] | [0x0b66..0x0b6f] |
    [0x0be7..0x0bef] | [0x0c66..0x0c6f] | [0x0ce6..0x0cef] | [0x0d66..0x0d6f] |
    [0x0e50..0x0e59] | [0x0ed0..0x0ed9] | [0x0f20..0x0f29] | [0xff10..0xff19];

// letters used in identifiers
letter = unicode_letter | '_';
letter_or_digit = unicode_letter | unicode_digit | '_';

// all kind of digits
digit = ['0' .. '9'];
digit_sequence = digit+;
nonzero_digit = [digit - '0'];
octal_digit = ['0' .. '7'];
octal_constant = '0' octal_digit*;
hex_prefix = '0x' | '0X';
hex_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
hex_digit_sequence = hex_digit+;
hex_constant = hex_prefix hex_digit*;
hex_quad = hex_digit hex_digit hex_digit hex_digit;

// suffixes
unsigned_suffix = 'u' | 'U';
long_suffix = 'l' | 'L';
long_long_suffix = 'll' | 'LL';
integer_suffix = unsigned_suffix long_suffix? |
    unsigned_suffix long_long_suffix |
    long_suffix unsigned_suffix? | long_long_suffix unsigned_suffix?;
sign = '+' | '-';
exponent_part = ('e' | 'E') sign? digit_sequence;
binary_exponent_part = ('p' | 'P') sign? digit_sequence;
floating_suffix = ('f' | 'l' | 'F' | 'L');

// constants
decimal_constant = nonzero_digit digit*;
fractional_constant = digit_sequence? '.' digit_sequence | digit_sequence '.';
decimal_floating_constant =
    fractional_constant exponent_part? floating_suffix? |
    digit_sequence exponent_part floating_suffix?;
hex_fractional_constant =
    hex_digit_sequence? '.' hex_digit_sequence |
    hex_digit_sequence '.';
hex_floating_constant =
    hex_prefix (hex_fractional_constant | hex_digit_sequence)
        binary_exponent_part floating_suffix?;
```

```
    // escape sequences
    simple_escape_seq =
        '\' ''' | '\"' | '\?' | '\\' | '\a' | '\b' | '\f' | '\n' |
        '\r' | '\t' | '\v';
    octal_escape_seq = '\' octal_digit octal_digit? octal_digit?;
    hex_escape_seq = '\x' hex_digit+;
    universal_char_name = '\u' hex_quad;
    escape_seq = simple_escape_seq | octal_escape_seq |
        hex_escape_seq | universal_char_name;

    // characters
    c_char = [all - [''' + ['\' + [cr + lf]]]] | escape_seq;
    c_char_seq = c_char+;
    single_character_constant = 'L'? ''' c_char_seq ''';

    // strings
    s_char = [all - ['"' + ['\' + [cr + lf]]]] | escape_seq;
    s_char_seq = s_char+;
    single_string_literal = 'L'? '"' s_char_seq? '"';

/*
 * Tokens and keywords.
 */
Tokens

    preprocessor_command = '#' not_eol* eol;

    blank = empty+;

    comment = ('//' not_eol* eol)
        | ('/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/')
        ;

    // these keywords had to be packed into one token
    // to get line/pos information easily to type node.
    int_keyword = 'int' | 'signed' |
        ('signed' empty+)? 'short' (empty+ 'int')? |
        ('signed' empty+)? 'long' (empty+ 'int')?;
    unsigned_int_keyword = 'unsigned' (empty+ 'int')? |
        'unsigned' empty+ 'short' (empty+ 'int')? |
        'unsigned' empty+ 'long' (empty+ 'int')?;
    char_keyword = ('signed' empty+)? 'char' | 'unsigned' empty+ 'char';
    double_keyword = ('long' empty+)? 'double';

    /* Types */
    kw_bool = 'bool';
    kw_const = 'const';
    kw_enum = 'enum';
    kw_float = 'float';
    kw_struct = 'struct';
    kw_void = 'void';
```

```
// these are already handled above.
/*
kw_char = 'char';
kw_double = 'double';
kw_int = 'int';
kw_long = 'long';
kw_short = 'short';
kw_signed = 'signed';
kw_unsigned = 'unsigned';
*/

/* Keywords */
kw_break = 'break';
kw_case = 'case';
kw_continue = 'continue';
kw_default = 'default';
kw_do = 'do';
kw_else = 'else';
kw_for = 'for';
kw_if = 'if';
kw_include = 'include';
kw_namespace = 'namespace';
kw_return = 'return';
kw_static_cast = 'static_cast';
kw_switch = 'switch';
kw_using = 'using';
kw_while = 'while';

// see boolean_constant below
/*
kw_false = 'false';
kw_true = 'true';
*/

/* Alternative representations for operators */
kw_and = 'and';         // &&
kw_not = 'not';         // !
kw_or = 'or';           // ||

/* Not used. Use of these keywords is forbidden. */
kw_and_eq = 'and_eq';   // &=
kw_asm = 'asm';
kw_auto = 'auto';
kw_bitand = 'bitand';   // &
kw_bitor = 'bitor';     // |
kw_compl = 'compl';     // ~
kw_boolean = '_Bool';
kw_catch = 'catch';
kw_class = 'class';
kw_complex = '_Complex';
```

```
kw_const_cast = 'const_cast';
kw_delete = 'delete';
kw_dynamic_cast = 'dynamic_cast';
kw_explicit = 'explicit';
kw_export = 'export';
kw_extern = 'extern';
kw_friend = 'friend';
kw_goto = 'goto';
kw_imaginary = '_Imaginary';
kw_inline = 'inline';
kw_mutable = 'mutable';
kw_new = 'new';
kw_not_eq = 'not_eq';   // !=  note: there's no 'eq' keyword for ==, dunno why
kw_operator = 'operator';
kw_or_eq = 'or_eq';     // |=
kw_private = 'private';
kw_protected = 'protected';
kw_public = 'public';
kw_register = 'register';
kw_reinterpret_cast = 'reinterpret_cast';
kw_restrict = 'restrict';
kw_sizeof = 'sizeof';
kw_static = 'static';
kw_static_assert = 'static_assert';
kw_template = 'template';
kw_this = 'this';
kw_throw = 'throw';
kw_try = 'try';
kw_typedef = 'typedef';
kw_typeid = 'typeid';
kw_typename = 'typename';
kw_union = 'union';
kw_virtual = 'virtual';
kw_volatile = 'volatile';
kw_xor = 'xor';         // ^
kw_xor_eq = 'xor_eq';   // ^=
kw_wchar_t = 'wchar_t';

/* Although these aren't keywords in C++, they are in this language */
kw_string = 'string';
kw_vector = 'vector';
kw_vip_out = '__vip';
kw_vip_in = '__vip_in';
kw_cin = 'cin';
kw_cout = 'cout';
kw_getline = 'getline';

/* Tokens */
tok_lbracket = '[';
tok_rbracket = ']';
tok_lpar = '(';
```

```
tok_rpar = ')';
tok_lbrace = '{';
tok_rbrace = '}';
tok_dot = '.';
tok_arrow = '->';
tok_plus_plus = '++';
tok_minus_minus = '--';
tok_amp = '&';
tok_star = '*';
tok_plus = '+';
tok_minus = '-';
tok_exclamation = '!';
tok_slash = '/';
tok_percent = '%';
tok_lshift = '<<';
tok_rshift = '>>';
tok_lt = '<';
tok_gt = '>';
tok_lt_eq = '<=';
tok_gt_eq = '>=';
tok_eq_eq = '==';
tok_not_eq = '!=';
tok_amp_amp = '&&';
tok_bar_bar = '||';
tok_colon = ':';
tok_colon_colon = '::';
tok_semicolon = ';';
tok_eq = '=';
tok_star_eq = '*=';
tok_slash_eq = '/=';
tok_percent_eq = '%=';
tok_plus_eq = '+=';
tok_minus_eq = '-=';
tok_comma = ',';

/* Tokens which are not used */
/*
tok_dot_star = '.*';
tok_arrow_star = '->*';
tok_tilde = '~';
tok_urshift = '>>>';
tok_caret = '^';
tok_bar = '|';
tok_question = '?';
tok_colon_colon_star = '::*';
tok_ellipsis = '...';
tok_lshift_eq = '<<=';
tok_rshift_eq = '>>=';
tok_urshift_eq = '>>>=';
tok_amp_eq = '&=';
tok_caret_eq = '^=';
```

```
    tok_bar_eq = '|=';
    tok_hash = '#';
    tok_hash_hash = '##';
    tok_lt_colon = '<:';        // [
    tok_colon_gt = ':>';        // ]
    tok_lt_percent = '<%';      // {
    tok_percent_gt = '%>';      // }
    tok_percent_colon = '%:';   // #
    tok_percent_colon_percent_colon = '%:%:'; // ##
    tok_qq_eq = '??=';          // #
    tok_qq_lpar = '??(';        // [
    tok_qq_rpar = '??)';        // ]
    tok_qq_lt = '??<';          // {
    tok_qq_gt = '??>';          // }
    tok_qq_slash = '??/';       // \
    tok_qq_apostrophe = '??' ''';// ^
    tok_qq_exclamation = '??!'; // |
    tok_qq_minus = '??-';       // ~
    tok_qq_question = '???';    // ?
    */

    // constants and literals
    integer_constant = decimal_constant integer_suffix? |
        octal_constant integer_suffix? |
        hex_constant integer_suffix?;
    floating_constant = decimal_floating_constant | hex_floating_constant;
    character_constant = single_character_constant;
    boolean_constant = 'true' | 'false';
    string_literal = single_string_literal (empty* single_string_literal)*;

    identifier = letter letter_or_digit*;

/*
 * Ignored tokens
 */
Ignored Tokens
    comment,
    blank;

/*
 * Productions
 */
Productions

program {-> program} =
    progdeclaration*
        {-> New program([progdeclaration.statement]) }
    ;

progdeclaration {-> statement?} =
    {function} function
```

```
        {-> function.statement}
    | {struct} structdecl tok_semicolon
        {-> structdecl.statement}
    | {enum} enumdecl tok_semicolon
        {-> enumdecl.statement}
    | {using_namespace} kw_using kw_namespace simple_name tok_semicolon
        {-> New statement.using(kw_using, tok_semicolon)}
    | {using} kw_using simple_name tok_colon_colon using_identifier tok_semicolon
        {-> New statement.using(kw_using, tok_semicolon)}
    | {preprocessor_command} preprocessor_command
        {-> New statement.preprocessor(preprocessor_command)}
    | {variable} variable_declaration tok_semicolon
        {-> New statement.declaration(variable_declaration.declaration)}

    /* Not c++ standard!! Included to allow interpreter functionality.
       Variable declaration (above) separated because it's in standard
       (global variables) */
    | {global_statement} global_statement
        {-> New statement.global(global_statement.statement)}
    ;

// variable name
name {-> variable} =
    {simplename} simple_name
        {-> simple_name.variable}
    | {name} name tok_dot identifier
        {-> New variable.field(name.variable, identifier)}
    ;

simple_name {-> variable} =
    identifier
        {-> New variable.variable(identifier)}
    ;

using_identifier =
    {simple_name} simple_name
    | {string} kw_string
    | {vector} kw_vector
    | {cin} kw_cin
    | {cout} kw_cout
    ;

function {-> statement} =
    {declaration} type simple_name tok_lpar paramdeclpart? tok_rpar tok_semicolon
        {-> New statement.function_declaration(type.type, simple_name.variable,
            [paramdeclpart.declaration])}
    | {definition} type simple_name tok_lpar paramdeclpart? tok_rpar statementseq
        {-> New statement.function_definition(type.type, simple_name.variable,
            [paramdeclpart.declaration], [statementseq.statement])}
    ;
```

```
paramdeclpart {-> declaration*} =
    {void} kw_void
        {-> []}
    | {list} paramdecl_list
        {-> [paramdecl_list.declaration]}
    ;

paramdecl_list {-> declaration*} =
    {variable_declaration} variable_declaration
        {-> [variable_declaration.declaration]}
    | {list} paramdecl_list tok_comma variable_declaration
        {-> [paramdecl_list.declaration, variable_declaration.declaration]}
    ;

// different alternatives for simple and array declaration.
// No multidimensional arrays possible.
variable_declaration {-> declaration} =
    {simple} type simple_name variable_init_simple
        {-> New declaration.simple(type.type, simple_name.variable,
            [variable_init_simple.expression])}
    | {array} type simple_name tok_lbracket shift_expression? tok_rbracket
            variable_init_array
        {-> New declaration.array(New type.array(type.type), simple_name.variable,
            shift_expression.expression, [variable_init_array.expression])}
    ;

variable_init_simple {-> expression*} =
    {empty}
        {-> []}
    | {initializer} tok_eq variable_initializer
        {-> [variable_initializer.expression]}
    ;

variable_init_array {-> expression*} =
    {empty}
        {-> []}
    | {empty_braces} tok_eq tok_lbrace tok_rbrace
        {-> []}
    | {initializers} tok_eq tok_lbrace variable_initializer_list tok_rbrace
        {-> [variable_initializer_list.expression]}
    ;

variable_initializer_list {-> expression*} =
    {initializer} variable_initializer
        {-> [variable_initializer.expression]}
    | {list} variable_initializer_list tok_comma variable_initializer
        {-> [variable_initializer_list.expression,
             variable_initializer.expression]}
    ;

variable_initializer {-> expression*} =
```

```
    {expression} expression
        {-> [expression.expression]}
    | {struct_initializer} tok_lbrace struct_initializer_list tok_rbrace
        {-> [struct_initializer_list.expression]}
    ;

struct_initializer_list {-> expression*} =
    {single} expression
        {-> [expression.expression]}
    | {multi} struct_initializer_list tok_comma expression
        {-> [struct_initializer_list.expression, expression.expression]}
    ;

structdecl {-> statement} =
    kw_struct simple_name tok_lbrace variable_declarationlist tok_rbrace
        {-> New statement.struct(simple_name.variable,
            [variable_declarationlist.declaration])}
    ;

variable_declarationlist {-> declaration*} =
    {single} variable_declaration tok_semicolon
        {-> [variable_declaration.declaration]}
    | {list} variable_declarationlist variable_declaration tok_semicolon
        {-> [variable_declarationlist.declaration,
             variable_declaration.declaration]}
    ;

enumdecl {-> statement} =
    kw_enum simple_name tok_lbrace enum_variable_list tok_rbrace
        {-> New statement.enum(simple_name.variable,
                               [enum_variable_list.declaration])}
    ;

enum_variable_list {-> declaration*} =
    {single} enum_variable
        {-> [enum_variable.declaration]}
    | {list} enum_variable_list tok_comma enum_variable
        {-> [enum_variable_list.declaration, enum_variable.declaration]}
    ;

enum_variable {-> declaration} =
    simple_name enum_variable_init
        {-> New declaration.enum_variable(simple_name.variable,
            enum_variable_init.expression)}
    ;

enum_variable_init {-> expression?} =
    {empty}
        {-> Null}
    | {expression} tok_eq expression
        {-> expression.expression}
```

```
    ;

type {-> type} =
    {normal} non_const_type
        {-> non_const_type.type}
    | {const} const_type
        {-> const_type.type}
    ;

non_const_type {-> type} =
    {primitive} primitive_type
        {-> primitive_type.type}
    | {reference} non_const_type tok_amp
        {-> New type.reference(non_const_type.type)}
    | {pointer} type tok_star
        {-> New type.pointer(type.type)}
    ;

// Const reference is here separatedly because
// 'const int &' means a constant reference to int.
// However, 'const int *' means a pointer to const int. Logical, isn't it.
const_type {-> type} =
    {primitive_const_before} kw_const primitive_type
        {-> New type.const(kw_const, primitive_type.type)}
    | {primitive_const_after} primitive_type kw_const
        {-> New type.const(kw_const, primitive_type.type)}
    | {reference_const_before} kw_const primitive_type tok_amp
        {-> New type.const(kw_const, New type.reference(primitive_type.type))}
    | {reference_const_middle} primitive_type kw_const tok_amp
        {-> New type.const(kw_const, New type.reference(primitive_type.type))}
    | {pointer} type tok_star kw_const
        {-> New type.const(kw_const, New type.pointer(type.type))}
    ;

primitive_type {-> type} =
    {int} int_keyword
        {-> New type.int(int_keyword)}
    | {unsignedint} unsigned_int_keyword
        {-> New type.unsigned_int(unsigned_int_keyword)}
    | {string} kw_string
        {-> New type.string(kw_string)}
    | {bool} kw_bool
        {-> New type.bool(kw_bool)}
    | {char} char_keyword
        {-> New type.char(char_keyword)}
    | {double} double_keyword
        {-> New type.double(double_keyword)}
    | {float} kw_float
        {-> New type.float(kw_float)}
    | {void} kw_void
        {-> New type.void(kw_void)}
```

```
    | {vector} kw_vector tok_lt type tok_gt
        {-> New type.vector(type.type)}
    | {name} name
        {-> New type.userdefined(name.variable)}
    ;

statementseq {-> tok_lbrace statement* tok_rbrace} =
    {sequence} tok_lbrace statement+ tok_rbrace
        {-> tok_lbrace [statement.statement] tok_rbrace}
    | {empty} tok_lbrace tok_rbrace
        {-> tok_lbrace [] tok_rbrace}
    ;

statement {-> statement} =
    {without_trailing_substatement} statement_without_trailing_substatement
        {-> statement_without_trailing_substatement.statement}
    | {no_variable_declaration} statement_no_variable_declaration
        {-> statement_no_variable_declaration.statement}
    | {variable} local_variable_declaration tok_semicolon
        {-> New statement.declaration(local_variable_declaration.declaration)}
    ;

// Global statement, this is not in c++ standard.
// Variable declaration is separated because it's in standard.
global_statement {-> statement} =
    {without_trailing_substatement} statement_without_trailing_substatement
        {-> statement_without_trailing_substatement.statement}
    | {no_variable_declaration} statement_no_variable_declaration
        {-> statement_no_variable_declaration.statement}
    ;

statement_no_variable_declaration {-> statement} =
    {if_then_statement} if_then_statement
        {-> if_then_statement.statement}
    | {if_then_else_statement} if_then_else_statement
        {-> if_then_else_statement.statement}
    | {while_statement} while_statement
        {-> while_statement.statement}
    | {for_statement} for_statement
        {-> for_statement.statement}
    ;

// Inside if statement there can't be ifs without else, if the outer if
// has else part and there are no { } tokens.
statement_no_short_if {-> statement} =
    {without_trailing_substatement} statement_without_trailing_substatement
        {-> statement_without_trailing_substatement.statement}
    | {if_then_else_statement_no_short_if} if_then_else_statement_no_short_if
        {-> if_then_else_statement_no_short_if.statement}
    | {while_statement_no_short_if} while_statement_no_short_if
        {-> while_statement_no_short_if.statement}
```

```
    | {for_statement_no_short_if} for_statement_no_short_if
        {-> for_statement_no_short_if.statement}
    | {variable} local_variable_declaration tok_semicolon
        {-> New statement.declaration(local_variable_declaration.declaration)}
    ;

statement_without_trailing_substatement {-> statement} =
    {sequence} statementseq
        {-> New statement.sequence(statementseq.tok_lbrace,
            [statementseq.statement], statementseq.tok_rbrace)}
    | {expression} expression_statement
        {-> expression_statement.statement}
    | {do} do_statement
        {-> do_statement.statement}
    | {break} break_statement
        {-> break_statement.statement}
    | {empty} empty_statement
        {-> empty_statement.statement}
    | {continue} continue_statement
        {-> continue_statement.statement}
    | {return} return_statement
        {-> return_statement.statement}
    | {cin} cin_statement
        {-> cin_statement.statement}
    | {cout} cout_statement
        {-> cout_statement.statement}
    | {vipout} vipout_statement
        {-> vipout_statement.statement}
    | {vipin} vipin_statement
        {-> vipin_statement.statement}
    | {switchstatement} switch_statement
        {-> switch_statement.statement}
    ;

local_variable_declaration {-> declaration} =
    variable_declaration
        {-> variable_declaration.declaration}
    ;

cin_statement {-> statement} =
    kw_cin tok_rshift left_hand_side tok_semicolon
        {-> New statement.cin(left_hand_side.expression)}
    ;

cout_statement {-> statement} =
    kw_cout tok_lshift cout_list tok_semicolon
        {-> New statement.cout([cout_list.expression])}
    ;

cout_list {-> expression*} =
    {cout_expression} cout_expression
```

```
        {-> [cout_expression.expression]}
    | {list} cout_list tok_lshift cout_expression
        {-> [cout_list.expression, cout_expression.expression]}
    ;

cout_expression {->expression} =
    {additive_expression} additive_expression
        {-> additive_expression.expression}
    ;

// vip out stream is for outputting text to screen, different from cout
vipout_statement {-> statement} =
    kw_vip_out tok_lshift cout_list tok_semicolon
        {-> New statement.vip_out([cout_list.expression])}
    ;

// vip in stream is for inputting text to cin buffer
vipin_statement {-> statement} =
    kw_vip_in tok_lshift cout_list tok_semicolon
        {-> New statement.vip_in([cout_list.expression])}
    ;

empty_statement {-> statement} =
    tok_semicolon
        {-> New statement.empty(tok_semicolon)}
    ;

if_then_statement {-> statement} =
    if_beginning statement
        {-> New statement.if(if_beginning.expression, statement.statement, Null)}
    ;

if_then_else_statement {-> statement} =
    if_beginning statement_no_short_if kw_else statement
        {-> New statement.if(if_beginning.expression,
            statement_no_short_if.statement, statement.statement)}
    ;

if_then_else_statement_no_short_if {-> statement} =
    if_beginning [statement1]:statement_no_short_if kw_else
    [statement2]:statement_no_short_if
        {-> New statement.if(if_beginning.expression, statement1.statement,
            statement2.statement)}
    ;

if_beginning {-> expression} =
    kw_if tok_lpar conditional_expression tok_rpar
        {-> conditional_expression.expression}
    ;

while_statement {-> statement} =
```

```
    kw_while tok_lpar conditional_expression tok_rpar statement
        {-> New statement.while(conditional_expression.expression,
                                statement.statement)}
    ;

while_statement_no_short_if {-> statement} =
    kw_while tok_lpar conditional_expression tok_rpar statement_no_short_if
        {-> New statement.while(conditional_expression.expression,
            statement_no_short_if.statement)}
    ;

do_statement {-> statement} =
    kw_do statement kw_while tok_lpar conditional_expression tok_rpar tok_semicolon
        {-> New statement.do_while(statement.statement,
            conditional_expression.expression)}
    ;

switch_statement {-> statement} =
    kw_switch tok_lpar additive_expression tok_rpar tok_lbrace
    case_or_default_statement* tok_rbrace
        {-> New statement.switch(additive_expression.expression,
            [case_or_default_statement.statement])}
    ;

case_or_default_statement {-> statement} =
    {case} kw_case case_constant tok_colon statement*
        {-> New statement.case(case_constant.constant, [statement.statement])}
    | {default} kw_default tok_colon statement*
        {-> New statement.default(kw_default, [statement.statement])}
    ;

for_statement {-> statement} =
    kw_for tok_lpar for_init_opt [semicolon1]:tok_semicolon
    conditional_expression_opt [semicolon2]:tok_semicolon for_update_opt
    tok_rpar statement
        {-> New statement.for([for_init_opt.expression],
            conditional_expression_opt.expression,
            [for_update_opt.expression], statement.statement)}
    ;

for_statement_no_short_if {-> statement} =
    kw_for tok_lpar for_init_opt [semicolon1]:tok_semicolon
    conditional_expression_opt [semicolon2]:tok_semicolon for_update_opt
    tok_rpar statement_no_short_if
        {-> New statement.for([for_init_opt.expression],
            conditional_expression_opt.expression,
            [for_update_opt.expression], statement_no_short_if.statement)}
    ;

conditional_expression_opt {-> expression?} =
    {empty}
```

```
        {-> Null}
    | {conditional_expression} conditional_expression
        {-> conditional_expression.expression}
    ;

for_init_opt {-> expression*} =
    {empty}
        {-> []}
    | {init} for_init
        {-> [for_init.expression]}
    ;

for_init {-> expression*} =
    {list} statement_expression_list
        {-> [statement_expression_list.expression]}
    | {variable} local_variable_declaration
        {-> [New expression.declaration(local_variable_declaration.declaration)]}
    ;

for_update_opt {-> expression*} =
    {empty}
        {-> []}
    | {update} for_update
        {-> [for_update.expression]}
    ;

for_update {-> expression*} =
    statement_expression_list
        {-> [statement_expression_list.expression]}
    ;

statement_expression_list {-> expression*} =
    {statement_expression} statement_expression
        {-> [statement_expression.expression]}
    | {list} statement_expression_list tok_comma statement_expression
        {-> [statement_expression_list.expression, statement_expression.expression]}
    ;

expression_statement {-> statement} =
    statement_expression tok_semicolon
        {-> New statement.expression(statement_expression.expression)}
    ;

statement_expression {-> expression} =
    {assignment} assignment
        {-> assignment.expression}
    // can't be expression here because 'foo * bar;' would cause a conflict:
    // it can mean multiplication or a pointer declaration.
    | {expression} unary_expression
        {-> unary_expression.expression}
    ;
```

```
assignment {-> expression} =
    {eq} left_hand_side tok_eq expression
        {-> New expression.assignment_eq(left_hand_side.expression,
            expression.expression)}
    | {mul_eq} left_hand_side tok_star_eq expression
        {-> New expression.assignment_mul_eq(left_hand_side.expression,
            expression.expression)}
    | {div_eq} left_hand_side tok_slash_eq expression
        {-> New expression.assignment_div_eq(left_hand_side.expression,
            expression.expression)}
    | {mod_eq} left_hand_side tok_percent_eq expression
        {-> New expression.assignment_mod_eq(left_hand_side.expression,
            expression.expression)}
    | {plus_eq} left_hand_side tok_plus_eq expression
        {-> New expression.assignment_plus_eq(left_hand_side.expression,
            expression.expression)}
    | {minus_eq} left_hand_side tok_minus_eq expression
        {-> New expression.assignment_minus_eq(left_hand_side.expression,
            expression.expression)}
    ;

left_hand_side {-> expression} =
    unary_expression
        {-> unary_expression.expression}
    ;

break_statement {-> statement} =
    kw_break tok_semicolon
        {-> New statement.break(kw_break)}
    ;

continue_statement {-> statement} =
    kw_continue tok_semicolon
        {-> New statement.continue(kw_continue)}
    ;

return_statement {-> statement} =
    kw_return return_expression tok_semicolon
        {-> New statement.return(kw_return, return_expression.expression)}
    ;

return_expression {-> expression?} =
    {empty}
        {-> Null}
    | {expression} expression
        {-> expression.expression}
    ;

expression {-> expression} =
    conditional_expression
```

```
        {-> conditional_expression.expression}
    ;

conditional_expression {-> expression} =
    {conditional_and_expression} conditional_and_expression
        {-> conditional_and_expression.expression}
    | {bar_bar} conditional_expression tok_bar_bar conditional_and_expression
        {-> New expression.logical_or(conditional_expression.expression,
            conditional_and_expression.expression)}
    | {or} conditional_expression kw_or conditional_and_expression
        {-> New expression.logical_or(conditional_expression.expression,
            conditional_and_expression.expression)}
    ;

conditional_and_expression {-> expression} =
    {inclusive_or_expression} inclusive_or_expression
        {-> inclusive_or_expression.expression}
    | {amp_amp} conditional_and_expression tok_amp_amp inclusive_or_expression
        {-> New expression.logical_and(conditional_and_expression.expression,
            inclusive_or_expression.expression)}
    | {and} conditional_and_expression kw_and inclusive_or_expression
        {-> New expression.logical_and(conditional_and_expression.expression,
            inclusive_or_expression.expression)}
    ;

// bit operations are removed from C--
inclusive_or_expression {-> expression} =
    exclusive_or_expression
        {-> exclusive_or_expression.expression}
//  | {bar} inclusive_or_expression tok_bar exclusive_or_expression
    ;

exclusive_or_expression {-> expression} =
    and_expression {-> and_expression.expression}
//  | {caret} exclusive_or_expression tok_caret and_expression
    ;

and_expression {-> expression} =
    equality_expression {-> equality_expression.expression}
//  | {amp} and_expression tok_amp equality_expression
    ;

equality_expression {-> expression} =
    {relational_expression} relational_expression
        {-> relational_expression.expression}
    | {eq_eq} [relexp1]:relational_expression tok_eq_eq
              [relexp2]:relational_expression
        {-> New expression.eq_eq(relexp1.expression, relexp2.expression)}
    | {not_eq} [relexp1]:relational_expression tok_not_eq
               [relexp2]:relational_expression
        {-> New expression.not_eq(relexp1.expression, relexp2.expression)}
```

```
    ;

relational_expression {-> expression} =
    {shift_expression} shift_expression
        {-> shift_expression.expression}
    | {lt} relational_expression tok_lt shift_expression
        {-> New expression.lt(relational_expression.expression,
            shift_expression.expression)}
    | {gt} relational_expression tok_gt shift_expression
        {-> New expression.gt(relational_expression.expression,
            shift_expression.expression)}
    | {lt_eq} relational_expression tok_lt_eq shift_expression
        {-> New expression.lt_eq(relational_expression.expression,
            shift_expression.expression)}
    | {gt_eq} relational_expression tok_gt_eq shift_expression
        {-> New expression.gt_eq(relational_expression.expression,
            shift_expression.expression)}
    ;

// bit operations are removed from C--
shift_expression {-> expression} =
    {additive_expression} additive_expression
        {-> additive_expression.expression}
//  | {lshift} shift_expression tok_lshift additive_expression
//  | {rshift} shift_expression tok_rshift additive_expression
    ;

additive_expression {-> expression} =
    {multiplicative_expression} multiplicative_expression
        {-> multiplicative_expression.expression}
    | {plus} additive_expression tok_plus multiplicative_expression
        {-> New expression.plus(additive_expression.expression,
            multiplicative_expression.expression)}
    | {minus} additive_expression tok_minus multiplicative_expression
        {-> New expression.minus(additive_expression.expression,
            multiplicative_expression.expression)}
    ;

multiplicative_expression {-> expression} =
    {cast_expression} cast_expression
        {-> cast_expression.expression}
    | {star} multiplicative_expression tok_star cast_expression
        {-> New expression.multiply(multiplicative_expression.expression,
            cast_expression.expression)}
    | {slash} multiplicative_expression tok_slash cast_expression
        {-> New expression.divide(multiplicative_expression.expression,
            cast_expression.expression)}
    | {percent} multiplicative_expression tok_percent cast_expression
        {-> New expression.mod(multiplicative_expression.expression,
            cast_expression.expression)}
    ;
```

```
cast_expression {-> expression} =
    unary_expression {-> unary_expression.expression}
    ;

unary_expression {-> expression} =
    {postfix_expression} postfix_expression
        {-> postfix_expression.expression}
    | {preincdec} unary_expression_preincrement_predecrement
        {-> unary_expression_preincrement_predecrement.expression}
    | {plus_minus_star_amp} unary_expression_plus_minus_star_amp
        {-> unary_expression_plus_minus_star_amp.expression}
    | {not_plus_minus} unary_expression_not_plus_minus
        {-> unary_expression_not_plus_minus.expression}
    ;

unary_expression_preincrement_predecrement {-> expression} =
    {plus_plus} tok_plus_plus cast_expression
        {-> New expression.preincrement(cast_expression.expression)}
    | {minus_minus} tok_minus_minus cast_expression
        {-> New expression.predecrement(cast_expression.expression)}
    ;

unary_expression_plus_minus_star_amp {-> expression} =
    {plus} tok_plus cast_expression
        {-> New expression.unary_plus(cast_expression.expression)}
    | {minus} tok_minus cast_expression
        {-> New expression.unary_minus(cast_expression.expression)}
    | {star} tok_star cast_expression
        {-> New expression.dereference(cast_expression.expression)}
    | {amp} tok_amp cast_expression
        {-> New expression.address(cast_expression.expression)}
    ;

// bit operations are removed from C--
unary_expression_not_plus_minus {-> expression} =
    {exclamation} tok_exclamation cast_expression
        {-> New expression.not(cast_expression.expression)}
    | {not} kw_not cast_expression
        {-> New expression.not(cast_expression.expression)}
//  | {tilde} tok_tilde cast_expression
//      {-> New expression.inverse(cast_expression.expression)}
    ;

postfix_expression {-> expression} =
    {primary} primary
        {-> primary.expression}
    | {name} name
        {-> New expression.variable(name.variable)}
    | {static_cast} kw_static_cast tok_lt type tok_gt tok_lpar expression tok_rpar
        {-> New expression.static_cast(type.type, expression.expression)}
```

```
    | {postincrement} postfix_expression tok_plus_plus
        {-> New expression.postincrement(postfix_expression.expression)}
    | {postdecrement} postfix_expression tok_minus_minus
        {-> New expression.postdecrement(postfix_expression.expression)}
    ;

primary {-> expression} =
    {literal} literal
        {-> New expression.constant(literal.constant)}
    | {expression} tok_lpar expression tok_rpar
        {-> New expression.parentheses(expression.expression)}
    | {function_call} function_call
        {-> function_call.expression}
    | {pointer_access} pointer_access
        {-> pointer_access.expression}
    | {array_access} array_access
        {-> array_access.expression}
    | {field_access} field_access
        {-> field_access.expression}
    ;

literal {-> constant} =
    {constant} constant
        {-> constant.constant}
    | {boolean} boolean_constant
        {-> New constant.boolean(boolean_constant)}
    | {string} string_literal
        {-> New constant.string(string_literal)}
    ;

constant {-> constant} =
    {integer} integer_constant
        {-> New constant.integer(integer_constant)}
    | {float} floating_constant
        {-> New constant.float(floating_constant)}
    | {char_seq} character_constant
        {-> New constant.char(character_constant)}
    ;

case_constant {-> constant} =
    {int} integer_constant
        {-> New constant.integer(integer_constant)}
    | {char_seq} character_constant
        {-> New constant.char(character_constant)}
    | {variable} simple_name
        {-> New constant.variable(simple_name.variable)}
    ;

function_call {-> expression} =
    {primary} primary tok_lpar argument_list tok_rpar
        {-> New expression.function_call(primary.expression, [argument_list.expression])}
```

```
    | {name} name tok_lpar argument_list tok_rpar
        {-> New expression.function_call(New expression.variable(name.variable),
            [argument_list.expression])}
    | {getline} kw_getline tok_lpar kw_cin tok_comma expression tok_rpar
        {-> New expression.getline(expression.expression)}
    ;

argument_list {-> expression*} =
    {empty}
        {-> []}
    | {expression} expression
        {-> [expression.expression]}
    | {list} argument_list tok_comma expression
        {-> [argument_list.expression, expression.expression]}
    ;

pointer_access {-> expression} =
    {primary} primary tok_arrow identifier
        {-> New expression.pointer_access(primary.expression, identifier)}
    | {name} name tok_arrow identifier
        {-> New expression.pointer_access(New expression.variable(name.variable),
            identifier)}
    ;

array_access {-> expression} =
    {primary} primary tok_lbracket shift_expression tok_rbracket
        {-> New expression.array_access(primary.expression,
            shift_expression.expression)}
    | {name} name tok_lbracket shift_expression tok_rbracket
        {-> New expression.array_access(New expression.variable(name.variable),
            shift_expression.expression)}
    ;

field_access {-> expression} =
    primary tok_dot identifier
        {-> New expression.field_access(primary.expression, identifier)}
    ;


/*
 * Abstract Syntax Tree
 */
Abstract Syntax Tree

program =
    statement*
    ;

statement =
    // the braces (and some other tokens too) are included to be able to check
    // line numbers
```

```
    {sequence} tok_lbrace statement* tok_rbrace // { s1; s2; ...}
    | {declaration} declaration                 // int x = 5;
    | {expression} expression                   // 5 + 7;
    | {empty} tok_semicolon                     // ;
    | {global} statement    // statement outside a function, not c++ standard
    | {function_declaration} type variable declaration* // int x(int y, ...);
    | {function_definition} type variable declaration*
                            statement*          // int x(int y, int z, ...) { s1; ... }

    | {enum} variable declaration*              // enum x = { decl1, decl2, ...}
    | {struct} variable declaration*            // struct x = { decl1; decl2; ...}

    | {cin} expression                          // cin >> expression;
    | {cout} expression*                        // cout << exp1 << exp2 << ...;
    | {vip_in} expression*                      // __vip << exp1 << exp2 << ...;
    | {vip_out} expression*                     // __vip_out << exp1 << exp2 << ...;

    | {if} expression [s1]:statement [s2]:statement? // if (expression) s1 else s2
    | {while} expression statement              // while (expression) statement
    | {do_while} statement expression           // do statement while (expression);
    | {for} [exp1]:expression* [exp2]:expression?
          [exp3]:expression* statement          // for (exp1; exp2; exp3) statement

    | {switch} expression statement*            // switch (expression) { s1; s2; }
    | {case} constant statement*                // case 5: s1; s2;
    | {default} kw_default statement*           // default: s1; s2;
    | {break} kw_break                          // break;
    | {continue} kw_continue                    // continue;
    | {return} kw_return expression?            // return expression;

    // these are only for visualizing
    | {preprocessor} preprocessor_command       // #include <iostream>
    | {using} kw_using tok_semicolon            // using namespace std;
    ;

declaration =
    {simple} type variable expression*          // int x = exp
    | {array} type variable [param]:expression? [init]:expression*
       // int x[param] = {init1, init2, ...}
    | {enum_variable} variable expression?      // x = exp
    ;

expression =
    {constant} constant                         // 5
    | {variable} variable                       // x
    | {declaration} declaration                 // int x = 5 (in for)

    | {assignment_eq} [exp1]:expression [exp2]:expression       // exp1 = exp2
    | {assignment_mul_eq} [exp1]:expression [exp2]:expression   // exp1 *= exp2
    | {assignment_div_eq} [exp1]:expression [exp2]:expression   // exp1 /= exp2
    | {assignment_mod_eq} [exp1]:expression [exp2]:expression   // exp1 %= exp2
```

```
    | {assignment_plus_eq} [exp1]:expression [exp2]:expression  // exp1 += exp2
    | {assignment_minus_eq} [exp1]:expression [exp2]:expression // exp1 -= exp2

    | {logical_or} [exp1]:expression [exp2]:expression  // exp1 || exp2
    | {logical_and} [exp1]:expression [exp2]:expression // exp1 && exp2
    | {eq_eq} [exp1]:expression [exp2]:expression       // exp1 == exp2
    | {not_eq} [exp1]:expression [exp2]:expression      // exp1 != exp2
    | {lt} [exp1]:expression [exp2]:expression          // exp1 < exp2
    | {gt} [exp1]:expression [exp2]:expression          // exp1 > exp2
    | {lt_eq} [exp1]:expression [exp2]:expression       // exp1 <= exp2
    | {gt_eq} [exp1]:expression [exp2]:expression       // exp1 >= exp2
    | {plus} [exp1]:expression [exp2]:expression        // exp1 + exp2
    | {minus} [exp1]:expression [exp2]:expression       // exp1 - exp2
    | {multiply} [exp1]:expression [exp2]:expression    // exp1 * exp2
    | {divide} [exp1]:expression [exp2]:expression      // exp1 / exp2
    | {mod} [exp1]:expression [exp2]:expression         // exp1 % exp2

    | {preincrement} expression     // ++expression
    | {predecrement} expression     // --expression
    | {postincrement} expression    // expression++
    | {postdecrement} expression    // expression--
    | {unary_plus} expression       // +expression
    | {unary_minus} expression      // -expression
    | {dereference} expression      // *expression
    | {address} expression          // &expression
    | {not} expression              // !expression

    | {static_cast} type expression                     // static_cast<type>(expression)
    | {function_call} [exp1]:expression [exps]:expression*  // exp1(exps, exps2, ...)
    | {pointer_access} expression identifier                // expression->identifier
    | {array_access} [exp1]:expression [exp2]:expression    // exp1[exp2]
    | {field_access} expression identifier                  // expression.identifier
    | {parentheses} expression                              // (expression)
    | {getline} expression                                  // getline(cin, expression)
    ;

constant =
    {string} string_literal         // "foo"
    | {integer} integer_constant    // 5
    | {float} floating_constant     // 5.0
    | {char} character_constant     // 'A'
    | {boolean} boolean_constant    // true
    | {variable} variable           // FOO (enum or const variable)
    ;

variable =
    {variable} identifier         // x
    | {field} variable identifier // x.y (struct or variable.method in method call)
    ;

type =
```

```
{const} kw_const [inner_type]:type      // const type
| {reference} [inner_type]:type         // type&
| {pointer} [inner_type]:type           // type*
| {vector} [inner_type]:type            // vector<type>
| {array} [inner_type]:type             // type x[]
| {userdefined} variable                // variable (struct or enum)
| {int} int_keyword                     // int
| {unsigned_int} unsigned_int_keyword   // unsigned int
| {string} kw_string                    // string
| {bool} kw_bool                        // bool
| {char} char_keyword                   // char
| {double} double_keyword               // double
| {float} kw_float                      // float
| {void} kw_void                        // void
| {function} // function (just to be able to set symbol type to function)
;
```

LIITE 4: SUOMEKSI LOKALISOIDUT TEKSTIT

```
# finnish messages
parserError =                                Jäsennysvirhe kohdassa [{0}, {1}]: {2}
lexerError =                                 Leksikaalivirhe kohdassa [{0}, {1}]: {2}
otherError =                                 Virhe: {0}
semanticError =                              Semanttinen virhe rivillä {0}: {1}
semanticErrorWithoutLine =                   Semanttinen virhe: {0}
warning =                                    Varoitus riviltä {0}: {1}
warningWithoutLine =                         Varoitus: {0}
info =                                       Huomautus riviltä {0}: {1}
infoWithoutLine =                            Huomautus: {0}
runtimeError =                               Ajonaikainen virhe riviltä {0}: {1}
runtimeErrorWithoutLine =                    Ajonaikainen virhe: {0}
constraint =                                 Virhe rivillä {0}: {1}
constraintWithoutLine =                      Virhe: {0}

cinWarning.couldNotReadInt =                 Virhe luettaessa virrasta: Odotettiin kokonaislukua, \
                                             saatiin "{0}". Syötä kokonaisluku uudelleen.
cinWarning.couldNotReadFloat =               Virhe luettaessa virrasta: Odotettiin liukulukua, \
                                             saatiin "{0}". Syötä liukuluku uudelleen.


parserError.expecting =                      odotettiin jotain seuraavista
parserError.identifier =                     tunniste
parserError.preprocessorCommand =            esikääntäjän komento
parserError.integerConstant =                kokonaislukuvakio
parserError.floatingConstant =               liukulukuvakio
parserError.characterConstant =              merkkivakio
parserError.booleanConstant =                totuusarvo
parserError.stringLiteral =                  merkkijonoliteraali
parserError.EOF =                            syötteen loppu

parserError.vectorBrackets =                 Huomioi, että vektorin esittelyssä '>>' on kirjoitettava \
                                             '> >', koska yhteenkirjoitettuna '>>' \
                                             tarkoittaa virtaoperaattoria.
parserError.comma =                          Virhe saattaa johtua pilkusta, sillä C--:ssa et voi \
                                             esitellä montaa muuttujaa samalla rivillä.
parserError.conditional =                    Huomioi, että C--:ssa ehtolausekkeen (?:) käyttö on kiellettyä.
parserError.ellipsis =                       Huomioi, että C--:ssa muuttuva parametrien määrä (...) \
                                             on kielletty.
parserError.namespace =                      Huomioi, että C--:ssa nimiavaruuksien käyttö ei ole \
                                             mahdollista.
parserError.unimplementedKeyword =           Huomioi, että varattua sanaa "{0}" ei ole toteutettu \
                                             tässä tulkissa. Et voi silti käyttää sitä muuttujan nimenä.
parserError.bitOperator =                    Huomioi, että C--:ssa bittioperaattorit (erityisesti "{0}") \
                                             ovat kiellettyjä.
parserError.power =                          Jos tarkoitit potenssioperaattoria, kokeile funktiota \
                                             pow(double a, double b).


lexerError.unknownToken =                    tuntematon merkki

eval =                                       Laskettiin {0} {1} {2}, tulos: {3}
unknownAnnotationType =                      Tuntematon annotaatio: {0}
cantWriteToOutputStream =                    Tulostusvirtaan ei voida kirjoittaa.
cantCloseOutputStream =                      Tulostusvirtaa ei voida sulkea.
cantFlushOutputStream =                      Tulostusvirtaa ei voida tyhjentää.
cantReadFromInputStream =                    Syötevirrasta ei voida lukea.
cantMakeCalculationsWithBoolean =            Totuusarvoilla ei voi tehdä laskutoimituksia.
valueOfExpression =                          Lausekkeen arvo: {0}, tyyppi: {1}
```

```
pointer =                                {0}-osoitin
reference =                              {0}-viite
array =                                  {0}-taulukko
const =                                  const {0}
vector =                                 {0}-vektori
function =                               funktio
struct =                                 tietue
enum =                                   luettelotyyppi

error.variableAlreadyDeclared =          Muuttuja "{0}" on jo esitelty tällä näkyvyysalueella.
error.invalidCast =                      Tyyppiä "{0}" ei voi muuttaa tyypiksi "{1}".
error.arraySizeNotGiven =                Taulukon kokoa ei ole annettu.
error.arraySizeNotConstant =             Taulukon koko ei ole vakio.
error.tooManyParametersForArray =        Liian monta parametria annettu taulukolle sen alustuksessa.
error.invalidArraySize =                 Virheellinen taulukon koko.
error.invalidTypeForLogicalOperator =    Virheellinen operandi ({0}) annettu loogiselle \
                                         operaattorille, odotettiin totuusarvoa.
error.invalidTypeForComparingOperator =  Virheellinen tyyppi ({0}) annettu vertailuoperaattorille.
error.invalidTypesForOperator =          Virheelliset tyypit ({0} ja {1}) operaattorille {2}.
error.invalidTypeForOperator =           Virheellinen tyyppi ({0}) operaattorille "{1}".
error.invalidDereference =               Virheellinen tyyppi ({0}) operaattorille "*", odotettiin \
                                         osoitinta.
error.invalidIntegerConstant =           Virheellinen kokonaislukuvakio: {0}
error.invalidFloatingConstant =          Virheellinen liukulukuvakio: {0}
error.invalidCharacterConstant =         Virheellinen merkki: {0}
error.invalidUnicodeCharacter =          Virheellinen unicode-merkki: {0}
error.invalidBooleanConstant =           Virheellinen totuusarvo: {0}
error.variableNotDeclared =              Muuttujaa "{0}" ei ole esitelty.
error.noSuchStructField =                Tietueella "{0}" ei ole kenttää "{1}".
error.cantPrintType =                    Tyypin "{0}" muuttujaa ei voi tulostaa cout-virtaan.
error.cantReadType =                     Tyypin "{0}" muuttujaa ei voi lukea cin-virrasta.
error.invalidCondition =                 Virheellinen tyyppi ({0}) ehtolausekkeena, odotettiin \
                                         totuusarvoa.
error.caseExpressionNotConstant =        Valintalauseen vaihtoehdon tulee olla vakio.
error.invalidLValue =                    Sijoituksen kohteena olevaan muuttujaan ei voi sijoittaa.
error.invalidLValueInCin =               Virheellinen muuttuja luettaessa cin-virrasta.
error.cantAssignToConst =                Vakioon ei voi sijoittaa.
error.cantChangeConstValue =             Vakiomuuttujan arvoa ei voi muuttaa.
error.referenceNotInitialized =          Viitettä "{0}" ei ole alustettu.
error.variableIsNotArray =               Operaattoria [] ei voi käyttää lausekkeeseen, \
                                         joka ei ole taulukko.
error.parameterMustHaveDefaultValue =    Parametrilla {0} ei ole oletusarvoa. Lisää se tai poista \
                                         oletusarvo edellisiltä parametreilta.
error.notAFunction =                     "{0}" ei ole funktio.
error.tooMuchParametersForFunction =     Liian monta parametria annettu funktiolle {0} {1}.
error.notEnoughParametersForFunction =   Liian vähän parametreja annettu funktiolle {0} {1}.
error.mainReturnTypeNotInt =             Main-funktion paluuarvo tulee olla kokonaisluku (int).
error.mainFunctionCantHaveParameters =   Tässä tulkissa main-funktio ei voi ottaa parametreja.
error.stringDoesntHaveMember =           Merkkijonolla ei ole jäsenmuuttujaa "{0}".
error.vectorDoesntHaveMember =           Vektorilla ei ole jäsenmuuttujaa "{0}".
error.stringDoesntHaveMethod =           Merkkijonolla ei ole jäsenfunktiota "{0}".
error.vectorDoesntHaveMethod =           Vektorilla ei ole jäsenfunktiota "{0}".
error.structCantHaveMethods =            Tietueella ei voi olla jäsenfunktioita.
error.notAType =                         "{0}" ei ole kelvollinen tyyppi.
error.notAPointer =                      Operaattoria "->" ei voi käyttää lausekkeeseen, \
                                         joka ei ole osoitin.
error.cantUseAccessOperator =            Operaattoria "{0}" ei voi käyttää lausekkeeseen, \
                                         jonka tyyppi on {1}.
error.wrongAmountOfInitParamsToStruct =  Virheellinen määrä parametreja tietueelle "{0}" alustuksessa.
```

```
error.tooManyInitParameters =               Liian monta parametria muuttujalle "{0}" alustuksessa.
error.wrongAmountOfInitParamsArrayStruct =  Virheellinen määrä parametreja alustuksessa.
error.mustReturnValue =                     Funktiosta, jonka paluutyyppi ei ole void, on \
                                            palautettava arvo.
error.functionMustReturnValue =             Funktion "{0}" kaikki suorituspolut eivät palauta \
                                            tyypin "{1}" arvoa.
error.breakNotInLoop =                      Break-lause ei ole silmukan tai switch-rakenteen sisällä.
error.continueNotInLoop =                   Continue-lause ei ole silmukan sisällä.
error.invalidReferenceValue =               Virhe viitteen "{0}" alustuksessa: viite ei voi \
                                            viitata väliaikaiseen muuttujaan.
error.invalidReferenceAssignment =          Virhe viitteeseen sijoituksessa: viite ei voi \
                                            viitata väliaikaiseen muuttujaan.
error.invalidRefValInFunctionCall =         Virhe funktion kutsussa parametrin {0} välityksessä: \
                                            viite ei voi viitata väliaikaiseen muuttujaan.
error.invalidConstReferenceAssignment =     Virhe viitteen "{0}" alustuksessa: Viite, joka ei ole \
                                            vakio, ei voi viitata vakiomuuttujaan.
error.invalidConstRefValInFuncCall =        Virhe funktion kutsussa parametrin {0} välityksessä: viite, \
                                            joka ei ole vakio, ei voi viitata vakiomuuttujaan.
error.cantCallNonConstMethod =              Vakiolle ei voi kutsua ei-vakiometodia "{0}".
error.multipleDefaults =                    Monta default-lohkoa samassa valintalauseessa.
error.loopInStructs =                       Tietueiden esittelyissä on silmukka.
error.invalidPointerValue =                 Virheellinen osoittimen arvo sijoituksessa.
error.invalidPointerValueInFuncCall =       Virheellinen osoittimen arvo funktion kutsussa \
                                            parametrin {0} välityksessä.
error.variableNotConstant =                 Muuttuja ei ole vakio.
error.cantPutTypeInCinBuffer =              Tyypin "{0}" muuttujaa ei voi laittaa cin-virran puskuriin.
error.invalidArraySize =                    Virheellinen taulukon koko.
error.caseAlreadyUsed =                     Vakioa "{0}" on jo käytetty aiemmassa case-lohkossa.
error.divideByZero =                        Jako nollalla.
error.returnNotInAFunction =                Return-lause ei ole funktion sisällä.
error.globalStatements =                    Globaalit lauseet ja lausekkeet ovat kiellettyjä.
error.mainFunctionNotFound =                Funktiota "main" ei löytynyt.
error.notAVariable =                        "{0}" ei ole muuttuja.
error.functionNotDeclared =                 Funktiota "{0}" ei ole esitelty.
error.functionDoesNotHaveBody =             Funktiolla "{0}" ei ole toteutusta.
error.parameterAlreadyHasDefaultValue =     Parametrilla {0} on jo oletusarvo.
error.funcAlreadyDefined =                  Funktio "{0}" on jo määritelty.
error.funcDoesntMatchWithPrev =             Funktion "{0}" esittely ei vastaa aiempaa esittelyä. \
                                            Huomaa, että C--:ssa funktioiden kuormittaminen ei \
                                            ole sallittua.
error.addressOfNonLValue =                  Väliaikaisesta muuttujasta tai literaalista ei voi \
                                            ottaa osoitetta.
error.enumVariableSameAsEnum =              Luettelotyyppivakion "{0}" nimi on sama kuin vastaavan \
                                            luettelotyypin nimi.
error.invalidFunctionCall =                 Virheellinen funktion kutsu.
error.invalidEscapeSequence =               Virheellinen koodinvaihtomerkki: {0}
error.invalidCastInFuncCall =               Virheellinen parametri annettu funktiolle {0} {1}. \
                                            Tyyppiä "{2}" ei voi muuttaa tyypiksi "{3}".
error.declarationInCaseOrDefault =          Et voi esitellä muuttujaa tässä, koska hyppy seuraavaan \
                                            case-lohkoon ylittäisi esittelyn. Laita aaltosulut \
                                            esittelyn ympärille.
error.structMemberInitialized =             Tietueen kenttiä ei voi alustaa.
error.invalidTypeName =                     Virheellinen tyypin nimi.

warning.variableNotInitialized =            Muuttujaa "{0}" ei ole alustettu.
warning.variableHidesAnotherVariable =      Muuttuja "{0}" piilottaa aiemman muuttujan.
warning.unreachableCode =                   Tätä koodia ei voida koskaan suorittaa.
warning.emptyStatement =                    Löydettiin tyhjä lause. Poista tarpeeton puolipiste.
warning.variableNotUsed =                   Muuttujaa "{0}" ei ole käytetty.
```

```
warning.functionNotUsed =                   Funktiota "{0}" ei koskaan kutsuta.
warning.defaultCaseNotLast =                Default-lohko ei ole viimeisenä valintalauseessa.
warning.controlFlowsToThisCase =            Ohjelman suoritus valuu edellisestä lohkosta tähän lohkoon. \
                                            Lisää break-lause edelliseen lohkoon.
warning.globalVariable =                    Globaali muuttuja "{0}".
warning.symbolWasOverwritten =              Symboli "{0}" korvattiin. Vanha tyyppi: {1}, \
                                            uusi tyyppi: {2}.

info.usePreIncrement =                      Käytä jälkiliiteoperaattorin sijaan etuliiteoperaattoria, \
                                            koska se on tehokkaampaa.
info.usePreDecrement =                      Käytä jälkiliiteoperaattorin sijaan etuliiteoperaattoria, \
                                            koska se on tehokkaampaa.
info.variableBeginsWithUpperCase =          Muuttuja "{0}" alkaa isolla kirjaimella.
info.constVariableNameNotUpperCase =        Vakion "{0}" nimi ei ole isoilla kirjaimilla.
info.enumDoesntBeginWithUpperCase =         Luettelotyypin "{0}" nimi ei ala isolla kirjaimella.
info.structDoesntBeginWithUpperCase =       Tietueen "{0}" nimi ei ala isolla kirjaimella.
info.functionBeginsWithUpperCase =          Funktion "{0}" nimi alkaa isolla kirjaimella.

runtimeError.overindexing =                 Yli-indeksointi: yritettiin hakea alkiota indeksillä {0}. \
                                            Indeksoidun muuttujan koko oli {1}.
runtimeError.pointerOutOfBounds =           Osoitin on rajojen ulkopuolella.
runtimeError.tooDeepRecursion =             Liian syvä rekursio.
runtimeError.divideByZero =                 Jako nollalla.
runtimeError.nullPointer =                  Osoitin oli tyhjä.
runtimeError.uninitializedVariable =        Yritettiin käyttää alustamatonta muuttujaa "{0}".
runtimeError.infiniteLoop =                 Ohjelma on luultavasti päättymättömässä silmukassa.
runtimeError.mainFunctionNotFound =         Funktiota "main()" ei löytynyt.
runtimeError.functionDoesNotHaveBody =      Funktiolla "{0}" ei ole toteutusta.
runtimeError.returnedReferenceToLocalVar =  Funktion paluuarvo on viite paikalliseen muuttujaan.
runtimeError.returnedPointerToLocalVar =    Funktion paluuarvo on osoitin paikalliseen muuttujaan.

constraint.switch =                         Switch-lause ei ole sallittu.
constraint.pointers =                       Osoittimet eivät ole sallittuja.
constraint.forVariables =                   Silmukkamuuttujien muuttaminen for:in sisällä ei ole \
                                            sallittua.
constraint.functionOrProcedure =            Aliohjelman on oltava joko funktio, joka palauttaa \
                                            arvon mutta joka ei muuta parametrejään, tai proseduuri, \
                                            joka ei palauta arvoa mutta joka voi muuttaa parametrejään.
constraint.globalVariables =                Globaalit muuttujat ja vakiot ovat kiellettyjä.
constraint.arraysInStruct =                 Taulukot ja vektorit tietueiden kenttinä ovat kiellettyjä.

clip.usage = Command Line InterPreter for C--\
\nKäyttö: clip [tiedostonimi]

clip.info = Command Line InterPreter for C--\
\nErikoiskomennot: quit, list, help, clear, show [<regex>], load <tiedosto>,\
\n  save <func> <tiedosto>, saveall <tiedosto>, run, language [<kieli>].

clip.help = Command Line InterPreter for C--, tekijä: {0} {1}\
\n  Lähetä virheraportit ja muut postit osoitteeseen etunimi.sukunimi@tut.fi.\
\nOhje:\
\n  Voit kirjoittaa C++-lauseita komentokehotteeseen, kuin olisit\
\n  main-funktion sisällä. Voit kuitenkin kirjoittaa myös tietueita\
\n  ja funktioita, kuin olisit globaalilla näkyvyysalueella. Kun\
\n  kirjoitat funktiota tai jotain muuta rakennetta, joka ei mahdu\
\n  yhdelle riville, tulkki kysyy lisää rivejä, jolloin sinun on\
\n  kirjoitettava koodisi loppuun.\
\n  Jos skandinaaviset merkit eivät toimi, tarkista, että \
\n  päätteeseesi on määritetty merkkien koodaukseksi UTF-8.\
```

```
\nErikoiskomennot:\
\n  quit             Poistu tulkista\
\n  list             Tulosta näytölle tähän mennessä kirjoitettu koodi\
\n  help             Tulosta näytölle tämä ohjeteksti\
\n  clear            Tyhjennä muistista kaikki muuttujat, funktiot ja tyypit\
\n  show             Näytä kaikki symbolitaulussa olevat muuttujat\
\n  show <regex>     Näytä muuttujat, joiden nimi sopii annettuun lausekkeeseen\
\n  load <tiedosto>  Lataa tiedostossa oleva koodi\
\n  save <f> <t>     Tallentaa funktion toteutuksen tiedostoon\
\n  saveall <t>      Tallentaa kaiken syötetyn koodin tiedostoon\
\n  run              Aja kaikki syötetty koodi\
\n  language         Näytä kaikki käytettävissä olevat kielet\
\n  language <kieli> Vaihda käyttöliittymän kieltä\
\nÄlä kirjoita puolipistettä näiden erikoiskomentojen jälkeen.

clip.prompt =                              clip>
clip.promptContinuing =                    >
clip.clear =                               Kaikki muuttujat, funktiot ja tyypit poistettu.

clip.show =                                Symbolitaulun sisältö:
clip.symbolTableVariable =                 muuttuja {1} {0}, arvo: {2}
clip.symbolTableFunction =                 funktio {1} {0}
clip.symbolTableEnum =                     luettelotyyppi {0}
clip.symbolTableStruct =                   tietue {0}, kentät: {1}
clip.invalidRegularExpression =            Virheellinen säännöllinen lauseke.
clip.emptyLine =                           Annettiin tyhjä rivi. Syötteen lukeminen keskeytettiin.
clip.propertyNotFound =                    Ominaisuutta "{0}" ei löytynyt tiedostosta clip.properties.

clip.load.noFileName =                     Ei tiedostonimeä. Anna käsky muodossa: load <file>
cantReadFile =                             Tiedostoa "{0}" ei löydy tai sitä ei voi lukea.
unsupportedEncoding =                      Tuntematon merkistön koodaus: {0}

clip.save.overwrite =                      Tiedosto "{0}" on jo olemassa. \nHaluatko korvata vanhan \
                                           tiedoston uudella? (1=kyllä, 2=ei):
clip.save.cannotWrite =                    Tiedostoon "{0}" ei voi kirjoittaa.
clip.save.writeCancelled =                 Kirjoitus keskeytettiin.
clip.save.directory =                      "{0}" on hakemisto. Kirjoitus keskeytettiin.
clip.savefunc.syntaxError =                Ei funktiota tai tiedostonimeä. Anna käsky muodossa: \
                                           save <funktio> <tiedosto>
clip.savefunc.functionNotFound =           Funktiota {0} ei löytynyt.
clip.savefunc.successful =                 Funktio {0} tallennettiin onnistuneesti tiedostoon "{1}".
clip.saveall.syntaxError =                 Ei tiedostonimeä. Anna käsky muodossa: saveall <tiedosto>
clip.saveall.successful =                  Kaikki koodi tallennettiin onnistuneesti tiedostoon "{0}".

clip.language.noLanguageGiven =            Kieltä ei annettu. Anna käsky muodossa: language <kieli>
clip.language.availableLanguages =         Käytettävissä olevat kielet:
clip.language.languageChanged =            Kieleksi vaihdettiin suomi.
clip.language.languageNotFound =           Kieltä {0} ei löytynyt tai sitä ei pystytty lataamaan.
```

LIITE 5: ESIMERKKIAJO CLIPILLÄ

```
luoma@punasiipitikka ~/workspace/vip2 $ clip
Command Line InterPreter for C--
Erikoiskomennot: quit, print, help, clear, show [<regex>], load <tiedosto>,
  save <func> <tiedosto>, saveall <tiedosto>, run, language [<kieli>].
clip> help
Command Line InterPreter for C--, tekijä: Harri Luoma 2007
  Lähetä virheraportit ja muut postit osoitteeseen etunimi.sukunimi@tut.fi.
Ohje:
  Voit kirjoittaa c++-lauseita komentokehotteeseen, kuin olisit
  main-funktion sisällä. Voit kuitenkin kirjoittaa myös tietueita
  ja funktioita, kuin olisit globaalilla näkyvyysalueella. Kun
  kirjoitat funktiota tai jotain muuta rakennetta, joka ei mahdu
  yhdelle riville, tulkki kysyy lisää rivejä, jolloin sinun on
  kirjoitettava koodisi loppuun.
  Jos skandinaaviset merkit eivät toimi, tarkista, että
  päätteeseesi on määritetty merkkien koodaukseksi UTF-8.
Komennot:
  quit              Poistu tulkista
  print             Tulosta näytölle tähän mennessä kirjoitettu koodi
  help              Tulosta näytölle tämä ohjeteksti
  clear             Tyhjennä muistista kaikki muuttujat, funktiot ja tyypit
  show              Näytä kaikki symbolitaulussa olevat muuttujat
  show <regex>      Näytä muuttujat, joiden nimi sopii annettuun lausekkeeseen
  load <tiedosto>   Lataa tiedostossa oleva koodi
  save <f> <t>      Tallentaa funktion toteutuksen tiedostoon
  saveall <t>       Tallentaa kaiken syötetyn koodin tiedostoon
  run               Aja kaikki syötetty koodi
  language          Näytä kaikki käytettävissä olevat kielet
  language <kieli>  Vaihda käyttöliittymän kieltä
clip> int järjestä(const vector<int>& vektori) {
> for (int i = 0; i < vektori.size(); ++i) {
>
Annettiin tyhjä rivi. Syötteen lukeminen keskeytettiin.
clip> void swap(int& i, int& j);
clip> int järjestä(vector<int>& vektori) {
> for (int i = 0; i < vektori.size(); ++i) {
> for (int j = i + 1; j < vektori.size(); ++j) {
> if (vektori[j] < vektori[i]) swap(vektori[i], vektori[j]);
> }
> }
> }
Semanttinen virhe rivillä 1: Funktion "järjestä" kaikki suorituspolut eivät
palauta tyypin "int" arvoa.
int järjestä(vector<int>& vektori) {
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
clip> void järjestä(vector<int>& vektori) {
> for (int i = 0; i < vektori.size(); ++i) {
> for (int j = i + 1; j < vektori.size(); j++) {
> if (vektori[j] < vektori[i]) swap(vektori[i], vektori[j]);
> }
```

```
> }
> }
Huomautus riviltä 3: Käytä jälkiliiteoperaattorin sijaan etuliiteoperaattoria,
koska se on tehokkaampaa.
clip> vector<int> esim;
clip> for (int i = 0; i < 10; ++i) esim.push_back(rand());
clip> järjestä(esim);
Ajonaikainen virhe riviltä 4: Funktiolla "swap" ei ole toteutusta.
if (vektori[j] < vektori[i]) swap(vektori[i], vektori[j]);
                             ^^^^
clip> void swap(int& i, int& j) {
> int temp = i;
> i = j;
> j = temp;
> }
clip> järjestä(esim);
clip> cout << esim;
Semanttinen virhe rivillä 1: Tyypin "int-vektori" muuttujaa ei voi tulostaa
cout-virtaan.
cout << esim;
        ^^^^
clip> esim;
Lausekkeen arvo: 2431 3204 4863 7295 9727 10748 12932 20609 24700 26884
clip> print
   1: void swap(int& i, int& j);
   2: void järjestä(vector<int>& vektori) {
   3:   for (int i = 0; i < vektori.size(); ++i) {
   4:     for (int j = i + 1; j < vektori.size(); ++j) {
   5:       if (vektori[j] < vektori[i]) swap(vektori[i], vektori[j]);
   6:     }
   7:   }
   8: }
   9: vector<int> esim;
  10: for (int i = 0; i < 10; ++i) esim.push_back(rand());
  11: void swap(int& i, int& j) {
  12:   int temp = i;
  13:   i = j;
  14:   j = temp;
  15: }
  16: järjestä(esim);
  17: esim;
clip> show
Symbolitaulun sisältö:
  funktio void swap(int-viite i, int-viite j)
  funktio void järjestä(const int-vektori-viite vektori)
  muuttuja int-vektori esim, arvo: 2431 3204 4863 7295 9727 10748 12932 20609
24700 26884
clip> run
Lausekkeen arvo: 3844 8708 10108 10881 14591 14972 15745 19455 24319 31748
clip> RAND_MAX;
Lausekkeen arvo: 32767
```

```
clip> enum Kuukausi { TAMMIKUU, HELMIKUU, MUU };
clip> HELMIKUU;
Lausekkeen arvo: 1
clip> Kuukausi esim = MUU;
Varoitus riviltä 1: Symboli "esim" korvattiin. Vanha tyyppi: int-vektori,
uusi tyyppi: Kuukausi.
clip> cout << esim;
2
clip> switch(esim) {
> case TAMMIKUU: cout << "Kylmää";
> case HELMIKUU: cout << "Lunta"; break;
> default: cout << "Lämmintä";
> }
Varoitus riviltä 3: Ohjelman suoritus valuu edellisestä lohkosta tähän lohkoon.
Lisää break-lause edelliseen lohkoon.
Lämmintä
clip> show
Symbolitaulun sisältö:
  funktio void swap(int-viite i, int-viite j)
  funktio void järjestä(int-vektori-viite vektori)
  luettelotyyppi Kuukausi
  muuttuja Kuukausi TAMMIKUU, arvo: 0
  muuttuja Kuukausi HELMIKUU, arvo: 1
  muuttuja Kuukausi MUU, arvo: 2
  muuttuja Kuukausi esim, arvo: 2
clip> struct Tuoli {
> int jalkoja;
> Kuukausi ostettu;
> }
> ;
clip> Tuoli t;
Varoitus riviltä 1: Muuttujaa "t" ei ole alustettu.
clip> t.ostettu;
Ajonaikainen virhe riviltä 1: Yritettiin käyttää alustamatonta muuttujaa
"t.ostettu".
t.ostettu;
^^^^^^^^^
clip> Tuoli t = { 5, HELMIKUU };
Varoitus riviltä 1: Symboli "t" korvattiin. Vanha tyyppi: Tuoli,
uusi tyyppi: Tuoli.
clip> t.ostettu;
Lausekkeen arvo: 1
clip> vector<Tuoli> tuolit;
clip> for (int i = 0; i < 10; ++i) { Tuoli t = { i, MUU }; tuolit.push_back(t); }
clip> tuolit[5].jalkoja;
Lausekkeen arvo: 5
clip> tuolit.push_back(5);
Semanttinen virhe rivillä 1: Tyyppiä "int" ei voi muuttaa tyypiksi "const Tuoli".
tuolit.push_back(5);
                 ^
clip> struct Tietue { Tietue t; };
```

```
Semanttinen virhe: Tietueiden esittelyissä on silmukka.
clip> language english
Language changed to english.
clip> clear
All variables, functions and types cleared.
clip> int a = 1; int b = 2; int* c = &a; int* d = &b;
clip> ++*d;
clip> *c = *d - b;
clip> cout << a << ' ' << b << ' ' << c << ' ' << d << ' ' << *c << ' ' << *d;
0 3 0x000003b3 0x000003b5 0 3
clip> int kertoma(int n) {
> if (n = 0) return 1;
Parser error at [2, 7]: expecting: 'or', ')', '||'
if (n = 0) return 1;
      ^
clip> int kertoma(int n) {
> if (n == 0) return 1;
> return n * kertoma(n - 1);
> }
clip> kertoma(12);
Value of the expression: 479001600
clip> kertoma(23);
Value of the expression: 862453760
clip> double kertoma(double n) {
> if (n < 0.01) return 1;
> return n * kertoma(n - 1);
> }
Warning from line 1: Symbol "kertoma" meaning changed. Old type: function,
new type: function.
clip> kertoma(23);
Value of the expression: 2.58520e+22
clip> kertoma(234);
Runtime error from line 3: Recursion is too deep.
return n * kertoma(n - 1);
           ^^^^^^^^^^^^^^
clip> log(exp(63));
Value of the expression: 63
clip> sin(sqrt(exp(63)));
Value of the expression: 0.375698
clip> (pow(sin(0.234), 2) + pow(cos(0.234), 2));
Value of the expression: 1
clip> cout << 5 / 0;
Runtime error from line 1: Divide by zero.
cout << 5 / 0;
        ^^^^^
clip> for (int i = 0; i < 1000000; ++i);
Runtime error from line 1: The program is probably in an infinite loop.
for (int i = 0; i < 1000000; ++i);
                               ^
clip>
clip> string s = "testi";
```

```
clip> cin >> s; cout << "Kirjoitit: " << s << endl;
aha
Kirjoitit: aha
clip> class Pelaaja {
Parser error at [1, 1]: expecting: end of input
Note that the functionality of the keyword "class" is not implemented in
this interpreter.
However, you can't use it as an identifier.
class Pelaaja {
^
clip> cout << 5 | 4;
Lexer error at [1, 11]: unknown token: |
Note that bit operators (especially "|") are forbidden in C--.
cout << 5 | 4;
          ^
clip> for (int i = 0; i < 100; ++i) ++i;
Error on line 1: For variables are not allowed to alter inside a for
statement.
clip> break;
Semantic error on line 1: Break statement is not inside a loop or a switch
statement.
break;
^^^^^^
clip> quit
luoma@punasiipitikka ~/workspace/vip2 $
```