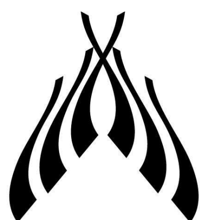
OPINNÄYTETYÖ

ESA NYRHINEN 2012

LAAJENNETTAVA OHJELMOINTIKIELI



Rovaniemen
ammattikorkeakoulu
University of Applied Sciences
LUC

TIETOTEKNIIKAN KOULUTUSOHJELMA

Rovaniemen
ammattikorkeakoulu
University of Applied Sciences
LUC

ROVANIEMEN AMMATTIKORKEAKOULU

TEKNIIKKA JA LIIKENNE

Tietotekniikan koulutusohjelma

Opinnäytetyö

LAAJENNETTAVA OHJELMOINTIKIELI

Esa Nyrhinen

2012

Ohjaaja Erkki Mattila

Hyväksytty ______ 2012 _____________________

Työ on kirjastossa lukusalikappale ja luettavissa Theseus-verkkokirjastossa.

Rovaniemen ammattikorkeakoulu
University of Applied Sciences
LUC

Tekniikka ja liikenne

Tietotekniikan koulutusohjelma

Opinnäytetyön tiivistelmä

Tekijä Esa Nyrhinen Vuosi 2012

Työn nimi Laajennettava ohjelmointikieli
Sivu- ja liitemäärä 53 + 1

Opinnäytetyössä käsitellään uutta laajennettavaa ohjelmointikieltä nimeltään Phase, joka yhdistää ominaisuuksia FORTH- ja LISP-kieliperheistä prototyyppipohjaiseen oliomalliin. Laajennettavuus tarkoittaa sitä, että kieli itse tarjoaa välineet laajentamisekseen esimerkiksi uusien ohjausrakenteiden muodossa.

Työn tavoitteena oli luoda yksinkertainen laajennettava ohjelmointikieli tekijän omaa käyttöä varten, arvioida se ja dokumentoida sen suunnittelu- ja toteutusprosessi.

Työssä käydään läpi kielen synnyn taustalla olevat ideat, toimintamalli, eri kielillä toteutetut prototyypit, miten kieltä voi käyttää ja lopuksi arvioidaan kieltä suhteessa muihin ohjelmointikieliin ja ohjelmointikielten yleisten arviointiperusteiden näkökulmasta.

Ohjelmointikielen prototyyppitoteutuksen luominen onnistui ja itse toteutus osoittautui mielenkiintoiseksi ympäristöksi metaohjelmointiin liittyville kokeiluille. Luotu ohjelmointikieli on vielä monessa suhteessa keskeneräinen ja jatkokehityksen tarpeessa.

Työ esittää vaihtoehtoisen lähestymistavan ohjelmointikielen toteutukseen ja voi auttaa lukijaansa paremmin ymmärtämään ohjelmointikielten suunnittelun ongelmia.

Avainsanat LISP, FORTH, ohjelmointikielet, makrot, olio-ohjelmointi, metaohjelmointi

Muita tietoja Työhön liittyy lähdekoodilistaus.

Rovaniemen ammattikorkeakoulu
University of Applied Sciences
LUC

School of Technology

Information Technology Programme

Abstract of Thesis

Author Esa Nyrhinen

Year 2012

Subject of thesis Extensible Programming Language
Number of pages 53 + 1

This thesis studied a new extensible programming language called Phase which combines features from the FORTH and the LISP language families with a prototype-based object model. Extensibility means that the language itself provides the means to extend itself, for example, in the form of new control structures.

The objective of the thesis was to create a simple extensible programming language for the author's own use, to evaluate it and to document its design and its implementation process.

The thesis studied the ideas behind the birth of the language, the working model of the language, the prototypes written in various languages, how the language can be used and finally compared it to other languages and evaluated it according to the general principles used to evaluate programming languages.

The creation of the prototype implementation of the programming language was a success and it proved to be an interesting environment for experiments on metaprogramming. The created programming language is still incomplete in many ways and in need of further development.

The thesis presented an alternative approach for implementing a programming language and can help its reader better understand the problems of programming language design.

Keywords: LISP, FORTH, programming languages, macros, object-oriented programming, metaprogramming

Special remarks: The thesis includes a source code listing.

SISÄLTÖ

KUVIOLUETTELO 1

LÄHDEKOODILISTAUSLUETTELO 2

TERMI- JA KÄSITELUETTELO 3

1 JOHDANTO 4

2 OHJELMOINTIKIELTEN LUOKITTELU 7

2.1 Ohjelmointikielten rakenne ja laajennettavuus 7

2.2 Luettavuus, kirjoitettavuus, luotettavuus ja hinta 7

2.3 Syntaksi 11

2.4 Koodin esitystapa 12

2.5 Meta- ja makro-ohjelmointi 14

2.6 Olio-ohjelmointi 16

3 KIELEN SUUNNITTELU JA TOTEUTTAMINEN 18

3.1 Alkutilanne 18

3.2 Kielen suunnittelun eteneminen 19

3.3 Prototyyppien toteutus 21

3.4 Raportin kirjoittaminen 22

4 PHASE-OHJELMOINTIKIELI 23

4.1 Phasen yleinen kuvaus 23

4.2 Tyypit ja oliomalli 24

4.3 Syntaksi 29

4.4 Suoritusmalli 30

4.5 Makrot 35

4.6 Phasen arviointi ohjelmointikielenä 38

5 TULOKSET JA JOHTOPÄÄTÖKSET 42

LÄHDELUETTELO 44

KUVIOLUETTELO

Kuvio 1. Luokkapohjainen oliomalli 16
Kuvio 2. Konkatenatiivinen oliomalli 16
Kuvio 3. Delegatiivinen oliomalli 16
Kuvio 4. Tyyppihierarkia UML-luokkakaaviona 24
Kuvio 5. Object-olio 26
Kuvio 6. Tuple-olio 26
Kuvio 7. Symbolitaulu 27
Kuvio 8. Hyvin yksinkertainen ominaisuuspuu 28
Kuvio 9. Monikon sisäinen esitystapa 30
Kuvio 10. Hei maailma! -ohjelman sisäinen esitystapa 31
Kuvio 11. Suoritusmalli UML-aktiviteettikaaviona 32

LÄHDEKOODILISTAUSLUETTELO

Lähdekoodilistaus 1. Sisäliitenotaatio 12
Lähdekoodilistaus 2. Etuliitenotaatio 12
Lähdekoodilistaus 3. Etuliitenotaatio aliohjelmakutsujen tyyliin 12
Lähdekoodilistaus 4. Etuliitenotaatio LISP-tyyliin 12
Lähdekoodilistaus 5. Jälkiliitenotaatio 12
Lähdekoodilistaus 6. Makron ja sen käyttämän apufunktion määrittely . . . 15
Lähdekoodilistaus 7. Makron käyttö 15
Lähdekoodilistaus 8. Makron generoima ja suorittama koodi 15
Lähdekoodilistaus 9. Arvon määritelmän ja sen sisällön haku 28
Lähdekoodilistaus 10. Syntaktiset perusoliot 29
Lähdekoodilistaus 11. Lauselukija Pythoniksi 29
Lähdekoodilistaus 12. Monikon esitystapa 30
Lähdekoodilistaus 13. Hei maailma! Phaseksi 31
Lähdekoodilistaus 14. Vaiheistettu ohjelma 33
Lähdekoodilistaus 15. Vaiheistetun ohjelman kehitys 33
Lähdekoodilistaus 16. Tulkin pääsilmukka Pythoniksi yksinkertaistettuna . . . 34
Lähdekoodilistaus 17. Symbolien ja lainausoperaatioiden käyttö 34
Lähdekoodilistaus 18. Metodien kutsuminen ja attribuutin hakeminen 35
Lähdekoodilistaus 19. dup-aliohjelman luova makrokoodi 36
Lähdekoodilistaus 20. Makrot sulkeuman aloittamiseen ja lopettamisen . . . 37
Lähdekoodilistaus 21. Esimerkki sulkeuman käytöstä 37
Lähdekoodilistaus 22. Hyvin yksinkertainen if- ja endif-makro 38
Lähdekoodilistaus 23. Esimerkki ehdollisesta Hei maailma! -ohjelmasta . . . 38

TERMI- JA KÄSITELUETTELO

Alkio Olion tai tietueen ilmentymälle kuuluva nimetty arvo tai attribuutti.

Haarautumasolmu Solmu, joka viittaa muihin solmuihin.

Hyötykuorma Esimerkiksi olion tai tietueen ilmentymän sisältämä data.

Jäsentäjä, parseri Analysoi selaajalta saamansa lekseemit ja rakentaa niistä rakennepuun ohjelman rakenteen kuvauksena.

Kääreolio Olio, joka toimii rajanpintana toiselle oliolle esimerkiksi muuntaen alkeistietotyypin olioksi.

Lehtisolmu Solmu, joka toimii tietorakenteen haaran loppuna.

Lekseemi, tekstialkio Kielen pienin yksittäinen syntaktinen yksikkö.

Makro Avoin parametrisoitu aliohjelma, jonka kutsu korvataan sen vartalolla, johon on sijoitettu parametrien arvot.

Notaatio Merkintätapa.

Sananen Analysoitu lekseemi, joka toimii solmuna syntaksipuussa.

Selaaja, skanneri, lekseri Tunnistaa syötteestä eli lähdekoodista lekseemit ja antaa ne jäsentäjälle.

Semantiikka Merkitysoppi, joka määrää, mitä kielen sanat ja lausekkeet tarkoittavat.

Solmu Linkitettyyn tietorakenteeseen, kuten vaikka listaan tai puuhun, kuuluva olio, joka on joko haarautuma- tai lehtisolmu.

Syntaksi Lauseoppi eli säännöt siitä, miten erilaiset merkkijonot kielessä luokitellaan lekseemeiksi ja miten noista lekseemeistä saa muodostaa lausekkeita.

Syntaksipuu Ohjelman rakennetta kuvaava puumainen tietorakenne.

UML Unified Modeling Language on ohjelmistoteollisuuden de facto standardi mallinnus- ja kuvauskieli.

Vartijasolmu, valesolmu Tietorakenteen alun tai lopun merkkinä toimiva lehtisolmu, jolla ei ole omaa hyötykuormaa.

1 JOHDANTO

John Backus totesi jo vuosikymmeniä sitten, että sen ajan ohjelmointikielet olivat tarpeettoman isoja, monimutkaisia ja kankeita. Hän myös totesi, että lähes jokainen uusi kieli vain matkii aiempia kuitenkin lisäten ominaisuuksinsa mukaan muutaman uuden ja muodikkaan ominaisuuden. Tämä johtaa lopulta siihen, että nuo kielet vain lisäävät sekaannusta sen suhteen, miten ohjelmat tulisi hahmottaa. On selvä fakta, hän sanoo, että vain harvat kielet tekevät ohjelmien tuottamisesta niin paljon halvempaa ja luotettavampaa, että se oikeuttaa noiden kielten tuottamisen ja opettelemisen. (Backus 1978, 614.)

Tuon jälkeen ovat syntyneet esimerkiksi ohjelmointikielet Ada ja C++, jotka luokitellaan moniparadigmakieliksi. Nämä kielet syntyivät juuri niin kuin Backus totesi eli vain edeltäjiään laajempina sekoituksina aiempien kielten ominaisuuksia sen sijaan, että olisivat tuoneet mukanaan jotakin täysin uutta tai olisivat merkittävällä tavalla yksinkertaistaneet asioita ohjelmistontuotannon näkökulmasta. Kielet, kuten C# ja Java, edustavat uudempia kieliä, jotka pyrkivät yksinkertaistamaan asioita, mutta nekin ovat kehittyessään monimutkaistuneet uusien muodikkaiden ominaisuuksien lisäämisen kautta.

Backus jakaa ohjelmointikielen kahteen osaan: muuttumattomaan kehykseen, joka määrää kielen syntaksin ja semantiikan ja vaihdettaviin osiin eli ohjelman suorituksen aikaisiin kirjastoihin ja käyttäjän oman koodiin. Hän nostaa esille sen, kuinka olisi parempi, jos tuo kehys olisi pieni ja koostuisi ilmaisuvoimaltaan vahvoista ja vaihdettavissa olevista primitiiveistä sen sijaan, että kielessä olisi suurikokoinen kehys, jonka osat ovat ilmaisuvoimaltaan heikkoja ja erotettu toisistaan asettamalla rajoituksia niiden yhdistelemiselle näin vähentäen kielen ilmaisuvoimaa vielä enemmän. (Backus 1978, 617–618.)

Minskyn ohjelmien ja ohjelmointikielten syntaksiin eli lauseoppiin keskitytään liiaksi semantiikan eli merkitysopin hinnalla. Hänen mukaansa vähemmälläkin syntaksilla selviää, kuten esimerkiksi LISP-kielet osoittavat. Minskyn mukaan tutulla monimutkaisella matematiikan syntaksilla on tietysti oma paikkansa käyttöliittymissä, mutta kaikkien kielten pakottaminen staattisen syntaksin muottiin saattaa vain toimia esteenä sellaisten kielten synnylle, joiden syntaksi on käyttäjän muutettavissa. (Minsky 1970, 200–204.)

Perinteinen käsitys ohjelmointikielistä on, että ohjelmointikielen täytyy olla ohjelmoijalle kuin “musta laatikko”, joka abstrahoi sen alla olevan käyttöjärjestelmän ja laitteiston niin, että ohjelmat voidaan kirjoittaa noista riippumattomiksi ja siten mahdollisimman siirrettäväksi eri alustojen välillä. Tämä näkemys on kuitenkin ainakin osaksi kyseenalaistettu, kun on todettu, että mikään yksittäinen abstraktio ei ole optimaalinen kaikkiin niihin käyttötarkoituksiin, joihin sitä voi teoriassa käyttää. “Yksi koko sopii kaikille” -

ajattelu yksinkertaistaa ohjelmointia vähentämällä ohjelmointirajapintojen kokoa, mutta samalla se myös rajoittaa, jos abstraktio ei autakaan tekemään sitä, mitä ohjelmoijan pitäisi saada tehtyä, vaan pakottaa tekemään ylimääräistä työtä tuon abstraktion kiertämiseksi. (Kiczales 1992, 1.)

Tämä sama periaate pätee myös ohjelmointikieliin, sillä nehän myös ovat vain ohjelmia muiden joukossa. Yleiskäyttöiset ohjelmointikielet ovatkin kaikki Turing-täydellisiä eli laskentakyvyltään samalla tasolla kuin Turing-koneet, joilla voi laskea kaiken sen, mikä laskettavissa on. Tämän vuoksi ne ovat yhtä ilmaisuvoimaisia. Ne kuitenkin on suunniteltu ilmaisemaan asioita eri tavoin ja siksi saman asian ilmaiseminen eri kielillä ei ole yhtä vaivatonta. C-kielelläkin voi kirjoittaa luokkapohjaisen olio-ohjelmointimallia mukailevaa koodia, mutta se on paljon helpompaa C++:lla, joka on suunniteltu nimenomaan auttamaan tuollaisen koodin tuottamisessa.

C++:n “isä” Bjarne Stroustroup sanoo, että ohjelmointikieli olisi hyvä suunnitella niin, että laajennukset kieleen tai ainakin niiden toteutukset voisi valita kirjastoista. Standardoidun kielen räätälöitävyys kärsii juuri siitä, että standardointiprosessi on hidas ja kankea tapa saada uusia ominaisuuksia kieleen. Kirjastot ovat paljon nopeampia ja halvempia kehittää ja testata. (Stroustrup 2010.)

Queinnec kysyy, että kuka ei ole unelmoinut keksivänsä (tai voivansa käyttää) kieltä, jossa minkä tahansa asian voi uudelleenmääritellä, mielikuvitus voi laukata suitsematta ja jossa voimme leikkiä täydellisessä ohjelmointivapaudessa ilman ansoja tai esteitä? Tuon unelman mukaisilla järjestelmillä on kuitenkin varjopuolensa. Sellaiset järjestelmät ovat turhauttavan hitaita, lähes mahdottomia kääntää tehokkaasti ja syöksevät ohjelmoijan maailmaan, jossa on vain muutamia lakeja ja tuskin edes painovoimaa. (Queinnec–Callaway 2003, 302.)

Yksi varsinaisista syistä tämän työn aihevalinnalle on oma turhautumiseni olemassaoleviin ohjelmointikieliin, jotka kaikki ovat joko liian isoja, monimutkaisia tai kankeita minun makuuni. Suurin osa kielistä ei salli itsensä laajentamista tilanteen mukaan ja nekin, jotka sallivat, ovat niin monimutkaisia, että en voi varmuudella sanoa täysin ymmärtäväni, mitä koodini oikeastaan tekee tai aiheuttaa. Wirthin mukaan ALGOL-W epäonnistui kielenä juuri siksi, että sen kääntäjä oli niin iso ja monimutkainen, että hän ei kokenut ymmärtävänsä, miten se toimii ja siksi ei kyennyt luottamaan sillä kirjoitettuihin ohjelmiin (Wirth 1985, 161).

Hoaren mukaan ohjelmointi on monimutkaisuuden hallintaa. Jos käytetty kieli on myös monimutkainen, niin siitä tulee pikemminkin osa itse ongelmaa kuin ratkaisua. Jos ohjelmoija ei ymmärrä sekä ratkaistavaa ongelmaa että sen ratkaisemiseen käyttämiänsä työkaluja on turha odottaa hänen aikaansaavan luotettavaa ratkaisua. Hoaren mukaan ainoa lääke, joka auttaa tähän ongelmaan, on turhan monimutkaisuuden välttäminen. (Hoare 1981, 79–82.)

Olen jo pitkän aikaa halunnut ohjelmoida kielellä, joka keskittyisi vähemmän sen rajoittamiseen, miten voin ilmaista erilaisia käsitteitä ja algoritmeja ja enemmän siihen, miten se voisi auttaa minua metaohjelmoinnissa, eli ohjelmia tuottavien ohjelmien kirjoittamisessa. Olennaista on se, että metaohjelmointia voisi harjoittaa ohjelman ajon aikana ilman, että täytyisi turvautua ympäristön ulkopuolisiin ohjelmiin.

Hoare puhuu laajennettavista kielistä kuvaten niitä kieliksi, jotka tekevät ohjelmoijalle mahdolliseksi määritellä omia tietorakenteitaan ja niitä käsittelevät operaatiot. Hän sanoo, että tällaisen kielen tulisi välttää syntaksilaajennuksia ja automaattisia tyyppimuunnoksia, jotka saattavat muodostua esteeksi laajennettavuudelle. Monet ovat yrittäneet rakentaa tällaisia kieliä, mutta Hoaren mukaan useimmat kielten suunnittelijat ovat lopulta vältelleet tällaisen kielen vaatimia teknisiä yksinkertaistuksia. (Hoare 1989, 206.)

Tämä opinnäytetyö keskittyy kysymykseen, onko mahdollista luoda sellainen ohjelmointikieli, jota käyttäjä voi vapaasti laajentaa kirjoitettavan ohjelman vaatimusten mukaan. Yksinkertainen kieli, jonka ilmaisuvoimaa ei rajoita ennenaikainen suorituskyvyn optimointi ja on kokonaisuudessaan helppo oppia ja ymmärtää kenelle tahansa kokeneelle ohjelmoijalle. Tämä yksinkertaisuuden tavoittelu tarkoittaa myös sitä, että ei ole mitään aikomusta vältellä noita Hoaren mainitsemia yksinkertaistuksia, vaan sen sijaan pikemminkin pyrkiä yksinkertaistamaan kieltä vielä vaadittua enemmän, jos vaikka tuon kautta siitä voisi saada vielä laajennettavamman.

Pohjana työlle käytän jo olemassaolevia laajennettaviksi luonnehdittuja ohjelmointikieliä, kuten esimerkiksi LISPiä ja FORTHia. Toteutuksen kielelle kirjoitan useamman kerran eri kielillä nähdäkseni, että onko se oikeasti helppo toteuttaa käytännössä. Kielelle olen antanut nimen Phase johtuen sen käyttämästä metaohjelmointia helpottavasta tasonotaatiosta, josta kerron enemmän luvussa neljä.

Luvussa kaksi käyn läpi ohjelmointikielten ominaisuuksia läpi yleisellä tasolla, luvussa kolme työn etenemisen vaihe vaiheelta ja luvussa neljä, miten kehittämäni ohjelmointikieli toimii ja tarkastelen kieltä yleisten ohjelmointikielten arviointiperusteiden valossa. Luvussa viisi käyn läpi tekemäni johtopäätökset koko työstä, sen merkityksestä, jatkokehitysmahdollisuuksista ja arvioin työtä itseään prosessina. Ainoana liitteenä on hyvin alkeellinen Python-toteutus tämän työn käsittelemästä ohjelmointikielestä.

2 OHJELMOINTIKIELTEN LUOKITTELU

2.1 Ohjelmointikielten rakenne ja laajennettavuus

Backus jakaa ohjelmointikielten rakenteen kahteen osaan: kehykseen, joka ei muutu, ja vaihdettaviin osiin. Vaihdettavat osat tarkoittavat kirjastoja ja käyttäjän omaa koodia. Kehykseen kuuluu ne välineet, joilla vaihdettavia osia yhdistellään, ja sen pitäisi olla mahdollisimman pieni ja ilmaisuvoimainen (Backus 1978, 617). Näitä kehykseen kuuluvia välineitä kutsumme primitiiveiksi ja ne ovat kielen ydin.

Hyvin yleisellä tasolla ilmaistuna ohjelmointikielen voi ymmärtää primitiiveinä ja mekanismeina niiden yhdistelyyn ja noiden yhdistelmien nimeämiseksi niin, että niitä voi käsitellä yksikköinä samankaltaisella tavalla kuin primitiivejä (Abelson–Sussman–Sussman 1996, 4). Useimmat kielet kuitenkin jakavat kielen parissa esiintyvät elementit ryhmiin esimerkiksi niin, että käyttäjän omia funktioita ja kirjastofunktiota käytetään erilaisella syntaksilla kuin kielen ytimen sisäänrakennettuja mekanismeja. Poikkeuksia ovat esimerkiksi LISP ja FORTH, jotka eivät tee eroa kielen omien ja käyttäjän luomien asioiden välillä. Ohjelmointikielen laajennettavuuteen vaikuttaa se, voiko primitiiveistä rakentaa, mitä tarvitsee.

Laajennettavan ohjelmointikielen, jos sillä on tarkoitus voida käyttää mitä tahansa käyttäjän parhaaksi näkemää ohjelmointityyliä, täytyy valita kaikista ilmaisuvoimaisimat primitiivit, joilla voi helpoiten rakentaa kaikki muut korkeamman tason abstraktiot. Tämä kuitenkin tulee väkisinkin rajoittamaan kielen luettavuudella tavalla tai toisella, jos kaikkea matalan tason toiminnallisuutta ei saada abstrahoitua täysin näkymättömäksi. Olettaen siis, että se on oikeasti pakko kätkeä sen sijaan, että sitä voisi käyttää hyväkseen myös korkeamman abstraktiotason koodissa.

2.2 Luettavuus, kirjoitettavuus, luotettavuus ja hinta

Sebestan mukaan ohjelmointikieliä voidaan arvioida myös tarkastelemalla niiden luettavuutta, kirjoitettavuutta, luotettavuutta ja niihin liittyviä kustannuksia suhteessa niiden ominaisuuksiin. Näitä ominaisuuksia ovat esimerkiksi yksinkertaisuus, ortogonaalisuus eli asioiden erillisyys, ohjausrakenteet, tietotyypit ja tietorakenteet, syntaksi eli lauseoppi, tuki abstraktioille, ilmaisuvoima, tyyppien tarkistus, poikkeusten käsittely ja aliasointi eli samaan asiaan usealla eri nimellä viittaminen. Sebesta tosin huomauttaa, että tällainen jako on karkea eikä kaikki noista kriteereistä ole samanarvoisia. (Sebesta 2008, 8.)

Seuraavaksi kerron lyhyesti vähän tarkemmin siitä, mitä Sebesta asiasta sanoo. Sen jälkeen käyn vielä erikseen itse läpi kielen siirrettävyyden, yleiskäyttöisyyden ja sen, kuinka hyvin kieli on määritelty.

Kielen luettavuus on ymmärrettävä suhteessa siihen, mihin sitä käytetään. Se, onko kieli korkean vai matalan tason kieli jossain tietyssä käytössä, määräytyy sen mukaan, kuinka hyvin se sopii ongelma-alueen kuvaamiseen ja missä määrin se pakottaa käyttäjän kiinnittämään huomiota itse ongelman kannalta merkityksettömiin yksityiskohtiin. Juuri tästä syystä sovellusaluekohtaiset minikielet ovat suosittuja, sillä niillä päästään ohjelmassa korkeamman tason esitykseen itse sovelluksen ongelma-alueen suhteen ja näin voidaan keskittyä olennaiseen.

Kielen yksinkertaisuus riippuu osaksi siihen kuuluvien perusosasten määrästä. Mitä enemmän niitä on, sitä enemmän niiden suhteen on opittavaa ja muistettavaa. Tilanne on samankaltainen ominaisuuksien moninaisuuden suhteen. Mitä useammalla eri tavalla voi kielessä asiat sanoa, sitä enemmän noissa kaikissa eri tavoissa on opettelemista ja muistamista. Operaattoreiden kuormittaminen voi niin parantaa kuin heikentää kielen luettavuutta riippuen siitä, miten kuormitusta käytetään. Sen avulla voi kirjoittaa geneerisempää eli yleisempää ja abstraktimpaa koodia, mutta toisaalta samanaikaisesti ymmärtääkseen, mitä mikäkin operaatio tarkoittaa missäkin asiayhteydessä, vaatii ylimääräistä päättelyä. Yksinkertaisuuden voi viedä liian pitkälle myös toisella tavalla. Esimerkiksi assembly-kielien perusosaset ovat hyvin matalan tason operaatioita, jonka vuoksi ajatusta koodin takana voi olla vaikeampi seurata.

Ortogonaalisuus tulee käsitteenä matematiikasta, jossa se tarkoittaa asioiden riippumattomuutta ja erillisyyttä toisistaan samalla tavalla kuin pisteen sijainti kaksiulotteisessa avaruudessa koostuu x- ja y-koordinaatista, joiden arvot ovat toisistaan riippumattomat. Ohjelmointikielten kohdalla se tarkoittaa yleensä kykyä käyttää ominaisuutta asiayhteydestä riippumatta eli muista ominaisuuksista erillään. Ortogonaalisuuden puute näkyy yleensä poikkeuksina ja rajoituksina kielen säännöissä sen suhteen, miten ominaisuuksia saa yhdistellä toisiinsa. Jokainen noista poikkeuksista ja rajoituksista on myös sääntö, joka pitää erikseen opetella ja vähentää kielen säännöllisyyttä näin monimutkaistaen sitä. Vaikka ortogonaalisuus lisääkin kielen ilmaisuvoimaa sitä on usein hyvä rajoittaa, että ohjelmista ei tule liian monimutkaisia.

Nykyään useimmissa ohjelmointikielissä käytetään tavallisia rakenteisen ohjelmoinnin ohjausrakenteita eli esimerkiksi if-lauseita ja for-silmukoita. Tämä tasoittaa eroja kielten välillä ja toisaalta vähentää ohjausrakenteiden merkitystä eronaiheuttajana kielten luettavuudessa. Näiden ohjausrakenteiden etu on se, että ohjelman suorituksen eteneminen on paljon säännönmukaisempaa ja siksi helpommin seurattavissa kuin ei-rakenteisessa ohjelmoinnissa.

Tietotyypit ja tietorakenteet ovat keskeisessä asemassa ohjelmointikielen luettavuu-

dessa. Tietotyypit, joilla on hyvin kuvaava nimi ja tarkkaan määritelty merkitys, helpottavat ohjelman ymmärtämistä, jos tietotyypin oikea käyttötapa näkyy suoraan sen tyypistä. Sama pätee myös tietorakenteisiin, jotka parhaassa tapauksessa auttavat suoraan näkemään, mitä rakenne kuvaa ja mikä sen merkitys on.

Ohjelmointikielen syntaksi määrää, millaisten nimien käytön kieli sallii ja millaisia rakenteita niistä voi rakentaa. Se, että kieli pakottaa käyttämään lyhyitä nimiä, heikentää luettavuutta, jos on pakko käyttää lyhenteitä tai vähemmän kuvaavia nimiä. Lohkoja rajoittavat merkit myös vaikuttavat luettavuuteen. Esimerkiksi pelkät pareittain sisäkkäin olevat kaarisulut eivät kerro yhtä selkeästi, mikä lohko loppuu missäkin kohdassa kuin esimerkiksi `if` ja `end if`. Jos käyttäjä saa kuormittaa tai uudelleenmääritellä kielen varattuja sanoja on suurempi mahdollisuus sekaannuksiin. Usein ajatellaan, että syntaksin pitäisi vastata semantiikkaa niin, että eri tavalla toimivat asiat myös näyttäisivät erilaisilta.

Kirjoitettavuuteen vaikuttavat samat asiat kuin luettavuuteen, jonka vuoksi kielen kirjoitettavuus ja luettavuus ovat käytännössä hyvin lähellä toisiaan. Kirjoitettavuudessakin olennaisin asia on se, kuinka korkean tason kieli käytettävä kieli on suhteessa ongelmaan. Kielen yksinkertaisuus ja ortogonaalisuus lisäävät kielen ilmaisuvoimaa ja kirjoitettavuutta, mutta samalla voivat aiheuttaa enemmän virheitä, jos niiden kanssa ei ole riittävän varovainen. Kirjoitettavuudessa hyvin tärkeää on se, kuinka hyvin kieli tukee uusien abstraktioiden luomista. Asioiden abstrahoinnissa ydinajatus on yleistäminen kätkemällä yksityiskohtia helppokäyttöisemmän rajapinnan taakse. Esimerkiksi aliohjelmat ja toiminnallisuuden kapselointi luokkaan ovat tavallisimpia tapoja rakentaa abstraktioita. Abstraktioiden kautta päästään kielen ilmaisuvoiman vaikutukseen sen kirjoitettavuuteen. Mitä lyhyemmin ja eksaktimmin kieli antaa ilmaista asian, sitä suurempi sen ilmaisuvoima asian suhteen voidaan ajatelleja olevan.

Ohjelmointikielen luotettavuudessa kyse on siitä, kuinka helposti sillä voi tuottaa mahdollisimman virheettömiä ja luotettavia ohjelmia. Mitä parempi kirjoitettavuus ja luettavuus kielellä on, sitä helpommin sillä pitäisi pystyä luotettavan ohjelmiston tuottamiseen. Tarkka ja ilmaisuvoimainen tyyppijärjestelmä auttaa luotettavien ohjelmien kirjoittamisessa ja sitä enemmän, mitä aiemmin tyyppivirheet huomataan. Kaikki ajonaikaiset virheet eivät ole niin kriittisiä, että ne pakottaisivat lopettamaan ohjelman suorituksen välittömästi. Ohjelman luotettavuuden kannalta on tärkeää, että virhetilanteista voidaan toipua korjaamalla mahdollinen ongelma ja jatkaa ohjelman suoritusta, jos se on mahdollista. On siis tärkeää voida luotettavasti havaita, käsitellä ja jatkaa suoritusta virhetilanteen jälkeen.

Aliasoinnilla tarkoitetaan samaan asiaan viittamista useilla eri nimillä. Aliasoinnin mahdollisuus lisää kielen ilmaisuvoimaa, mutta samalla se voi johtaa vaikeasti havaittaviin virheisiin ja sekaannuksiin, joiden vuoksi usein kielet tavalla tai toisella rajoittavat sitä.

Ohjelmointikieliin ja niiden käyttöön liittyy monenlaisia kustannuksia. Ohjelmointikieli täytyy ennen käyttöä opetella, mikä on sitä enemmän aikaavievää ja kalliimpaa, mitä monimutkaisempi kieli on kyseessä. Ohjelmiston luominen kieltä käyttäen myös maksaa. Korkeamman tason ohjelmointikielten kehittäminen aikoinaan aloitettiin juuri siksi, että voitaisiin halvemmalla tuottaa parempaa ohjelmistoa. Hyvin monien ohjelmointikielten toteutukset on toteutettu kääntäjinä, joiden käyttäminen ei sekään ole ilmaista eikä aina välttämättä kovin tehokastakaan ohjelmistotuotannon näkökulmasta, jos kääntäjä on hidas. Hyvin hidas kääntäjä voi tehokkaasti estää kielen laajempaa käyttöönottoa. Kääntäminen varsinkin optimoivalla kääntäjällä alentaa ohjelmiston suorittamisen eli käytön hintaa. Optimointiin kuitenkin sisältyy vaihtokauppa hitaamman kääntämisen ja nopeamman suorituksen välillä. Nykyään laitteiston ollessa halvempaa kuin ennen on ohjelmiston kehittämiskustannukset usein suuremmat kuin sen käyttökustannukset, minkä vuoksi vähemmänkin tehokkaita korkeamman tason ohjelmointikieliä suositaan. Uuden kielen suunnitteluun ja toteutukseen liittyy myös oma hintansa, joka riippuu sen monimutkaisuudesta ja siitä, millä kielellä se toteutetaan. Ohjelmointikielikin voi olla liian kallis toteuttaa käytännössä. Ohjelmointikielen käyttööotto voi olla kallista riippuen siitä, mitä se vaatii. Ohjelmiston huonosta laadusta johtuva huono luotettavuus voi myös käydä hyvin kalliiksi esimerkiksi menetetyn asiakasluottamuksen muodossa, vaikka ei kyse olisikaan kriittisestä järjestelmästä. Ohjelmiston ylläpitäminen korjaamisen ja jatkokehittämisen muodossa ei myös ole ilmaista. Ohjelmiston lähdekoodin huono luettavuus voi tehdä tämän hyvinkin kalliiksi ja vaivalloiseksi.

Siirrettävyys on ohjelmiston uudelleenkäytettävyyden muoto, joka vähentää tarvetta kirjoittaa ohjelmisto uusiksi eri alustoille. Uudelleenkäytettävyyden etu on se, että ohjelmisto täytyy kehittää vain kerran, joka vähentää kustannuksia niin sen kehityksen kuin ylläpidon kohdalla.

Ohjelmointikielen yleiskäyttöisyys myös tekee koodista uudelleenkäytettävämpää sen kautta, että yleiskäyttöistä kieltä käytetään moneen tarkoitukseen ja siksi on paremmin saatavilla eri tarkoituksiin kirjoitettua koodia, jota voi uudelleenkäyttää toiseen asiaan liittyvän ongelman ratkaisemiseen. Jos kieli ei ole yleisohjelmointikieli, niin sillä kirjoitettua koodia on luonnollisesti vaikeampi uudelleenkäyttää. Yksi esimerkki käytöltään rajoitetummasta ohjelmointikielestä on SQL, joka toimii vain relaatiotietokannoille tehtävissä kyselyissä eikä sillä siksi yksinkertaisesti vain ole järkevää yrittää tehdä juurikaan muuta.

Ohjelmointikielen luotettavuuteen vaikuttaa se, onko kieli muodollisesti hyvin määritelty. Jos kieli ei ole hyvin määritelty, ei välttämättä voi varmuudella tietää, miten se käyttäytyy niissä tilanteissa, joita ei ole määritelty. Siirrettävyyskin kärsii siitä, jos kielen toteutukset eri alustoilla toimivat eri tavalla.

2.3 Syntaksi

Ohjelmointikielten syntaksi tai sen osat voidaan luokitella sen mukaan, miten operandit sijoittuvat lähdekoodissa suhteessa operaatioon. Operaatio tarkoittaa sitä, mitä tehdään, ja operandit ovat ne arvot, joille operaatio tehdään. Tavallisen ohjelmoinnin terminologian mukaisesti voisi myös puhua funktiosta ja sille annetuista argumenteista. Esimerkiksi laskutoimituksessa `1 + 2` operaatio on `+` ja operandit ovat `1` ja `2`. Operaatio voi olla operandien keskellä, ennen niitä tai niiden jälkeen. Näitä notaatioita eli merkintätapoja kutsutaan sisä-, etu- ja jälkiliitenotaatioiksi eli infiksi-, prefiksi- ja suffiksinotaatioiksi.

Useimmat yleisesti käytössä olevat ohjelmointikielet eivät käytä vain yhtä näistä, vaan sekoittavat niitä keskenään tarpeen mukaan pyrkiessään syntaksiin, joka vastaa kielen suunnittelijoiden tavoitteita. Yleensä näihin tavoitteisiin kuuluu luonnollisesti se, että syntaksi olisi käyttäjille mahdollisimman tuttu esimerkiksi näyttämällä samankaltaiselta kuin tavallinen matemaattinen notaatio, vaikka se aiheuttaisikin sekaannusta uusien käyttäjien joukossa, koska se käytännössä tarkoittaa jotain aivan muuta (Wirth 2002, 1).

Syntaksin tunnistamista ja käsittelemistä varten ohjelmointikielet sisältävät yleensä selaajan ja jäsentäjän, joita kutsutaan myös skanneriksi ja parseriksi. Selaajaa kutsutaan myös lekseriksi joissakin teksteissä. Selaajan tehtävä on tunnistaa ja lukea sille annetusta datasta, joka on yleensä tekstimuodossa olevaa lähdekoodia, tekstialkioita eli lekseemejä. Jäsentäjä analysoi lekseemit ja tekee niistä tokeneita eli sanasia, joiden perusteella se rakentaa tietorakenteen, joka esittää ohjelman sisältöä ja rakennetta. Tätä tietorakennetta kutsutaan syntaksipuuksi ja sen monimutkaisuus vaihtelee kielen syntaksin monimutkaisuuden mukaan.

Listauksissa 1, 2, 3, 4 ja 5 näkyy eri kielillä kirjoitettuna sama laskutoimitus. Etuliitenotaatiota kutsutaan myös puolalaiseksi notaatioksi ja jälkiliitenotaatiota käänteiseksi puolalaiseksi notaatioksi. Käänteistä puolalaista notaatiota käytetään esimerkiksi joissakin Hewlett-Packardin kalliimmissa teknisissä laskimissa ja FORTH-ohjelmointikielessä. LISP-kielet yleensä käyttävät pelkästään puolalaista notaatiota.

Kaikki näistä notaatioista eivät ole yhtä yksinkertaisia käsitellä ja ymmärtää. Sisäliitenotaatio on näistä monimutkaisin lukea, jos mitataan monimutkaisuutta notaatiolla kirjoitettujen lauseiden yksiselitteiseen tulkitsemiseen tarvittavien sääntöjen määrällä. Jokainen näistä notaatioista edellyttää joko operaatioiden vaatimien operandien määrän tietämistä etukäteen tai jonkinlaista merkintää koodissa, jolla yksiselitteisesti kerrotaan, mikä operandi kuuluu millekin operaatiolle. Sisäliitenotaation lukeminen edellyttää myös operaatioiden presedenssin eli arvojärjestyksen tietämistä. Etu- ja jälkiliitenotaatiossa ei ole mitään tarvetta operaatioiden erilliselle arvojärjestykselle, kos-

ka operaatioiden suorittamisjärjestys näkyy aina yksiselitteisesti koodista, ellei ole ole käytössä jotain epätavallisia sääntöjä. Kielessä, jossa käytetään näitä kaikkia sekaisin, on tilanne vielä monimutkaisempi, sillä silloin pitää huomioida myös operaatioiden assosiatiivisuus eli se, kummalla puolella olevaan operandiin esimerkiksi yhden argumentin operaatiot viittavat.

Listauksen 1 kaltaisen syntaksin jäsentäminen yksiselitteisesti tehdään yleensä rakentamalla syntaksipuu siitä. Listausten 2, 3 ja 4 jäsentäminen vaatii myös syntaksipuun rakentamista, vaikka se ei olekaan niiden kohdalla yhtä monimutkaista. Yksinkertaisin jäsentämisen puolesta näistä notaatioista on listauksen 5 jälkiliitesyntaksi, joka ei vaadi syntaksipuun rakentamista vaan vain yhden pinon käyttämistä operandien ja operaatioiden tulosten välittämiseen operaatioiden välillä.

```
(1 + 2) * 3 / -(4 - 5)
```

Lähdekoodilistaus 1. Sisäliitenotaatio

```
/ * + 1 2 3 (- - 4 5)
```

Lähdekoodilistaus 2. Etuliitenotaatio

```
/(*(+(1,2),3),-(-(4,5)))
```

Lähdekoodilistaus 3. Etuliitenotaatio aliohjelmakutsujen tyyliin

```
(/ (* (+ 1 2) 3) (- (- 4 5)))
```

Lähdekoodilistaus 4. Etuliitenotaatio LISP-tyyliin

```
1 2 + 3 * 0 4 5 - - /
```

Lähdekoodilistaus 5. Jälkiliitenotaatio

2.4 Koodin esitystapa

Nykyaikaiset tietokoneet voi nähdä universaaleina Turing-koneina eli koneina, jotka jäljittelevät niille annetun ohjelman kuvaaman Turing-koneen toimintaa. Yleisellä tasolla ajateltuna ainoa ero koodin ja datan välillä on se, että koodia varten on olemassa kone, joka osaa suorittaa sitä. Koodin ja datan välinen raja on epäselvä, sillä jos datalla on jonkin kielen kuvaama merkitys, niin sille on mahdollista rakentaa sitä käsittelevä

kone, joka voi joko suorittaa sitä koodina tai tehdä sillä jotain muuta. Tämä ei periaatteen tasolla eroa mitenkään siitä, että kuka tahansa voi käyttää ymmärtämällään kielellä kirjoitettua hernekeiton reseptiä joko hernekeiton tekemiseen suorittamalla sen antamat ohjeet tai käyttämällä sitä datana hernekeittoreseptejä vertaillessaan.

Tietokoneohjelma voidaan esittää hyvin monella eri tavalla ja nuo esitystavatkin voi luokitella monella eri tavalla. Tietokoneohjelman esitystapa myös vaihtelee sen käytön ja elinkaaren aikana. Ensin saattaa olla pelkkä joukko spesifikaatiodokumentteja, jotka kuvaavat sen toiminnan ja ominaisuudet esimerkiksi UML-kuvauskieltä käyttäen. Sen jälkeen se voidaan kääntää toiselle kielelle, kuten esimerkiksi jollekin ohjelmointikielelle koodaamalla sille toteutus. Tämän jälkeen se voidaan esimerkiksi kääntää konekielelle ja todennäköisesti tuon käännösprosessin aikana ohjelma esitetään monin eri tavoin tietokoneen muistissa ennen sen lopullisen muodon tuottamista.

Lähdekoodimuotoa lukuunottamatta tietokoneohjelmien koodi esitetään yleensä esimerkiksi konekoodina, tavukoodina, säikeistettynä koodina tai jonkinlaisena syntaksipuuna. Konekoodi tarkoittaa binäärimuodossa olevaa konekielistä koodia, jota laitteiston prosessori osaa tulkita. Tavukoodi muistuttaa hyvin läheisesti konekoodia, mutta sitä ei tulkitse itse laitteisto vaan virtuaalikone eli toinen ohjelma, joka toimii laitteiston prosessorin kaltaisessa tehtävässä. Säikeistetty koodi muistuttaa konekoodia ja tavukoodia, mutta ei käytä niiden tavoin kiinteää käskykantaa, vaan kutsuttavien aliohjelmien muistiosoitteita. Säikeistetty koodi koostuu niiden aliohjelmien osoitteista, jotka muodostavat koodin käyttämän kielen, ja sen kuvaamat ohjelmat suoritetaan kutsumalla koodin kuvaamassa järjestyksessä niitä aliohjelmia, joihin siinä viitataan.

Tästä eteenpäin käytän termejä ulkoinen ja sisäinen esitystapa. Ulkoisella esitystavalla tarkoitan ohjelmointikielen lauseoppia ja sitä heijastelevaa ohjelmien lähdekoodin rakennetta. Sisäisellä esitystavalla tarkoitan kaikkia niitä erilaisia tietorakenteita, joilla ohjelma ja sen eri osat esitetään ajonaikaisesti kielen tulkin tai kääntäjän muistissa.

Kaikki ohjelmointikielet voidaan luokitella sen mukaan, kuinka homoikonisia ne ovat. Mitä homoikonisempi kieli on, sitä lähempänä ohjelmien sisäinen ja ulkoinen esitystapa ovat toisiaan (Mooers–Deutsch 1965, 232). Esimerkiksi yksinkertaiset LISP-kielet käyttävät samaa listoihin perustuvaa ohjelmien esitystapaa sekä sisäisesti että ulkoisesti, joka tekee näistä kielistä hyvin vahvasti homoikonisia. Homoikonisuuden suurin etu on se, että data ja koodi ovat hyvin pitkälti vaihdettavissa keskenään, mikä helpottaa esimerkiksi koodin generointiin perustuvaa metaohjelmointia huomattavasti.

2.5 Meta- ja makro-ohjelmointi

Metaohjelmointi kirjaimellisesti tarkoittaa ohjelmoinnin ohjelmointia, jota kaikki ohjelmoijat tekevät kaiken aikaa. Useimmiten tätä ei kuitenkaan tiedosteta, koska perinteiset ohjelmointikielet pitävät yksinkertaisuuden vuoksi asian piilossa. Esimerkiksi C-kielessä tietueen tai funktion prototyypin määritelmä edustavat deklaratiivista kääntäjän toiminnan ohjelmointia, jonka voi nähdä metaohjelmointina. Ne eivät kerro siitä, mitä ohjelma tekee, vaan sen rakenteesta. Muita tavallisia metaohjelmoinnin tekniikkoja ovat esimerkiksi C-esiprosessorin makrot ja C++:n luokkamallit.

Cheatham jakaa makrot kolmeen eri tyyppiin sen mukaan, missä vaiheessa ohjelman käännös- tai tulkintaprosessia niitä käytetään: tekstimakrot, syntaksimakrot ja laskennalliset makrot. C-kielen esiprosessorin makrot edustavat tekstimakroja, jotka käsittelevät ohjelman lähdekoodia ennen kuin se syötetään eteenpäin skannerille ja parserille. Syntaksimakrot muokkaavat ohjelman syntaksipuuta skannauksen ja jäsennyksen aikana. Laskennalliset makrot toimivat vasta sen jälkeen, kun ohjelma on luettu syntaksipuuksi. (Cheatham, Jr. 1966, 630.)

Common Lisp -ohjelmoijat käyttävät normaalisti laskennallisia makroja, joilla on mahdollista kielen muiden ominaisuuksien vuoksi helposti manipuloida ja generoida ohjelmia. LISP-makrot ovat siis ohjelmia tuottavia ohjelmia. Näiden kirjoittamisessa LISP-ohjelmoijia auttaa backquote-operaattori, jonka avulla on helppo kirjoittaa kirjoittaa koodilohkoja datana. Backquote-operaattorin tarkoitus on tehdä mahdolliseksi kirjoittaa koodia generoivaa koodia, joka näyttää mahdollisimman paljon generoitavalta koodilta. Tämä tekee koodia generoivasta koodista paljon helppolukuisempaa. (Bawden 1999.)

Backquote-operaattori ei itse kuitenkaan ole yksinkertainen toteuttaa eikä ymmärtää. Steelen mukaan backquote ei ole kovin intuitiivinen, mutta pitää sitä silti paljon parempana kuin tunnettuja vaihtoehtoja (Steele, Jr. 1990, 530). Tämä myös selittää, miksi käytännössä aina, kun muita kieliä on yritetty laajentaa LISP-tyylisillä laskennallisilla makroilla, on niihin lisätty myös vastine backquote-operaattorille, jota ilman makrot ovat hyvin työläitä kirjoittaa (Weise–Crew 1993, 157). LISP-tyylisissä makroissa ovat kuitenkin omat ongelmansa, kuten Flatt huomauttaa sanoessaan, että makrojen rakentaminen makrojen päälle luo ongelmia niiden keskinäisten riippuvuuksien setvimisen tarpeen muodossa (Flatt 2002, 1). Graham tuo puolestaan esille sen, että backquote-operaattorin käyttäminen monessa tasossa sisäkkäin tunnetusti tekee ohjelmista hankalia ymmärtää (Graham 1993, 214).

LISP-kielissä, kuten Common Lispissä ja Schemessä makrojen käyttäminen on tavallinen tapa laajentaa ja muokata kielen syntaksia tarpeen mukaan. LISP-makrot eivät kuitenkaan ole sama asia kuin esimerkiksi C-kielen esiprosessorin makrot. Ero on sii-

nä, että C-esiprosessori on oma C:stä erillinen kielensä ja käsittelee syötteenä saamaansa ohjelmaa vain tekstin tasolla toisin kuin LISP-makrot, jotka kirjoitetaan LISPillä itsellään ja ne käsittelevät LISP-ohjelmia tavallisina LISP-olioina, kuten esimerkiksi listoina, numeroina ja symboleina. LISPissä makro on funktio, joka ei laske saamiensa argumenttien arvoja, vaan jonka paluuarvo suoritetaan. Toisinsanoen makro on funktio, joka vastaanottaa koodia ja jonka paluuarvo on sen paluuarvona palauttaman koodin suorituksen paluuarvo. Listauksessa 6 määritellään makro, jota listauksessa 7 käytetään. Listaus 8 näyttää, millaista koodia listauksen 7 makrokutsu generoi ja suorittaa.

```
(defun make-attr (p)
    (if (listp p)
        (append p `(:accessor ,(car p) :initarg ,(intern
            (symbol-name (car p)) 'keyword)))
        `(,p :accessor ,p :initarg ,(intern
            (symbol-name p) 'keyword))))

(defmacro stdclass (name parents attrs)
    `(defclass ,name ,parents
        (,@(loop for p in attrs collect (make-attr p)))))
```

Lähdekoodilistaus 6. Makron ja sen käyttämän apufunktion määrittely

```
(stdclass vm ()
    (ip cp memory env stack cstack layer sources))
```

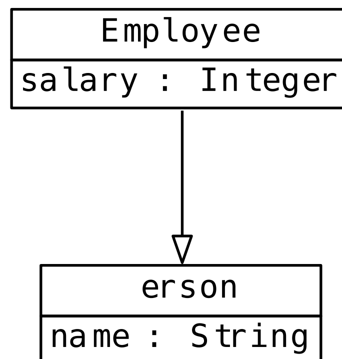
Lähdekoodilistaus 7. Makron käyttö

```
(DEFCLASS VM NIL
    ((IP :ACCESSOR IP :INITARG :IP)
     (CP :ACCESSOR CP :INITARG :CP)
     (MEMORY :ACCESSOR MEMORY :INITARG :MEMORY)
     (ENV :ACCESSOR ENV :INITARG :ENV)
     (STACK :ACCESSOR STACK :INITARG :STACK)
     (CSTACK :ACCESSOR CSTACK :INITARG :CSTACK)
     (LAYER :ACCESSOR LAYER :INITARG :LAYER)
     (SOURCES :ACCESSOR SOURCES :INITARG :SOURCES)))
```

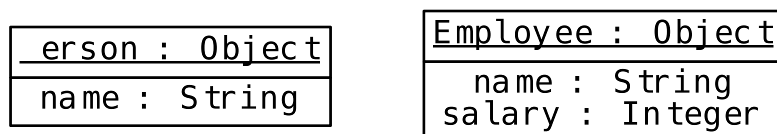
Lähdekoodilistaus 8. Makron generoima ja suorittama koodi

2.6 Olio-ohjelmointi

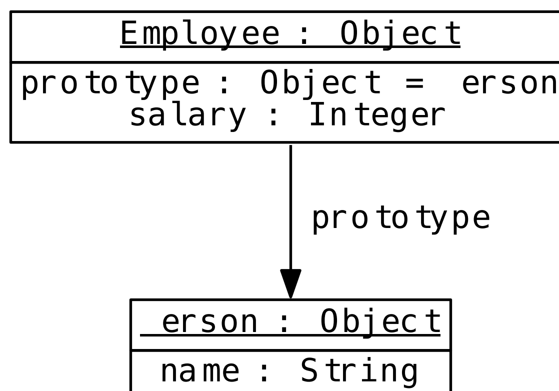
Olio-ohjelmointi yleensä samaistetaan luokkapohjaiseen olio-ohjelmointiin, jonka käyttämä staattinen oliomalli on suosituin laajimmin käytetyissä oliokielissä, kuten esimerkiksi Javassa. Luokkapohjaisessa olio-ohjelmoinnissa olioiden, eli luokkien ilmentymien, rakenteen määräävät luokat, jotka toimivat olioiden rakennuspiirrustuksina. Luokat johdetaan toisistaan niin, että aliluokka kertoo vain, miten se eroaa vanhemmistaan. Vaihtoehtoisia mekanismeja luokkapohjaisen mallin käyttämän johtamisen tilalle periytymisen toteuttamiseksi ovat konkatenaatio ja delegaatio. Molempia näistä käytetään myös luokkapohjaisessa mallissa, mutta siinä ne ovat vain sivuroolissa. Nämä kolme oliomallia esitetään kaavioissa 1, 2 ja 3.



Kuvio 1. Luokkapohjainen oliomalli



Kuvio 2. Konkatenatiivinen oliomalli



Kuvio 3. Delegatiivinen oliomalli

Konkatenatiivisessa olio-ohjelmoinnissa oliot luodaan aiemmista olioista kopioimalla, joka voi olla kielestä riippuen joko matalakopiointi tai syväkopiointia. Konkatenatiivisessa oliokielessä ei siis luokkia ole, vaan vain hyvin vapaamuotoisia oliota, jotka muistuttavat enemmän avain-arvo-parien joukkoa, ja jotka toimivat esimerkkeinä uusia olioita luodessa. (Taivalsaari–Moore–Noble 1999, 28–34.)

Delegatiivisessa olio-ohjelmoinnissa oliot jakavat attribuutit ja metodit delegaation kautta. Käytännössä tämä tarkoittaa sitä, että jos oliolta pyydetään attribuuttia tai metodia, jota sillä ei ole, se delegoi pyynnön toiselle oliolle, joka toimii sen prototyyppinä. Periytymishierarkia delegatiivisissa kielissä muodostuu prototyyppien ketjun kautta, joka muodostuu sen kautta, että jokainen olio hierarkian juuriolioita lukuunottamatta määrittelee oman prototyyppinsä. Esimerkiksi metodikutsun kohdalla itse metodikutsua ei kuitenkaan delegoida vaan vain pelkkä metodin haku, jolloin metodia voidaan käyttää alkuperäiseen olioon eikä johonkin olioon sen prototyyppien ketjussa. (Taivalsaari 1995, 23.)

Olio-ohjelmointiin liittyy muitakin peruskäsitteitä periytymisen lisäksi, kuten esimerkiksi kapselointi, tiedon kätkeminen ja myöhäisen sidonnan kautta toteutettavat metodikutsut. Kapselointi tarkoittaa tiedon ja sitä käsittelevien metodien yhdistämistä moduuliksi esimerkiksi luokkamääritelmän muodossa. Tiedon kätkeminen voidaan käsittää abstrahointina, jossa jonkin asian toteutuksen yksityiskohdat ja monimutkaisuudet piilotetaan yksinkertaisemman rajapinnan taakse. Metodikutsujen myöhäisellä sidonnalla pyritään siihen, että olion metodia kutsuttaessa sen varsinaisen luokan metodia kutsuttaisiin, vaikka oliota käytettäisiin jonkin sen kantaluokan tyyppisen viittauksen kautta. Olio-ohjelmoinnin ja sen mahdollistaman minimaalisen ominaisuusjoukon määritteleminen ei ole yksinkertaista, minkä vuoksi tutkijat eivät ole täysin samaa mieltä siitä, miten olio-ohjelmointi tulisi tarkalleen määritellä.

3 KIELEN SUUNNITTELU JA TOTEUTTAMINEN

3.1 Alkutilanne

Työn taustalla olleista tekijöistä erityismaininnan ansaitsee neljä asiaa, joista ensimmäinen on tekijän omat kokemukset aina tarpeen tullen makroja käyttäneenä Common Lisp -ohjelmoijana ja turhautuminen siihen, että useimmissa muissa kielissä ei ole LISP-makroja vastaavaa ominaisuutta koodin generoimiseen ohjelman ajon aikana. Tuota puutetta voi tietysti yrittää paikata monella eri tavalla, mutta yksikään muu käyttämäni kieli ei ole oman kokemukseni mukaan yltänyt LISP-makrojen kanssa samalle tasolle ilmaisuvoimassa ja helppokäyttöisyydessä.

Toinen syy oli kyllästyminen Common Lispiin sen monimutkaisuuden vuoksi. Common Lisp ei yksinkertaisesti vain tuntunut niin pieneltä ja yksinkertaiselta, että olisi riittänyt luottamusta siihen, että voisi korjata minkä tahansa vastaan tulevan ongelman, jos kielen toteutus itse olisi jostain syystä ollut rikki tai olisi vaatinut räätälöimistä tiettyä sovellusta varten.

Kolmas syy oli oma halu tehdä vihdoinkin oma ohjelmointikieli, joka sopisi myös matalan tason ohjelmointiin ja olisi FORTHin tavoin niin yksinkertainen, että se olisi järjestelmänä kokonaisuudessaan yhden ihmisen ymmärrettävissä. Olennaisinta tässä siis kuitenkin se, että olisi ohjelmointikieli, josta voisi sanoa, että: "Minä loin sen."Ei kenties kaikista rationaalisin syy tällaisen työn tekemiseen, mutta varmasti hyvin helposti ymmärrettävä sellainen.

Neljäs syy oli tyytymättömyys saatavilla oleviin makroassemblereihin, jotka tarjoavat käyttäjilleen oman erillisen pienen kielen makrojen kirjoittamiseen. Ongelmaksi kaikkien noiden makrokielten kohdalla on osoittautunut ennemmin tai myöhemmin se, että niiden ilmaisuvoima ja ominaisuudet eivät riitä kaikkeen siihen, mitä käyttäjä niillä haluaisi tehdä. Tässä Common Lisp vie voiton useimmista muista ratkaisuista makrojensa kautta, joiden vuoksi kieli itse toimii metakielen tehtävässä koodin generoinnissa, jolloin makroja toteutettessa ei käytössä ole pelkkä minikieli, vaan ihan kokonainen yleiskäyttöinen ohjelmointikieli.

Tehtäväksi ja tavoitteeksi työssä siis muodostui selvittää, miten suunnitella ja toteuttaa ohjelmointikieli, joka tavalla tai toisella tekee mahdolliseksi metaohjelmoinnin Common Lisp -tyylisillä makroilla ja samalla suurinpiirtein samassa yksinkertaisuusluokassa toteutuksen osalta kuin yksinkertainen FORTH-tulkki. Tavoitteisiin ei kuulunut tuotoksen optimointi tuotantokäyttöön, vaan tarkoitus oli vain rakentaa ajatuksen toimivuuden todistava prototyyppi.

3.2 Kielen suunnittelun eteneminen

Aluksi lähtökohtana toimi ajatus yksinkertaisen LISP-tulkin rakentamisesta, mutta se kaatui selaajan ja jäsentähän monimutkaisuuteen ja siihen, että tavoite oli välttää lähtökohtaisesti rakenteellista ohjelmointia sen asettamien rajoitusten vuoksi koodin generoinnissa. LISP vaatii parserin siksi, että LISP-koodi on jäsennysvaiheessa pakko muuntaa lähdekoodista monimutkaiseksi tietorakenteeksi, ennen kuin sen voi kääntää tai suorittaa. LISPin syntaksi lähtökohtaisesti tekee mahdolliseksi koodin käsittelemisen lohkoina, mutta samalla tekee välttämättömäksi backquoten tapaisen mekanismin, jos koodin generoinnin pitää olla mahdollisimman vaivatonta tehdä. Tarkoitus oli kuitenkin välttää monimutkaisuutensa vuoksi backquoten tapaisia lähestymistapoja, joten hylkäsin ajatuksen LISP-tulkista.

Tässä vaiheessa harkitsin FORTH-tulkin tekemistä, koska FORTH ainakin täytti vaatimukset yksinkertaisuuden ja helppotajuisuuden suhteen, mutta ongelmana oli se, miten FORTHiin voisi tuoda LISP-tyyliset makrot. FORTHissa on perinteisesti ollut makrot myös, mutta ne eivät vaikuttaneet siltä, että ne skaalautuisivat samalla tavalla kuin backquote useamman tason rakenteiden kuvaamiseen kerralla. FORTH perinteisesti rajoittaa kussakin koodilohkossa tasojen määrän vain kahteen, joka ei olisi riittänyt. Esille nousi myös kysymys siitä, että olisiko jotenkin mahdollista tuoda kieleen makrot ensimmäisen luokan olioina eli voisiko makroista tavallisia olioita samalla tavalla kuin funktiot ovat olioita esimerkiksi Javascriptissä? Ratkaisuja en tosin vielä tässä vaiheessa löytänyt näihin kysymyksiin.

Seuraava kysymys, joka nousi esille, oli kielen suhde olio-ohjelmointiin. Harkitsin ensin vanhaa ja tuttua luokkapohjaista olio-ohjelmoinnin mallia, mutta se tuntui vain monimutkaistavan kieltä ja sen toteutusta tehden sen arvosta kyseenalaisen tähän tarkoitukseen. LISPissä kaikki arvot ovat oliota tai ainakin käsiteltävissä olioina ja kaikilla niistä on mahdollisesti oma ominaisuuslista, vaikka tuota ominaisuuslistaa ei yleensä käytetäkään olio-ohjelmoinnissa. Tuosta sain kuitenkin sen ajatuksen, että voisin käyttää aivan tavallisia listoja LISP-tyyliin luodakseni puun olioiden attribuuteista ja metodeista ja mahdollisesti jakaa osia noista listoista olioiden kesken ja näin matkia sitä, miten oliot luokkapohjaisissa oliokielissä perivät attribuutteja ja metodeja luokiltaan ja niiden kantaluokilta. Osoittautui, että Javascript toimii samankaltaisella tavalla ja että tällaisia oliokieliä kutsutaan prototyyppipohjaisiksi oliokieliksi. Se, miten tämä pitäisi kielessä toteuttaa, jäi kuitenkin vielä tässä vaiheessa avoimeksi kysymykseksi.

Luettuani vähän laajemmin LISPin backquotesta tuli selväksi, että monet ohjelmoijat pitävät sitä hankalana käyttää sen epäintuitiivisuuden vuoksi, kun lainausta ja sen purkamista esiintyy sisäkkäin useammassa tasossa. Käytännössä tuollainen epäintuitiivisuus johtaa juuri siihen tilanteeseen, josta Backus varoittaa von Neumann -kielten huonona puolena, eli koodin merkitys ei avaudu vain katsomalla sitä, vaan se pitää

suorittaa päässä ymmärtääkseen, mitä se tekee (Backus 1978, 616). Asiassa auttaa se, jos opettelee tavallisimmat käyttötavat ulkoa, mutta jo pelkästään se, että täytyy noin tehdä, todistaa menetelmän liiallisesta monimutkaisuudesta. Backquoten vaatima listamainen syntaksi myös asetti sen käyttökelpoisuuden kyseenalaiseksi, kun tavoitteeksi oli jo muodostunut minimaalinen syntaksin määrä.

Etsiessäni vaihtoehtoja backquotelle huomasin, että sitä oli matkittu esimerkiksi kielissä 'C ja MetaML. MetaML ML-suvun kielenä tietysti vaatii, että backquoten mukana tuleva uusi notaatio voitaisiin sovittaa kielen tyyppijärjestelmään ja 'C:n kohdalla myös kieli vaati tyyppienkin osalta laajennuksen. Tässä vaiheessa valitsin kielelleni nimeksi Phase, kun keksin hyvin yksinkertaisen keinon toteuttaa kielessä koodin generointi jakamalla ohjelmien suoritus useampaan eri vaiheeseen tavalla, joka muistuttaa sitä, miten kääntäjät tavallisesti kääntävät ohjelmat useassa eri vaiheessa. Ero näiden tavallisten monivaiheisten kääntäjien ja Phasen välillä kuitenkin on se, että Phasessa vaiheet ovat suoraan ohjelmoijan hallittavissa.

Minua jäi vaivaamaan 'C:n ja MetaML:n suhteen se kysymys, kuinka helppo niillä olisi kuvata sisäkkäisiä makroja. LISPillä tuo on täysin mahdollista, mutta koodi muuttuu hankalaksi ymmärtää, kun makroja on useampia sisäkkäin. FORTH tekee oman vastaavan mekanisminsa helpommaksi rajoittamalla käytettävissä olevien tasojen määrän kahteen sen kautta, että tavalliset FORTH-ympäristöt voivat olla vain joko käännös- tai tulkintatilassa. Ratkaisuni ongelmaan oli siis eksplisiittinen tasonotaatio, jonka selitän luvussa 4.4.

Ennen siirtymistä varsinaiseen toteutukseen piti vielä suunnitella kielen sisäiset tietorakenteet, joista yritin tehdä mahdollisimman yksinkertaisia niin, että mitään erillistä tukea reflektiolle, eli ohjelman kyvylle tarkistella ja muokata itseään, ei kielessä tarvisi, vaan sisäiset tietorakenteet ja muu vastaava olisi suoraan ohjelmoijan käytettävissä ja tarpeen mukaan myös muokattavissa. Ajatus oli, että jo kielen ydin olisi rakenteeltaan sellainen, että ohjelmoija voisi räätälöidä sitä tarpeen mukaan ohjelman suorituksenkin aikana. Suunnittelun lopputulos oli hyvin LISP-kielten hengen mukainen ja varsin yksinkertainen niin ymmärtää kuin toteuttaa, vaikka ei todennäköisesti kovinkaan tehokas muistinkäytön tai suoritusajan näkökulmasta.

Jo alusta lähtien oli itsestäänselvää, että Phasessa olisi roskankeruu eli automaattinen muistinhallinta niin, että käyttäjän ei tarvitsisi käsin varata ja vapauttaa muistia. Phase on hyvin tuhlaavainen muistinkäytön suhteen tekemällä jatkuvasti tietorakenteita, jotka heitetään heti käytön jälkeen pois. En pitänyt järkevänä enkä realistisena sitä ajatusta, että käyttäjältä voisi vaatia virheettömän muistinhallintakoodin kirjoittamista sellaisen kielen kanssa, joka koko ajan varaa muistia käyttäjältä asiasta kysymättä.

3.3 Prototyyppien toteutus

Ratkaistuani nuo isoimmat kysymykset siirryin sitten painimaan protyyppien luomisen ongelman kanssa. Kirjoitin lopulta neljällä eri kielellä kullakin oman versionsa prototyypistä. Suunnitelmaa kielen rakenteesta piti korjata moneen kertaan ja ennenkaikkea yksinkertaistaa, mikä sinänsä ei mikään huono asia ollut lopputuloksen kannalta. Ominaisuuksien määrä väheni ja tietotyyppien määrä lisääntyi, mutta nuo tietotyypit olisi pitänyt lisätä kuitenkin ennemmin tai myöhemmin.

Ensimmäisen prototyypin kirjoitin C:llä arvellen, että se osoittautuisi helpoksi, jos pystyisin käyttämään osoittimia oikein. Toteutuksesta tuli lopulta lähdekoodiltaan noin 6000 riviä pitkä eli suhteellisen pieni. Ongelmaksi muodostui odotetusti ajonaikaisen tyyppitiedon puute, joka piti toteuttaa tekemällä jokaiselle oliotyypille oma tietue. Kaikilla noista tietueista samanmuotoinen otsikko, joka takaa, että on helppo tarkistaa esimerkiksi olion tyyppi. C:n oma tyyppijärjestelmä asetti omat haasteensa myös sen kautta, että se ei juurikaan auttanut varmistamaan, että muunnokset eri osoitintyyppien välillä menisivät oikein. Lopulta päädyin käyttämään ajonaikaista tyyppitietoa tähänkin ja kaikki typemuunnokset käärin erillisten funktioiden sisälle, jotka varmistavat, että käännösaikainen tyyppitieto on mahdollisimman hyvin käytössä. Muita ongelmia oli makrojen ja variadisten funktioiden ja makrojen yhdistämien toimivalla tavalla esimerkiksi järkevän virheraportoinnin aikaansaamiseksi. Itse virheiden käsittely ja ei paikalliset hypyt hoituivat `setjump` ja `longjump` -funktioilla. Niiden käytössä kuitenkin piti olla hyvin varovainen, että ei syntyisi hankalasti paikallistettavia virheitä. Roskankeruuseen käytin avoimen lähdekoodin Boehmsin konservatiivista roskankeruukirjastoa, koska se oli helposti saatavilla ja yksinkertainen ottaa käyttöön.

Javascript-toteutuksen suurimmaksi ongelmaksi muodostui se, että vanhan selaimen ja toisaalta kielen toteuttaminen toisen korkeamman tason ohjelmointikielen avulla tekivät siitä hyvin hitaan. Tilanne korjaantui kuitenkin uudempaan selaimeen kehitysalustana siirtymällä, koska Javascript-moottorien nopeudet ovat parantuneet hyvin paljon viime vuosina. Javascriptillä toteutettuna kielin toteutus lyheni huomattavasti, niin kuin odotinkin, koska C-version tyyppimäärittelyt hoitavat otsikkotiedostot pystyi jättämään pois. Ei paikallisiin hyppyihin jouduin käyttämään Javascriptin poikkeuksia, joka ei tuntunut mitenkään puhtaimmalta mahdolliselta ratkaisulta. Anonyymit funktiot ja sulkeumat lyhensivät koodin määrää myös huomattavasti. Javascriptin prototyyppipohjaista oliomallia ei kuitenkaan ihmetyksekseni päätynyt käyttämään hyväksi kovinkaan laajalti. Syy tähän oli mahdollisesti se, että sille ei juurikaan lopulta tarvetta ollut, koska kielen toteutus ei vaatinut monimutkaisia tietorakenteita. Javascriptissa itsessään on roskankeruu, jonka vuoksi minun ei tietenkään tarvinnut muistinhallintaan kiinnittää mitään huomiota tätä toteutusta tehdessä. Testatessani kielellä ohjelmointia tällä toteutuksella huomasin ongelmia muunmuassa siinä, miten koodia generoitiin ja

tein sen pohjalta parannuksia kieleen.

Assembly-kielinen toteutus oli aluksi vain yksinkertainen käännös C-kielisestä versiosta. Ensimmäinen huomattava ero syntyi sen kautta, että korvasin tulkin käynnistyessä suoritettavan järjestelmän olioiden alustuskoodin jo valmiiksi esiluoduille oliolla, jotka oli makroassemblerilla hyvin helppo kuvata. Tämä vähensi ohjelman suorituksen alussa suoritettavan koodin määrää huomattavasti. Roskankeruun toteuttamista kieleen en edes harkinnut tämän toteutuksen kanssa, vaan sen sijaan varmistin, että tietorakenteet olisivat yhteensopivia yksinkertaisen roskankerääjän kanssa, jonka käyttäjä voisi koodata ja lisätä kieleen ajonaikana. Työkalujen puute oli se, mikä eniten haittasi tämän toteutuksen koodaamisessa ja debuggaamisessa. Loppuun asti en tätä toteutusta ehtinyt edes tehdä ajanpuutteen vuoksi.

Pythonilla kirjoitettu toteutus oli myös vain yksinkertainen käännös Javascript-versiosta. Python Javascriptin tavoin ei vaadi käyttäjältä muistinhallintakoodin kirjoittamista, joten tässäkin toteutuksessa pystyin helposti sivuuttamaan koko asian. Erityisen ärsyttävä rajoitus ohjelmoidessa tätä toteutusta oli Pythonin lambda-funktioiden rajoittuneisuus. Javascriptilla koodatessa lambda-funktiot ovat aivan tavallisia funktioita, mutta Pythonilla lambda-funktion vartalossa ei saa olla yhtä lauseketta enempää koodia. Tämä pakotti minut kirjoittamaan ylimääräisiä nimettyjä funktioita, vaikka selkeästi nuo funktiot eivät olisi nimeä edes tarvinneet.

3.4 Raportin kirjoittaminen

Raportin kirjoittamisessa ongelmaksi muodostui se, että ei ollut selkeää käsitystä siitä, miten Phasen kaltaista kieltä pitäisi kuvata. Kirjallisuudessa yleiset menetelmät kun on suunniteltu sellaisten kielten kuvaamiseen, joiden rakenne on paljon staattisempi. Luokkakaavioista saatava hyöty on hyvin rajallinen kuvatessa sellaista kieltä, joka nojaa enemmän olioiden ajonaikaiseen yhdistelemiseen kuin staattiseen luokkahierarkiaan. Oliokaaviot lopulta osoittautuivat helppotajuisimmaksi menetelmäksi, vaikka UML-kuvauskieli onkin hyvin suurpiirteinen niiden kuvaamisen suhteen. Kielen kuvaamista hankaloitti lisäksi vielä se, että en tuntenut asianmukaisia välineitä ja notaatiota kielen semantikan kuvaamiseen. Syntaksi ei yksinkertaisuutensa vuoksi ollut ongelma missään vaiheessa. Matemaattisemmatkin kuvausmenetelmät olisivat varmaankin auttaneet, mutta pidin parempana olla käyttämättä niitä, koska niiden tulkinnan selittäminen olisi ollut liian vaivalloista ja aikaavievää.

4 PHASE-OHJELMOINTIKIELI

4.1 Phasen yleinen kuvaus

Tässä luvussa esittelen ja käyn läpi pala palalta Phase-ohjelmointikielen rakenteen, toimintamallin, sen käytön perusteet ja lopuksi tarkastelen sen merkitystä ja arvoa ohjelmointikielenä suhteessa muihin ohjelmointikieliin. Kielen kuvaamiseen käytän metakielenä lähdekoodilistauksissa Pythonia. Lähdekoodilistausten kieli on Phase ellei erikseen toisin mainittu. Kaaviot ovat tavallisia UML-kaavioita, jonka vuoksi en niiden tulkintaa erikseen selitä. Alkeistietotyyppeinä luokka- ja oliokaavioissa käytän tyyppejä `int`, `str` ja `ptr` eli kokonaisluku, merkkijono ja osoitin. Phasessa ei käytetä null-osoittimia, vaan viitataan sitä vastaavaan `none`-olioon. Taulukot merkitään lisäämällä `[]` attribuutin nimen perään. Koska UML ei määrittele, miten oliokaavioissa taulukon alkoista lähtevät assosiaatiot pitäisi nimetä, käytän ALGOL-sukuisten kielten syntaksista tuttua `attribuutti[indeksi]`-muotoa.

Phasessa on FORTHin tavoin sekä parametri- että kontrollipino ja joidenkin LISP-kielten tavoin sekä leksikaalinen että dynaaminen ympäristö muuttujien sidonnoille. Sidonta on olio, joka määrittää muuttujan arvon. Aliohjelmien argumentit ja paluuarvot kulkevat parametripinon kautta ja kontrollipino vastaavasti on tallennuspaikka silmukkalaskureille, paluuosoitteille ja muille ohjelman suorituksen kannalta välttämättömille tiedoille. Phasessa kaikki paikalliset ja globaalit nimet viittaavat sidontoihin noissa ympäristöissä ja nuo ympäristöt ovat vain tavallisia ominaisuuspuita olioiden ominaisuuspuiden tavoin. Esimerkiksi kaikki esimääritellyt sanat ovat käytettävissä dynaamisen ympäristön sidontojen kautta.

Phase käyttää yksinkertaisuuden vuoksi syntaksinaan käänteistä puolalaista notaatiota ja selaajaa, joka toimii ennustavan jäsentäjän kaltaisella tavalla eli ei ikinä peruuta (Aho–Sethi–Ullman 1986, 44–48). Tämän vuoksi ohjelman tekstialkiot voidaan tunnistaa nopeasti ilman monimutkaista laskentaa lähdekoodia merkki kerrallaan lukien. Phasen jäsentäjä lukee ohjelmat lause kerrallaan ja luetut lauseet käsitellään sellaisenaan sana kerrallaan ilman erillistä syntaksipuun tuottavaa jäsennysvaihetta. Phasen jäsentäjä tuottaa vain listan käsiteltäviä olioita, joka vastaa säikeistetyn koodin listaa suoritettavista käskyistä.

Phasessa on automaattinen muistinhallinta eli roskankeruu, jonka vuoksi käyttäjän ei tarvitse huolehtia muistin varaamisesta ja sen vapauttamisesta. Periaatteessa roskankeruuta ei Phasessa tarvita, mutta käytännössä kyllä, koska emme voi olettaa, että tietokoneissa on rajattomasti muistia. En testannut, paljonko Phase käyttää muistia, mutta on syytä olettaa, että se on hyvin tuhlaavainen, jonka vuoksi roskankeruun li-

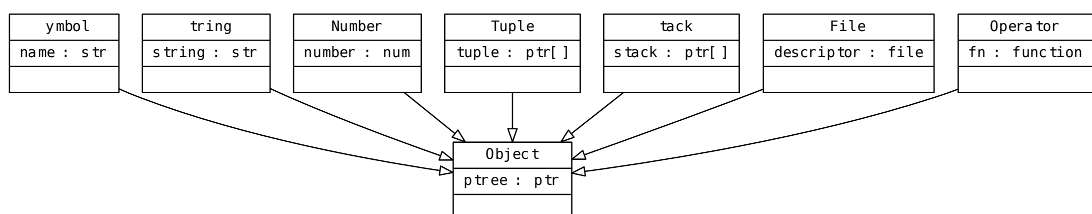
sääminen kieleen on välttämättömyys, jos haluaa suorittaa sillä pitkäaikaisempia prosesseja.

4.2 Tyypit ja oliomalli

Kaaviossa 4 on kuvattuna staattisesti Phasen tietotyypit ja niiden periytyminen toisistaan luokkapohjaisen olio-ohjelmoinnin tyyliin. Staattisella tarkoitan siis käännösaikaista ohjelman rakennetta, joka perustuu nimenomaan luokkien ja tietueiden kiinteään rakenteeseen. Tämä kiinteä rakenne kertoo, miltä luokan tai tietueen ilmentymä näyttää koneen muistissa.

Phasen oma dynaaminen prototyyppipohjainen oliomalli on rikkaampi kuin tämä luokkakaavio ja jokaisen luokan avulla voidaankin toteuttaa useampia erilaisia dynaamisia oliotyyppejä. Dynaamisella rakenteella tarkoitan sitä ajonaikaista rakennetta, joka syntyy luokkien ja tietueiden ilmentymien viitatessa toisiinsa muodostaen verkon toisiinsa viittaavia olioita. Samalla tavalla kuin luokkapohjaisessa oliokielessä tiedetään staattisen rakenteen perusteella, mikä metodi kuuluu millekin oliolle, tiedetään prototyyppipohjaisessa oliokielessä vastaava sen perusteella, mitkä metodeja esittävät oliot ovat löydettävissä seuraamalla viittauksia sovitulla tavalla tuosta oliosta muihin olioihin.

Jos Phase-tulkki toteutetaan oliopohjaisella kielellä, joka tukee luokkapohjaista periytymistä, voi kaavion 4 luokkakaaviota käyttää suoraan toteutuksen luokkahierarkian osana. Phasessa kaikki tietotyyppien ilmentymät ovat olioita lukuunottamatta alkeistietotyyppejä ja nuo alkeistietotyypit ovat aina kääreolion sisällä, jotta kieltä käyttäessä ei tarvitsisi käsitellä ikinä mitään muuta kuin olioita.



Kuvio 4. Tyyppihierarkia UML-luokkakaaviona

Phasen dynaaminen oliomalli ei käytä luokkapohjaisen oliomallin mukaista periytymistä olioiden välillä, vaan sen sijaan oliot, jotka ovat staattiselta rakenteeltaan yhteensopivat, voivat käyttää samoja ominaisuuspuita, joista jokainen sisältää tietynlaisten olioiden kanssa yhteensopivat operaattorit, sanat ja ominaisuudet. Käytännössä tämä operaattorien yhteensopivuus olioiden staattisen rakenteen suhteen tarkoittaa esimerkiksi sitä, että oliot, jotka ovat toteutuskielessä saman luokan ilmentymiä ja voivat siis käyttää samaa metodia, voivat käyttää tuota samaa metodia myös Phase-koodissa.

Ominaisuudet vastaavat staattisten oliokielten olioiden attribuutteja. Operaattorit ovat suoritettavissa olevia toteutuskielen aliohjelmia käärittynä olion sisälle ja sanat ovat linkitetyn listan tapaan rakennettuja ohjelmalohkoja, jotka koostuvat viittauksista operaattoreihin ja muihin sanoihin. Operaattorit ja sanat toimivat siis sekä olioiden metodien tehtävässä että yksittäisinä aliohjelmina ilman omaa käsiteltävää oliota.

Tietotyyppien nimet Phasessa ovat samantyyppiset kuin muissakin kielissä kolmea poikkeusta lukuunottamatta. Nämä poikkeustapaukset ovat `Operator`, `Word` ja `Tuple`. `Operator` on aiemmin selitetty operaattori eli olioon kääritty toteutuskielinen aliohjelma. Se on nimenä lyhyt ja sopii hyvin yhteen syntaksin teorian puolella käytettävien sanojen operaatio ja operandi kanssa. `Word` eli sana on käytössä FORTH-kielen terminologiassa, mutta siellä se tarkoittaa vähän eri asiaa johtuen siitä, että FORTH ei ole joka suhteessa samanlainen kuin Phase. Phasen tapa käyttää tätä sanaa on kuitenkin käyttötarkoitukseltaan sama kuin FORTHissa siinä mielessä, että molemmissa kielissä sana sisältää kaiken sen käyttämiseen tarvittavan tiedon. Phasessa sanaa ei voi ymmärtää ilman tietoa sen tasosta, jota pelkkä viittaus operaatioon tai sanaan ei sisällä.

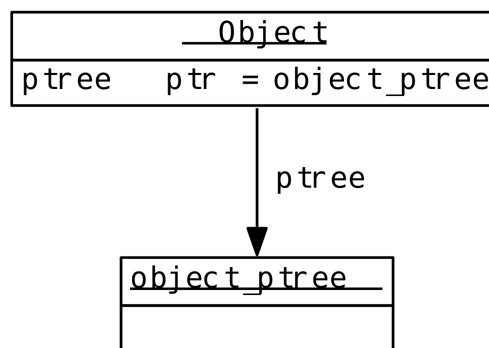
Symboli ei tietotyyppinä ole yleensä tuttu yleisimpien kielten käyttäjille. Symbolit ovat olioita, jotka toimivat esimerkiksi LISP-suvun kielissä muuttujien, funktioiden ja luokkien niminä. Symbolit ovat periaatteessa vain merkkijonoja, mutta toisin kuin merkkijonoja niitä ei verrata toisiinsa sisältönsä eli tekstinsä perusteella vaan sen sijaan identiteetin perusteella. Suorituksen aikana samasta merkkijonosta voi olla muistissa kaksi erillistä kopiota eri osoitteissa ja juuri tuo muistiosoite erottaa ne toisistaan. Phasessa LISPin tavoin luetut symbolit internoidaan käyttämällä apuna symbolitaulua, jossa on listattuna kaikki jo tunnetut symbolit. Merkkijonon internointi tarkoittaa sitä, että merkkijono muutetaan symboliksi joko luomalla uusi symboli tai hakemalla sen niminen jo olemassaoleva symboli symbolitaulusta. Internoinnin tehtävä on varmistaa, että samannimiset symbolit tarkoittavat samaa asiaa.

`Tuple` on käsitteenä tuttu niin relaatiotietokantojen teoriasta, matematiikasta kuin Pythonin ja MLn kaltaisista ohjelmointikielistä. Suomenkielinen vastine tälle sanalle on monikko ja yleensä puhutaankin n-monikoista, jotka voi käsittää n-paikkaisina listoina, yksiulotteisina taulukkoina tai vektoreina. On tavallista puhua esimerkiksi 2-monikosta parina, 3-monikosta kolmikkona ja 4-monikosta nelikkona. Phasen käyttämät monikot ovat järjestettyjä monikkoja, jotka toimivat osoitinlistoina. Pythonissa monikot ovat muuttumattomia listoja. Phasen monikoiden sisältö on muuttuva, mutta koko muuttumaton.

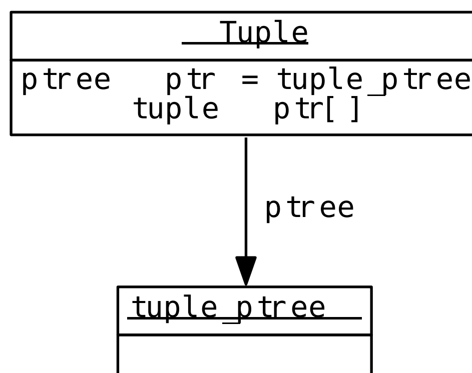
Sanat ovat Phasessa dynaamista tyyppiä `Word` ja staattista tyyppiä `Tuple`. Tämä tarkoittaa sitä, että `Word` on `Tuple`, jonka ptree eli ominaisuuspuuosoitin osoittaa `word_ptree`-olioon, joka on monikoista rakennettu puu, jonka loppu osoittaa

tuple_ptree-olioon, jonka loppu osoittaa object_ptree-olioon. Sana eli Word on 3-monikko, jonka tuple-attribuutti sisältää sen tason, olion ja linkin seuraavaan sanaan tai none-olioon, joka lopettaa koodilohkon. Sanan olio on viittaus joko operaatioon tai toiseen sanaan, jota kutsutaan sanaa suoritettaessa.

Kaavioissa 5 ja 6 ovat oliot tyyppiä Object ja Tuple eli olio ja monikko. Phasessa olion käytöksen määrää sen ominaisuuspuu eli sen staattisen rakenteen ptree-attribuutti, joka kertoo, mitä ominaisuuksia oliolla on ja mitä operaatioita ja sanoja se tukee. Pelkällä oliolla ei ole mukanaan minkäänlaista erityistä hyötykuormaa eli se ei staattisen rakenteensa puolesta tiedä mitään sen enempää. Monikko-olioilla on staattisen rakenteensa puolesta hyötykuormana ominaisuuspuuosoittimen lisäksi tuple-attribuutti, joka on yksiulotteinen taulukko osoittimia, jotka viittaavat muihin olioihin. Numero-oliot tietävät yhden numeron, tiedosto-oliot yhden tiedoston, jne. Phase-olioissa ei staattisen rakenteensa puolesta mitään tämän monimutkaisempaa ole.



Kuvio 5. Object-olio



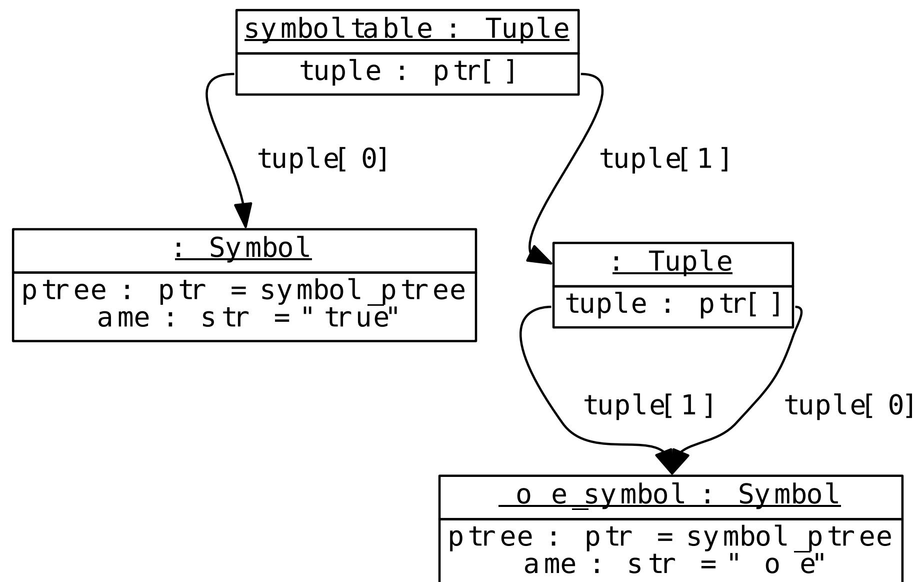
Kuvio 6. Tuple-olio

Olioiden ja monikoiden lisäksi Phasen muut dynaamiset oliotyypit ovat symboli, numero, merkkijono, pino, tiedosto, sana ja operaatio, joista jokainen tarjoaa oliotyypilleen tavalliset ja tarpeelliset ominaisuudet ja operaatiot. Kuten on jo aiemmin mainittu, nämä operaatiot on tallennettu monikoista rakennettuihin ominaisuupuihin, joilla toteute-

taan kielessä kolme erilaista perustietorakennetta: olioiden ominaisuuspuut, nimiavaruuksien vaatimat ympäristöt ja symbolilukijan tarvitsema symbolitaulu.

Yksinkertaisin näistä rakenteista on symbolitaulu, joka on 2-monikoista rakennettu linkitetty lista, jonka solmut eli monikot sisältävät kaksi alkiota: symbolin ja linkin seuraavaan solmuun eli monikkoon. Kun Phasen symbolilukija löytää lähdekoodista symbolin, niin se ensimmäiseksi tarkistaa, onko symbolitaulussa jo olemassa symboli samalla nimellä. Jos on, niin lukija palauttaa tuon olemassaolevan symbolin. Jos ei ole, niin sitten se luo uuden symbolin, lisää sen symbolitauluun ja palauttaa tuon symbolin.

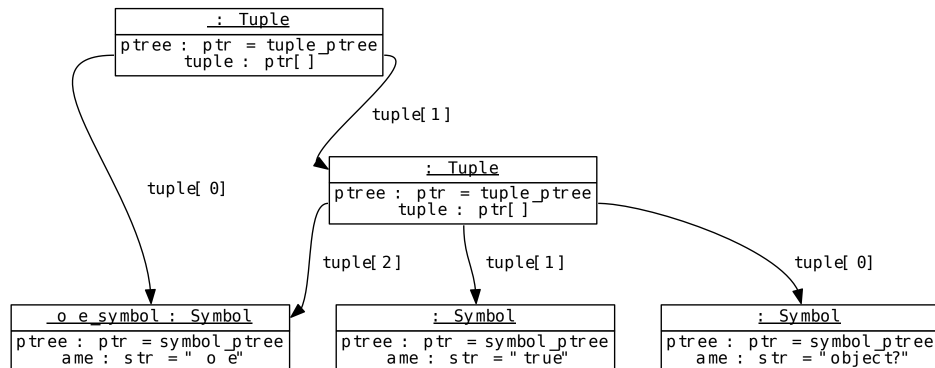
Koska symboleita verrataan toisiinsa identiteetin eikä sisällön perusteella, on siis tärkeää, että normaalisti ei ole kahta samannimistä symbolia. Tämä takaa sen, että symboleja voidaan käyttää muuttujien niminä. Symbolien käyttäminen on yksinkertaistaa kielen toteutusta juuri sen vuoksi, että symboleja voi helposti verrata toisiinsa pelkän identiteetin pohjalta eli muistiosoitteita vertaamalla. Tämä mahdollistaa muuttujan arvon etsimisen paljon nopeammin, sillä osoittimien eli kahden kokonaisluvun toisiinsa vertaaminen on paljon yksinkertaisempaa ja laskennallisesti halvempaa kuin merkkijonojen vertaaminen. Kaaviossa 7 on esimerkki lyhyestä symbolitaulusta, jossa on vain kaksi symbolia: `true` ja `none`. Symboli `none` toimii myös listan lopun merkkinä.



Kuvio 7. Symbolitaulu

Ympäristöt ja olioiden ominaisuuspuut kuvataan Phasessa täsmälleen samanlaisilla rakenteilla, jotka koostuvat pareista eli 2-monikoista, kolmikoista eli 3-monikoista ja puun loppua merkitsevistä viittauksista `none`-olioon. Parit viittaavat kahteen eri ympäristöön tehden puurakenteen mahdolliseksi. Kolmikot viittaavat symboliin, arvoon ja seuraavaan ympäristöön toimien keinona kuvata avain-arvo-pari. Pelkistä pareista koostuvaa listaa kutsutaan listaksi ja kolmikoista koostuvaa listaa sanastoksi. Ominai-

suuspuut ovat näiden kahden sekoitus. Kaaviossa 8 on esimerkki ominaisuuspuusta ja listauksessa 9 on funktio, joka etsii ominaisuuspuusta monikon, jossa avaimena on etsitty symboli.



Kuvio 8. Hyvin yksinkertainen ominaisuuspuu

Listauksen `find` on tavallinen yksinkertainen rekursiivinen ja iteratiivinen funktio. Rekursio tarkoittaa, että funktio kutsuu itseään aliohjelmana, ja iteraatio viittaa funktion ohjausrakenteena käyttämään silmukkaan. Jos annettu `ptree` eli ominaisuuspuu on `none` eli ominaisuuspuun loppua merkitsevä vartijasolmu, niin funktio toteaa, että sidontaa ei löytynyt etsitylle symbolille. Jos annettu ominaisuusopuu on 2-monikko, niin funktio kutsuu itseään monikon ensimmäistä alkiota argumenttina käyttäen. Tuo ensimmäinen alkio viittaa toiseen puuhun, joka tulisi käydä läpi ennen jatkamista alkion toisen alkiona viittaamaan seuraavaan ominaisuuspuun osaan. Jos annettu ominaisuuspuu on 3-monikko, niin sitten verrataan argumenttina saatua symbolia monikon ensimmäiseen alkioon eli avain-arvo-parin avaimeen, joka myös on symboli. Jos kyseessä on sama symboli, niin paluuarvona palautetaan funktiosta samantien koko tuon löydetty monikko, joka sitoo muuttujan nimenä toimivan symbolin sen arvoon.

```
def find (sym,ptree):
    while True:
        if ptree is none: return undefined
        elif len(ptree.tuple) == 2:
            x = find(sym,ptree.tuple[0])
            if x != undefined: return x
            ptree = ptree.tuple[1]
        else: # len(ptree.tuple) == 3
            if ptree.tuple[0] is sym:
                return ptree
            ptree = ptree.tuple[2]
```

Lähdekoodilistaus 9. Arvon määritelmän ja sen sisällön haku

4.3 Syntaksi

```
^0 7 "abc" ;
```

Lähdekoodilistaus 10. Syntaktiset perusoliot

Phasessa ei ole varsinaista jäsentäjää, joka rakentaisi sanasista ohjelman rakennetta esittävän puun olioita. Phasen jäsentäjä on lauselukija, joka pyytää selaajalta tekstialkioita olioina ja muodostaa niistä lauseita. Jokainen on lause on vain monikoista koostuva lista sanoja. Phase on syntaksinsa suhteen hyvin samankaltainen yksinkertaisten FORTHien kanssa. Tämä syntaksin yksinkertaisuus helpottaa makro-ohjelmointia huomattavasti, koska koodin generoinnin ja manipuloinnin rajapinta on hyvin yksinkertainen. Mitä enemmän syntaksia kielessä on, sitä enemmän on erilaisia asioita pakko ottaa huomioon koodia generoitaessa ja manipuloitaessa.

Listauksessa 10 ovat esimerkit kaikista syntaksiin kuuluvista lekseemeistä. `^0` on tasomerkintä, joka kertoo, mikä arvo sen jälkeen luettujen sanojen kopiointilaskuriin laitetaan. `7` on kokonaisluku, `"abc"` on merkkijono ja `;` on symboli. Erona Phasen ja FORTHin välillä on se, että Phase lukiessaan lähdekoodia luo luetuista tokeneista olioita, jotka se laittaa sanoihin yhdessä sanan tasoa edustavan numeron kanssa, joka määräytyy lukijan senhetkisen tiedostokohtaisen tason mukaan. Numerot, symbolit ja merkkijonot tekevät mahdolliseksi esittää kaikki muut oliot ilman erityismekanismeja. Laajemmissa Phasen toteutuksissa voi tietysti olla oma syntaksinsa esimerkiksi kommenteille ja muille mukavuuksille.

Phase-ohjelmat luetaan ja käsitellään lause kerrallaan. Lauseen lopettaa joko tiedoston loppu, jota edustaa `eof`-symboli, tai `;`-symboli. Luettu lause on linkitetty lista sanoja. Sanoista kerron tarkemmin luvussa 4.2. Jäsentäjä toimii listauksen 11 kuvaamalla tavalla. `append`-funktiolle annetaan senhetkinen taso, joka määräytyy edellisen tasomerkinnän mukaan tai sellaisen puuttuessa on 0, ja luettu olio, joista funktio sitten tekee sanan ja lisää sen luetun lauseen loppuun.

```
def read_sentence ():
    while True:
        object = read_object()
        append(level,object)
        if object in (semicolon_symbol,eof_symbol,):
            return
```

Lähdekoodilistaus 11. Lauselukija Pythoniksi

Selaaja, jota käytetään `read_object`-operaation kautta, on toimintamalliltaan ennustava ennustavien parsereiden tapaan eli se ei ikinä peruuta lukuprosessin aikana. Tä-

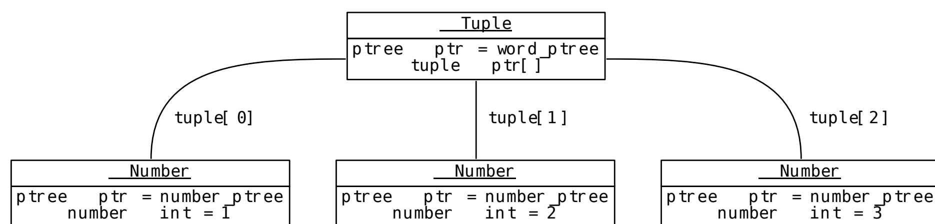
mä pakottaa hyvin yksinkertaisen syntaksin käyttöön siinä kielessä, jota se tunnistaa, ellei ole valmis lisäämään monimutkaisuutta skannerin tilojen lukumäärän kautta. Etu siinä, että se ei peruuta, on se, että silloin sen ei myöskään tarvitse ylläpitää puskuria jo luetuista merkeistä.

Phasen tyypeistä vain symboleilla, numeroilla ja merkkijonoilla on oma selaajan tunnistama notaatio. Muut oliot tietysti kuvataan käyttäen esimerkiksi symboleja, numeroita ja merkkijonoja. Mikään ei estä käyttäjää lisäämästä omia lukijaoperaattoreitaan kieleen, mutta niihin liittyy se rajoitus, että niiden täytyy olla operaattoreita eikä sanoja. Sanat siis ovat tulkittua Phase-koodia toisin kuin operaattorit, joiden toteutus on toteutuskielinen. Tämä rajoitus johtuu siitä, että kielen tämänhetkiset toteutukset eivät osaa suorittaa sanoja lukuvaiheessa.

Oliot, joilla ei ole omaa notaatiotaan, esitetään koodina, jonka suorittamalla olion saisi rakennettua. Syy sille, miksi data esitetään koodina, on se, että niin tekemällä koodin ja datan välinen ero pienenee, joka on haluttava ominaisuus kielessä, jonka tavoitteista yksi on tehdä koodista helposti muokattavaa dataa metaohjelmoinnin helpottamiseksi. Listauksessa 12 on esimerkki siitä, mitä tämä tarkoittaa käytännössä monikon esittämisessä. `(` on symboli, joka toimii vartijasolmuna ja `)` on operaattori, joka ensin katsoo, montako oliota pinossa on ennen päällimmäisintä `(`-symbolia, luo tuonkokoisen monikon, siirtää pinosta siihen nuo oliot ja heittää vartijasolmun pois. Kaavio 9 esittää, miltä kyseinen monikko näyttäisi ajonaikaisena rakenteena muistissa lauselukijan sen luettua.

```
( 1 2 3 )
```

Lähdekoodilistaus 12. Monikon esitystapa



Kuvio 9. Monikon sisäinen esitystapa

4.4 Suoritusmalli

Listauksessa 13 on tuttu “Hei maailma!” -ohjelma Phaseksi ja oliokaaviossa 10 on Phasen jäsentäjän tuottama tuota ohjelmaa esittävä ajonaikainen sisäinen tietorakenne. Sisäisellä ja ulkoisella esitystavalla tarkoitan koodin muotoa sen elinkaaren eri

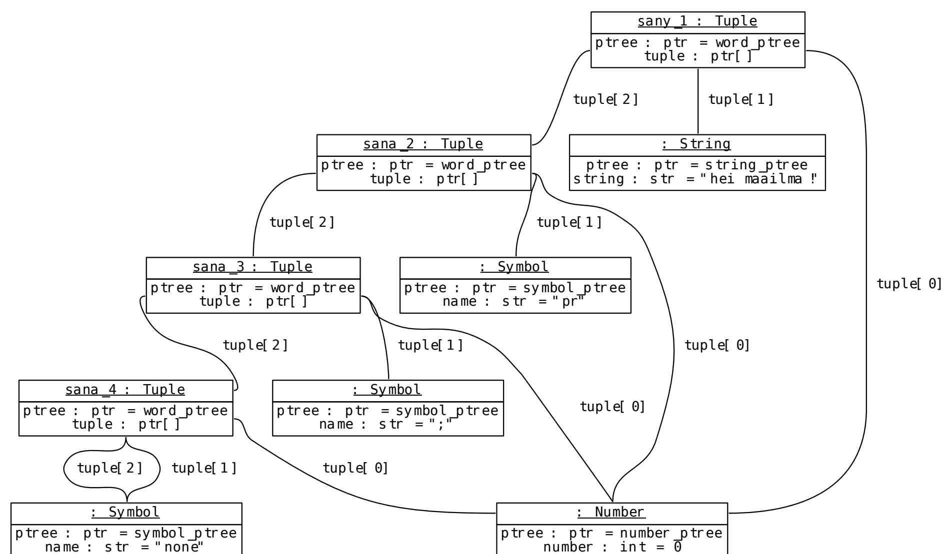
vaiheissa. Ulkoinen esitystapa tarkoittaa koodin ja datan esittämistä lähdekoodina ja sisäinen esitystapa tarkoittaa niiden tietorakenteiden muotoja, jotka lähdekoodista luodaan, kun ne jäsennetään olioiksi ohjelman ajon aikana.

Klassisen LISPin, jossa ohjelmat koostuvat pelkistä lausekkeista, suoritusmallissa käytetään `eval`-funktiota ohjelman paluuarvon laskemiseen, joka keskittää lausekkeiden merkityksen tulkinnan yhteen paikkaan. FORTHissa kaikki sanat suoritetaan suorittamalla aliohjelma, johon sanan otsikko viittaa. Tuo aliohjelma kertoo, miten sanan sisältö tulee tulkita. Phase seuraa FORTHin mallia oliopohjaisella tavalla. Phasessa jokaisella oliolla on `eval`-operaatio, joka on metodi ja määrittelee, mitä tuolle oliolle pitää tehdä suoritettaessa. Näiden `eval`-operaatioiden suhteen Phasessa voi ajatella olevan kolme erilaista sääntöä. Operaattorin suorittaminen suorittaa sen toteutuskielisen aliohjelman, joka on kyseisen operaation olioon kääritty. Sanan suorittaminen hoidetaan kuin aliohjelmakutsu Phasen tasolla eli tallentamalla senhetkinen ohjelmalaskurin osoite, eli lukuosoitteimen arvo, paluuosoitteena kontrollipinoon ja sitten vaihtamalla sen arvon tilalle suoritettava sana. Kaikkien muiden olioiden suorittaminen tarkoittaa yksinkertaisesti sitä, että viite kyseiseen olioon laitetaan parametripinon päälle.

Jotta muuttujien käyttäminen Phasessa olisi mahdollista, symboli yritetään korvata siihen sidotulla arvolla ennen sen suorittamista. Jos symbolia vastaavaa sidontaa ei löydy staattisesta eikä dynaamisesta ympäristöstä, tulkki antaa virheilmoituksen. Molemmat ympäristöt ovat samanlaisia ominaisuuspuita kuin olioiden ominaisuuspuut ja niistä sidontojen hakeminen myös hoidetaan samalla tavalla.

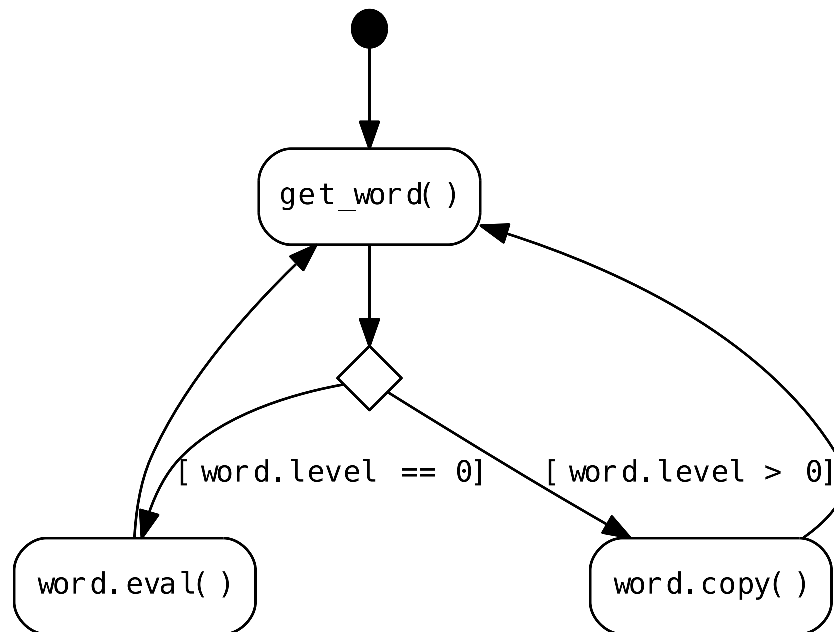
```
^0 "hei maailma!" pr ;
```

Lähdekoodilistaus 13. Hei maailma! Phaseksi



Kuvio 10. Hei maailma! -ohjelman sisäinen esitystapa

Listauksessa 13 on “Hei maailma!” -ohjelma yhtenä Phase-lauseena. Ohjelman suoritus alkaa sanasta `sana_1` ja etenee sanasta sanaan `tuple[2]` linkin eli sanaa esittävän monikon kolmannen alkion sisältämän viittauksen kautta. Jokaisen sanan kohdalla sitä esittävän monikon toisen eli `tuple[1]` alkion viittaama olio suoritetaan suorittamalla sen `eval`-operaatio eli metodi.



Kuvio 11. Suoritusmalli UML-aktiviteettikaaviona

Listauksen 13 ohjelma alkaa tasomerkinnällä `^0`, joka asettaa nollan lukijan senhetkiseksi tasoksi luetulle tiedostolle. Siitä eteenpäin seuraavaan tasomerkintään tai tiedoston loppuun asti taso pysyy samana ja kaikkien luettujen sanojen tasoksi merkitään tuo nolla. Sanojen taso on laskuri, joka kertoo montako kertaa kyseinen sana täytyy kopioida ennen kuin tulkki saa suorittaa sen. Laskurin arvoa tietysti vähennetään yhdellä joka kerta, kun sana kopioidaan. Kun sanan taso on nolla, sitä ei enää kopioida vaan se suoritetaan. Kopiointi tässä asiayhteydessä on verrattavissa lähinnä sanan uudelleenlukemiseen yhtä pienemmällä tasolla. Phasessa ohjelmat luetaan ja suoritetaan lause kerrallaan, minkä vuoksi sanojen kopiot laitetaan luetun lauseen loppuun sen jatkeeksi. Uusi lause luetaan vasta sitten, kun luettu lause kokonaisuudessaan sen itse lisäämät jatkeet mukaanlukien on käsitelty.

Listauksessa 14 on yhtenä lauseena esimerkkiohjelma, jossa käytetään useampia eri tasoja, ja listauksen 15 jokainen rivi kuvastaa, miten tuon ohjelman lause kehittyy suorituksen aikana. Jälkimmäisen listauksen jokainen rivi näyttää siis, mitä koodia edellinen rivi generoi lauseen loppuun. Kutsun tasomerkintää käyttävää vaiheistetuksi koodiksi, koska sillä luotujen lauseiden suorittaminen on helppo hahmottaa vaiheittain.

```
^3 ps pr ^2 3 ^0 2 ^1 1 ^2 0 ^0 ;
```

Lähdekoodilistaus 14. Vaiheistettu ohjelma

```
^2 ps pr ^1 3 ^0 1 ^1 0
^1 ps pr ^0 3 0
^0 ps pr
```

Lähdekoodilistaus 15. Vaiheistetun ohjelman kehitys

Tulkin käynnistyessä luku- ja kirjoituspää osoittavat samaan tyhjään sanaan, joka toimii vartijasolmuna ohjelmien sisäisessä esitystavassa. Tällaisen vartijasolmun käyttö helpottaa huomattavasti koodin lukemista ja kirjoittamista samaan sanoista koostuvaan listaan, sillä tyhjästä sanasta on helppo tehdä suoritettava sana ja se tarjoaa linkkialkion, jonka voi laittaa viittaamaan uuteen tyhjään sanaan. Ilman vartijasolmua täytyisi joka tapauksessa tallentaa viittaus edelliseen kirjoitettuun sanaan enkä löytänyt mitään perustelua sille, miten sen poisjättäminen tekisi kielen toteuttamisesta helpompaa. Vartijasolmu on helppo tunnistaa siitä, että se viittaa `none`-symboliin seuraavana listan solmuna.

Jos ohjelman suorituksen aikana lukupää viittaa vartijasolmuun, tulkki päättelee, että suoritettava koodi loppui kesken, ja yrittää lukea seuraavan lauseen lähdekoodia. Selaaja lukee olioita yksi kerrallaan ja jäsentäjä rakentaa niistä lauseen käärimällä ne sanojen sisään. Selaaja tekee tämän muuttamalla kirjoituspään osoittaman lauseen lopettavan tyhjän sanan suoritettavaksi sanaksi. Se tekee tämän kirjoittamalla senhetkisen lukutason sanan tasolle tarkoitettuun alkioon ja itse luetun olion toiseen alkioon. Sen jälkeen se luo uuden tyhjän sanan ja laittaa tuon vanhan tyhjän sanan kolmannen eli linkkialkion osoittamaan luotuun uuteen tyhjään sanaan. Jäsentäjä tekee näin jokaisen selaajalta saamansa olion kohdalla, kunnes on lisännyt joko tiedoston tai lauseen loppua merkitsevän olion eli joko olion `eof` tai olion `;`.

```
RP = [0, none_symbol, none_symbol]
WP = RP

def main():
    while True:
        if RP[LINK] is none_symbol:
            read_sentence()
            continue
        if RP[LEVEL] > 0:
            WP[OBJECT] = RP[OBJECT]
            WP[LEVEL] = RP[LEVEL] - 1
            WP[LINK] = [0, none_symbol, none_symbol]
            WP = WP[LINK]
        else:
            RP[OBJECT].eval()
        RP = RP[LINK]
```

Lähdekoodilistaus 16. Tulkin pääsilmukka Pythoniksi yksinkertaistettuna

Kun sanan olio suoritetaan sen `eval`-operaatio eli metodi haetaan ja tuo metodi suoritetaan. Yksinkertaisen globaalin operaation suorittaminen, jota vastaa esimerkiksi globaalin funktion kutsuminen C-kielessä, ei ole tuon monimutkaisempaa. Olioiden ominaisuuksien eli attribuuttien arvojen hakeminen ja metodien kutsuminen on Phasessa vähän monimutkaisempaa. LISPin ja FORTHin tavoin Phasessa muuttujan nimen käyttö sellaisenaan johtaa tuon muuttujan arvon hakemiseen, joka ei aina ole se, mitä halutaan. LISPissä symbolit voidaan "lainata" `'`-syntaksilla eli `quote`-funktiolla, joka käytännössä estää sitä seuraavan symbolin arvon laskemisen antaen käyttää symbolia itsenä arvona. Tämä on hyödyllistä, jos haluaa antaa esimerkiksi kutsuttavan operaation nimen toiselle operaatiolle. Esimerkkejä symbolien käytöstä ja niiden arvon hakemisen estämisestä on listauksessa 17. Listauksessa käytetään rivin loppuun yltävien kommenttien syntaksina PHP:sta, Bashista ja muista tuttua `#`-merkillä alkavaa syntaksia.

```
x # Hakee x:n arvon ja suorittaa sen.

' x # Hakee x:n arvon ja laittaa sen
    # pinoon suorittamatta sitä.

'' x # Laittaa symbolin x pinoon.
```

Lähdekoodilistaus 17. Symbolien ja lainausoperaatioiden käyttö

Muita `'`:n ja `''`:n kaltaisia operaatioita ovat ovat esimerkiksi `.`, `..`, `:` ja `:0`. Kaksoispisteellä alkavat symbolit viittaavat operaatioihin, joilla voi kätevämmin suorittaa olioiden

metodeja. `-merkeillä molemmin puolin rajoitetut merkkijonot ovat MySQLstä lainattu tapa merkitä symboleja, joiden nimiin saa kuulua tyhjiä merkkejä. Listauksessa 18 on esimerkkejä erilaisista tavoista hakea olioiden ominaisuuksia ja kutsua niiden operaatioita.

```
# Olion ominaisuuden hakeminen symbolilla x.
`some object` . x ;

# Olion ominaisuuden hakeminen muuttujan
# x sisältämällä symbolilla.
`some object` x .. ;

# Symbolilla nimetyn olion operaation
# suorittaminen ilman argumentteja.
`some object` '' `a method with no arguments` 0 : ;

# Olion `some object` muuttujan x nimeämän operaation
# suorittaminen yhdellä argumentilla.
`some object` x 1 : ;

# Lyhyempi tapa kutsua symbolien nimeämiä metodeja
`some object` :0 `a method with no arguments` ;
`some object` x :1 `a method with 1 argument` ;
`some object` x y :2 `a method with 2 arguments` ;
```

Lähdekoodilistaus 18. Metodien kutsuminen ja attribuutin hakeminen

Olion operaation eli metodin kutsumiseen käytetty `:`-operaatio on toimintaperiaatteeltaan hyvin yksinkertainen. Se ottaa pinosta numeron `n` ja metodin nimenä toimivan symbolin, hakee tuon metodin pinon `n`:nen olion ominaisuuspuusta ja suorittaa sen. Operaatio `:0` on vain lyhenne lauseelle, joka lainaa sitä seuraavan symbolin, laittaa nollan pinoon ja kutsuu `:`-operaatiota.

4.5 Makrot

Phasessa ei ole rakenteisista ohjelmointikielistä tuttuja ohjausrakenteita ja muutenkin on kielenä hyvin minimaalinen ohjausrakenteiden suhteen. Tämän vuoksi Phase kielenä nojaa runsaaseen makrojen käyttöön, joilla paikataan tuota puutetta. Tasomerkintöjen ja sanojen tason hallinnan avulla voidaan vapaasti valita, milloin mikäkin osa koodista suoritetaan tai milloin sitä käytetään koodin generoimiseen. Toisin kuin monissa muissa kielessä Phasessa ei ole mitään varsinaista eroa makrojen ja muunlaisten aliohjelmien välillä. Tämä tasojen hallinta kuitenkin on tehtävä käsin, mikä on juuri se piirre, joka erottaa Phasen useimmista muista kielistä, joihin on hyvin kiinteästi si-

säänrakennettu jo syntaksin tasolla, milloin mikäkin asia tehdään. Tällä tarkoitan sitä, että Phasella ohjelmoidessa on täysin normaalia luoda lopullinen suoritettava ohjelma kerros kerrokselta niin, että edellinen kerros generoi seuraavan. Esimerkiksi Javalla ja muilla tavallisilla käännetyillä kielillä ohjelmoitaessa tilanne on erilainen, sillä koodin generointi ei kuulu niiden perusfilosofiaan eikä yleiseen käytäntöön muuten kuin mahdollisesti esiprosessorien tai erillisten koodia generoivien ohjelmien kautta.

Listauksessa 19 on yksinkertainen ohjelma, jonka suorittaminen luo uuden aliohjelman. `code` luo ensin uuden tyhjän sanan pinon eli parametripinon päälle, josta `@ps :0 dup` ottaa kopion niin, että pinon päällä on kaksi viittausta tuohon samaan tyhjään sanaan. `!'d` luo dynaamiseen ympäristöön sidonnan symbolille `dup` ja laittaa sen arvoksi tuon tyhjän sanan. `pushwp` siirtää nykyisen kirjoitusosoittimen eli kirjoituspään kontrollipinon päälle talteen ja korvaa sen tuolla tyhjällä sanalla.

Seuraavaksi tuleva tason yksi koodi kirjoitetaan kopioimalla tason nolla koodiksi kirjoitusosoittimen osoittaman tyhjän sanan perään. `@ps :0 dup` luo kopion pinon päällimmäisimmästä alkiosta samalla tavalla kuin edellisen rivin koodissa ja `ret` suorittaa paluun aliohjelmasta korvaamalla lukuosoittimen kontrollipinon päältä ottamallaan oliolla. Tämä generoitu koodi siis tietysti suoritetaan vasta, kun generoitua aliohjelmaa kutsutaan.

Lopuksi palataan tasolle nolla, suoritetaan `popwp`, joka palauttaa kontrollipinon päällä olevan arvon kirjoitusosoittimeksi, ja `;` lopettaa lauseen. Todennäköisesti tätä ohjelmaa ei voi kirjoittaa millään toisella toimivalla tavalla, koska ohjelman pätkiminen lauseisiin vaikuttaa sekä luku- että kirjoitusosoittimen toimintaan ja väärän arvon laittaminen kumpaan tahansa niistä voi sotkea koko ohjelman toiminnan peruuttamattomalla tavalla. Lauseen lukemisen jälkeen kirjoitusosoitin osoittaa lauseen viimeisen luetun sanan jälkeiseen vartijasolmuun ja kaikki generoitava koodi kirjoitetaan sen kohdalle, ellei koodi erikseen manipuloi tuota kirjoitusosoitinta jollain tavalla. Jos luku- ja kirjoitusosoitin päätyvät osoittamaan eri listaan lauseen lopuksi tulkki päätyy loputtomaan lukusilmukkaan, mutta ei ole varmaa, voiko tätä ongelmaa edes korjata viemättä ohjelmoijalta mahdollisuutta täysin hallita luku- ja kirjoitusosoittimen toimintaa. Tämän vuoksi kielen tämänhetkinen toimintamalli tekee näinkin helpon virheen tekemisen mahdolliseksi.

```
^0 code @ps :0 dup !'d dup pushwp
^1 @ps :0 dup ret
^0 popwp ;
```

Lähdekoodilistaus 19. dup-aliohjelman luova makrokoodi

Listauksessa 20 on kaksi makroa, joista ensimmäinen luo `[`-aliohjelman ja jälkimmäinen `]`-aliohjelman, joilla voi yhdessä luoda sulkeuman. Sulkeuma on aliohjelma, joka

muistaa luontinsa aikaiset muuttujasidonnat. Listauksessa 21 on esimerkki sulkeuman käytöstä. Listauksen 20 tapa luoda makrot seuraa samaa kaavaa kuin listaus 19, mutta makrojen generoima koodi on paljon monimutkaisempaa, koska sekin on makrokoodia, joka generoi koodia. Koodin elinkaaren eri vaiheiden käsitteleminen yhtäaikaisesti tuo paremmin esille koodin sisäiset riippuvuudet eri vaiheiden välillä, mutta myös hankaloittaa sen ymmärtämistä ihan jo pelkän käsiteltävän tiedon määrän puolesta.

Listauksen 20 makron `[` koodi käsittelee kolmea eri vaihetta ohjelman suorittamisessa: makron luomista, makron käyttämistä ja makron generoiman koodin käyttämistä. Tämä makro luodaan generoimalla sen vartalona toimiva koodi, joka tallennetaan dynaamiseen ympäristöön nimellä `[`. Kun sitä käytetään se generoi koodia, joka muistaa senhetkisen staattisen ympäristön, ja suoritettaessa tuo koodi väliaikaisesti korvaa staattisen ympäristön sen generoinnin aikaisella niin, että makrojen `[` ja `]` välinen koodi voi käyttää tuon staattisen ympäristön muuttujien sidontoja. Tämän vuoksi listauksen 21 sulkeuman sisällä oleva `s pr` koodi muistaa, että muuttujan `s` arvo on `"hello world!"`.

```
^0 code !’d [
^0 ’ [ pushwp
^1 code @ps :0 dup pushwp
^2 @cs @le :1 push ’ ^1 @le 0 , ^2 !le ^1 ret
^0 popwp ;

^0 code !’d ]
^0 ’ ] pushwp
^2 @cs :0 pop !le ret
^1 popwp ret
^0 popwp ;
```

Lähdekoodilistaus 20. Makrot sulkeuman aloittamiseen ja lopettamisen

```
^0 { "hello world!" !’ s
^0 [ ^1 s pr ^0 ]
^0 } !’d hello-world ;

hello-world # Tulostaa tekstin "hello world!"
```

Lähdekoodilistaus 21. Esimerkki sulkeuman käytöstä

Listauksessa 22 on esimerkkinä kaksi hyvin yksinkertaista makroa: `if` ja `endif`. Näillä makroilla voi kirjoittaa hyvin alkeellisia if-lauseita, jotka tekevät mahdolliseksi koodilohkon ohittamisen jonkun ehtona toimivan arvon ollessa tosi. Näiden makrojen ydinajatus on, että `if` kirjoittaa ehdollisen hyppykäskyn ja tyhjän sanan, johon `endif` kirjoittaa hypyn määränpään. Tämän vuoksi `if` hakee kirjoitusosoittimen osoittaman olion

ja jättää sen pinoon `endif`in korjattavaksi. Listauksen 23 Hei maailma! -ohjelma on `if` ja `endif` makrojen sisällä, jotka ovat tasolla nolla kun taas varsinainen ohjelman sisältö on tasolla yksi. Ideana tuossa tietysti on se, että makrojen generoiman koodin täytyy lopulta olla samalla tasolla kuin ohjelman sisällön, että hyppykäsky voisi toimia oikeaan aikaan. Juuri tämä koodi ja metakoodin sekoittuminen on Phasen kaltaisen kielen käyttämisen monimutkaisuuden suurin syy.

```
^0 code !’d if
^0 ’ if pushwp
^2 ?jmp ^1 @cp ^2 none ^1 @cs :1 push ret
^0 popwp ;

^0 code !’d endif
^0 ’ endif pushwp
^1 @cp !word ret
^0 popwp ;
```

Lähdekoodilistaus 22. Hyvin yksinkertainen if- ja endif-makro

```
^1 ‘some boolean value‘ ^0 if
^1 "hello world!" pr
^0 endif ;
```

Lähdekoodilistaus 23. Esimerkki ehdollisesta Hei maailma! -ohjelmasta

4.6 Phasen arviointi ohjelmointikielenä

Phase pyrkii olemaan laajennettava ohjelmointikieli. Tämän tavoitteen saavuttaminen edellyttää sitä, että käyttäjän täytyy pystyä muokkaamaan kieltä ja sen käyttäytymistä tavoilla, joita kielen suunnittelussa ei voida ennakoida ja ottaa huomioon.

LISP, jota on kutsuttu ohjelmoitavaksi ohjelmointikieleksi, pyrkii ohjelmoitavuuteen jättämällä käyttäjälle kieleen koukkuja, joihin hän voi liittää omaa koodiaan räätälöidäkseen kielen toimintaa, ja antamalle käyttäjälle voimakkaat metaohjelmointivälineet koodin generoimiseen (Foderaro 1991, 27).

FORTH pyrkii samaan lopputulokseen olemalla mahdollisimman yksinkertainen ja tarjoamalla käyttäjän käyttöön ne samat työkalut, joita kieli itsekin käyttää oman toiminnallisuutensa määrittelemiseen. FORTH-ohjelmoinnille tyypillistä on muovata kielestä itsestään sovellukselle sopiva korkean tason kieli. Tämä kielipohjainen lähestymistapa on tietysti käytössä muissakin kielissä, kuten LISPissä ja nykyään enemmän suosiossa olevassa Rubyssä.

Jos katsotaan Phasea kielenä Backuksen näkökulmasta eli kehyksenä ja vaihdettavina osina, niin laajennettavuus vaatisi kyvyn laajentaa niin kielen kehystä kuin vaihdettavia osia (Backus 1978, 617). Vaihdettavien osien lisääminen tietenkään ei ole ongelma, vaan itse kehyksen laajentaminen. Tämän ratkaisemiseen Phase käyttää FORTHin lähestymistapaa eli tekemällä kielen ytimestä mahdollisimman yksinkertaisen ja pienen ja toteuttamalla sen samalla tavalla kuin vaihdettavat osat. Ongelma on siis se, miten voidaan tehdä kielen uudelleenohjelmointi ja laajentaminen mahdollisimman vaivattomaksi.

Phase yrittää ratkaista ohjelmoitavuuden ongelman yhdistämällä itsessään LISPin ja FORTHin ominaisuuksia eli koukut, LISP-makrot, yksinkertaisuuden ja reflektion dynaamiseen olio-ohjelmointiin. LISP ja FORTH käyttävät erilaisia, mutta hyvin yksinkertaisia syntakseja. Phase käyttää ulkoisesti FORTHin tapaista syntaksia, mutta sisäisesti esittää koodin samankaltaisella tavalla kuin LISP, jonka tekee laskennallisten makrojen käytön helpoksi. Mahdollisimman yksinkertaisen syntaksin ja semantiikan tarkoitus on tehdä mahdollisimman helpoksi ohjelmien manipulointi ei vain kielen sisällä, vaan myös ulkoisille ohjelmille tavalla, joka on yhteensopiva Unix-filosofian "ohjelmat ovat työkaluja" -ajatuksen kanssa (Kernighan–Plauger 1976, 15–20).

Phase on luettavuudeltaan verrattavissa lähinnä assemblyyn siksi, että kielen tarjoamat perusoperaatiot ovat hyvin matalan tason operaatioita verrattuna esimerkiksi rakenteisen ohjelmoinnin perus ohjausrakenteisiin. Tämän vuoksi koodi on koostuu hyvin helppotajuisista osasista, mutta niillä ei voi välttämättä ilmaista algoritmeja kovinkaan vaivattomasti tai helppotajuisesti. Kielen alimman tason osasten yksinkertaisuus tekee mahdolliseksi toteuttaa ne tehokkaalla tavalla, mutta tuosta ei juurikaan apua, koska tulkitussa kielessä noiden osien tulkitsemisen välillä suoritettavaa tulkin koodia joudutaan suorittamaan useammin näin lisäten ylimääräistä laskentaa.

Phase on laajennettava ohjelmointikieli, joten jos käyttäjästä tuntuu siltä, että hän tarvitsee lisää rakennetta esimerkiksi syntaksin muodossa, niin hän voi yksinkertaisesti lisätä kieleen haluamansa rakenteen toteuttamalla sen laajennuksena kieleen. Toisaalta se, että kieli paljastaa käyttäjälle enemmän tietoa omien rakenteidensa toteutuksesta, ja se, että käyttäjä joutuu usein ainakin vielä ottamaan kantaa noiden rakenteiden toimintaan, tekee kielestä hankalamman käyttää.

Ongelmakohtaisten minikielten luominen tarpeen mukaan on siis mahdollista ja niitä voi käyttää ihan paikallisesti vain tietyssä koodilohkossa ilman, että tarvitsee muuttaa kieltä koko ohjelman laajuisesti aiheuttaen bugeja epäyhteensopivuusongelmien muodossa koodilohkojen kanssa, jotka eivät toimi noiden laajennuksen kanssa. Minikielten käytön hyvä puoli on se, että ne voivat olla ongelma-alueen asioiden suhteen paljon luettavampia kuvaavuutensa vuoksi, mutta toisaalta ne vaativat toteuttamansa uuden notaation opettelun ja ymmärtämisen, mikä vähentää luettavuutta niiden näkö-

kulmasta, jotka eivät tuota notaatiota tunne. Toisaalta koska koodi itsessään on yleisesti tarkasteltuna vain yksi käyttöliittymä muiden joukossa, tuo ongelma ei ole mitenkään erikoislaatuinen tai vain tuohon asiaan rajoittuva. Minikielet ovat kieliä samalla tavalla kuin se kieli, jolla ne on tehty, joten jokainen niistä vaatii käyttäjältä opettelua. Toisaalta kaikki järjestelmät voidaan nähdä kielinä, jotka pitää opetella.

Käänteisen puolalaisen merkintätavan syntaksina käyttäminen tuo mukanaan kuitenkin ongelmia. Se ei ole kaikille tuttu eikä kovin moni ole tottunut lukemaan sitä. Tämä on kenties sen isoin ongelma. Asian kääntöpuoli toisaalta on se, että jälkiliitenotaatio on hyvin yksinkertainen suhteessa yleisesti käytössä olevaan sekanotaatioon, jossa käytetään operaattoreita, joiden assosiatiivisuuteen ja arviointijärjestykseen, eli presedenssiin, liittyvät säännöt täytyy muistaa voidakseen lukea sitä. Tuo sekanotaatio ei ole kovin helppokäyttöinen ja nopealukuinen ja vielä vähemmän sitten, jos täytyy joka tapauksessa eksplisiittisesti kirjata sulkeilla operaatioiden suoritusjärjestys joka kohtaan. Tulee myös muistaa, että sisä- ja etuliitenotaation jäsentäminen vaatii jäsentäjän, joka osaa rakentaa luetuista sanasista oikeanlaisen tietorakenteen. Jälkiliitenotaation jäsentämiseen riittää pelkkä selaaja ja jäsentäjä, joista jäsentäjän ei tarvitse tehdä muuta kuin rakentaa yksiulotteista listaa, joka suoraan kertoo sanasten suoritusjärjestyksen.

Phase kärsii samasta ongelmasta, että nimi voi kuormittamisen vuoksi tarkoittaa montaa eri asiaa riippuen asiayhteydestä, niin kuin muutkin oliokielet. Tällä siis viittaan erityisesti siihen, miten sama nimi voi viitata täysin erilaisiin käytöksiin tai ominaisuuksiin erilaisten olioiden kohdalla. Toisaalta tuo tekee myös mahdolliseksi luonnollisen kielen tavoin saman sanan käyttämisen eri asioista, jos vain osaa erottaa merkityksen asiayhteyden perusteella. Se monimutkaistaa asioita, mutta tekee myös geneerisen koodin kirjoittamisen mahdolliseksi. Geneerinen koodi kuvaa algoritmin yleisellä tasolla, jolloin sitä voi käyttää monessa eri tilanteessa täydentämällä sitä sopivilla parametreilla, jotka kertovat miten sitä sovelletaan kyseisessä tilanteessa. On kuitenkin aiheellista kysyä, rikotaanko tälläisella polymorfismilla Wallin puolestapuhumaa “eri tavalla toimivien asioiden tulisi myös näyttää erilaisilta” -periaatetta vastaan (Wirth–Hoare 1966, 429).

Phase on syntaktisesti hyvin ortogonaalinen, sillä se ei aseta mitään rajoituksia sen suhteen, mitä sanasta voi käyttää ja missä. Semanttisesti kuitenkin rajoituksia käytännössä on, sillä se, mitä olioita mikäkin sana tai proseduuri osaa käsitellä, riippuu tuosta sanasta tai proseduurista itsestään. Itse kielen ydin ei kuitenkaan aseta juuri minkäänlaisia rajoituksia etukäteen tuon suhteen, minkä vuoksi pitäisin kieltä tälläkin tasolla suhteellisen vähän rajoittavana ja siksi ortogonaalisena.

Phase on verrattavissa pinokoneen symboliseen konekieleen ja sen vuoksi se ei sisällä korkeamman tason rakenteellisia ohjausrakenteita, kuten if-lausetta tai for-silmukkaa.

Tosin nuo kaikki ovat helposti lisättävissä kieleen makroja käyttämällä, joten tämän asian suhteen kielen luettavuus riippuu käyttäjästä ja siitä, haluaako käyttäjä lisätä nuo kieleen.

Phase ei tarjoa lähtökohtaisesti spesifisiä tietotyyppejä, vaan käyttäjän täytyy rakentaa ne itse kielen alkeistietotyyppien pohjalta. Phasen omat tietotyypit kuitenkin ovat riittävän spesifisiä tehdääkseen tuollaisen rakentamisen mahdolliseksi enkä näe mitään estettä sille, että käyttäjä ei voisi itse lisätä kielen enemmän turvallisuutta ja tarkistuksia, jos kokee niitä tarvitsevansa. Phasen tuki uusien abstraktioiden rakentamiselle tulee olioiden käytöstä ja sen tulisi olla suhteellisen vaivatonta.

Phasen ilmaisevuus on jossain assembly-kielten ja oliokielten välimaastossa johtuen siitä, että vaikka Phase on oliokieli se ei kuitenkaan tarjoa lähtökohtaisesti assembly-kieltä enempää ohjausrakenteiden osalta. Monimutkaisempia ohjausrakenteita voi ohjelmoija kuitenkin itse rakentaa, jos haluaa, ja niiden rakentamisen suhteen ei ole mitään rajoituksia toisin kuin yleisimmissä kielissä, joissa käyttäjä ei voi vapaasti määritellä uusia abstraktioita ohjausrakenteiden suhteen.

Phase ei tee olioiden tyyppien tarkistusta, sillä olioilla ei ole muuttumattomia tyyppejä sinänsä. Sen sijaan olioiden tyyppi päätellään siitä, millä tavalla ja mihin muihin olioihin ne viittaavat ominaisuuspuussaan. Ominaisuuspuut ovat tavallisia linkitettyjä listoja, joten niitä voi vapaasti muokata. Se, kuinka vahvasti tyypitetty kieli Phase on, riippuu käyttäjästä ja siitä, kuinka paljon hän tyyppitarkistuksia ohjelmiinsa laittaa.

Phase ei käsittele virheitä. Se osaa vain kertoa virheestä ja kaatua. Olisi mahdollista kuitenkin suhteellisen vähällä vaivalla muuttaa kielen käytöstä niin, että virheen sattuessa kutsuttaisiin virheenkäsittelijäoperaattoria, joka voisi sitten matkia yleisten kielten poikkeuskäsittelyä.

Phase on tulkittu kieli, jonka vuoksi se ei ole kovin nopea. Se on myös matalan tason kieli, jonka vuoksi voi olettaa, että ohjelmiston kehittäminen sillä vie enemmän aikaa kuin korkeamman tason kielillä. Tämän vuoksi on ihan perusteltua sanoa, että alan harjoittajien yleisen mielipiteen näkökulmasta Phase on yhdistelmä matalan ja korkean tason kielien huonoimmista puolista. Toisaalta juuri tuon vuoksi se on ohjelmoitavampi, joka oli kielen päätavoite.

5 TULOKSET JA JOHTOPÄÄTÖKSET

Yksinkertaisen laajennettavan ohjelmointikielen luominen onnistui ja sen lopullinen muoto vastaa sille asetettuja vaatimuksia ja tavoitteita riittävän hyvin. Se on hyvin yksinkertainen. Ehkä se on liiankin yksinkertainen. Tämän vuoksi se asettaa hyvin vähän rajoituksia käyttäjälleen ja nuo vähätkin rajoitukset ovat todennäköisesti vähän vaivaa näkemällä kierrettävissä uudelleenohjelmoimalla eli laajentamalla kieltä ohjelman ajon aikana. Jos kielen muuttaminen vaatisi kielen itsensä uudelleenkääntämistä ja kielen itsensä ulkopuolisia työkaluja, niin sitä ei voisi laajennettavaksi sanoakaan.

Phase ei kielenä ole mitenkään uusi tai vallankumouksellinen, sillä se vain sotkee vanhoja vähemmän tuttuja ajatuksia vähemmän tunnetuista ohjelmointikielistä yhteen tavalla, jota tietääkseni kukaan ei kuitenkaan ole vielä julkisesti ilmoittanut kokeilleensa. Kieli näyttää mielenkiintoiselta lähtökohdalta matalamman tason ohjelmointikielten ohjelmoitavuuden tutkimiseen antaen siihen uuden erilaisen dynaamisemman näkökulman verrattuna yleisiin ratkaisuihin, jotka eivät anna ohjelmoijalle yhtä vapaita käsiä sen suhteen, millaisia abstraktioita hän voi kieleen rakentaa. Kielelle sopivin käyttötapa saattaisi ollakin sen laajentaminen ohjelmoitavaksi meta-assembleriksi. Meta-assembleri on siis assembleri, jolle annetaan syötteenä sekä käännettävä ohjelma että käännöksen kohteena olevan alustan tiedot käskykanta mukaanlukien niin, että sillä voidaan kääntää ohjelmia eri alustoille.

Phase on hyvin erilainen suosituimpiin ja laajimmin käytössä oleviin valtavirran kieliin verrattuna. Hankalin puoli uuden kielen oppimisessa yleensä on sen takana olevan ajattelutavan ymmärtäminen, jota kielen tehokkaalla tavalla käyttäminen vaatii. Phase oli allekirjoittaneellekin hankala sisäistää juuri sen takia, että sen antaman ilmaisunvapauden ja yksinkertaisuuden takia oli hankala saada otetta siitä, miten sillä pitäisi ohjelmia kirjoittaa. Yleensä tuo on ohjelmointikielen vaatiman ajattelutavan sisäistäminen on monivuotinen prosessi ja siksi todennäköisesti kestää vielä vuosia ennen kuin pystyn itsekään ymmärtämään, mihin kaikkeen Phase oikeastaan sopii työvälineeksi. On siis aiheellista olettaa, että tavallista oudomman filosofiansa vuoksi Phase yksinkertaisuudestaan huolimatta on hankala ymmärtää niille, jotka ovat tottuneet tavallisempiin kieliin, joilla ohjelmoitaessa ei tarvitse yhtä laajasti miettiä ohjelman suorittamisen prosessin eri vaiheita.

Työn tekemisen aikana tuli mieleen monenlaisia jatkokehitysideoita, joista osa olisi välttämättömiä ennen kuin kielen voisi ottaa vakavassa mielessä käyttöön työvälineenä. Phase on riittävän yksinkertainen siinä suhteessa, että sen toteuttamisen useille eri alustoille pitäisi onnistua hyvin vähällä vaivalla. Se, miten siihen voisi lisätä erilaisia uusia ominaisuuksia kuitenkaan tinkimättä kielen yksinkertaisuudesta, on avoin kysymys eikä välttämättä kovin helposti ratkaistavissa. Ensimmäinen asia, jonka suhteen

kieltä tulisi kehittää, on tuki debuggaamiselle. Mielenkiintoisin jatkokehitysajatus on kielen hiominen niin, että sillä kirjoitetut ohjelmat olisi helppo laskea myös paperilla.

Tätä opinnäytetyötä tehdessäni opin näkemään ohjelmoinnin, ohjelmointikielet, ohjelmointikielten ominaisuudet ja noiden ominaisuuksien toteuttamiseen liittyvät ongelmat realistisemmin ja uusista eri näkökulmista. Ohjelmointikielten rakenteeseen liittyvät valinnat niiden suunnittelun näkökulmasta eivät enää näytä yhtä mustavalkoisilta ja mielivaltaisilta kuin aiemmin, jolloin usein mietitytti, miksi mikäkin asia oli tehty tavallaan. Erityisen antoisa kokemus oli tietojenkäsittelytieteen teoriaan ja sen pohjana toimivaan matematiikkaan perehtyminen, vaikka siihen kovin syvällisesti en ehtinyt paneutuakaan. Tuon teoriatiedon omaksuminen auttoi paremmin hahmottamaan Phasenkin rakenteita ja antoi ideoita sen suhteen, millainen kielen tulisi olla ja miten sitä olisi paras kuvata. Siihen perehtyminen myös sytytti halun tutkia asiaa jatkossa enemmän, koska siitä oli välittömästi käytännön hyötyä asioiden ymmärtämisen helpottumisen kautta ja se näytti niin mielenkiintoiselta.

Projektin aikatauluttaminen kokonaisuudessaan ei onnistunut ja opinkin, kuinka vaikeaa tällaisen tutkimusprojektin aikataulun arviointi voi olla. Parhaiten mieleen jääneet opetukset tämän opinnäytteen tekemisestä onkin olleet, että hyvä teoria on erittäin käytännöllistä ja opettelemisen arvoista eikä ohjelmointikielten suunnittelun ja toteuttamisen ole pakko olla tylsää eikä ylitsepääsemättömän vaikeaa.

LÄHTEET

Abelson, H. – Sussman, G. J. – Sussman, J. 1996. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA.

Aho, A. V. – Sethi, R. – Ullman, J. D. 1986. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Backus, J. 1978. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM* 21:613–641.

Bawden, A. 1999. Quasiquotation in Lisp. Teoksessa *University of Aarhus, Department of Computer Science* 88–99.

Cheatham, T. E., Jr. 1966. The introduction of definitional facilities into higher level programming languages. Teoksessa *Proceedings of the November 7-10, 1966, fall joint computer conference* AFIPS '66 (Fall) 623–637. ACM, New York, NY, USA.

Flatt, M. 2002. Composable and compilable macros: you want it when? Teoksessa *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* 72–83. ACM, New York, NY, USA.

Foderaro, J. 1991. LISP: introduction. *Communications of the ACM* 34(9):27.

Graham, P. 1993. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Hoare, C. A. R. 1981. The emperor's old clothes. *Communications of the ACM* 24(2):75–83.

Hoare, C. A. R. 1989. *Essays in computing science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Kernighan, B. W. – Plauger, P. J. 1976. Software tools. *SIGSOFT Software Engineering Notes* 1(1):15–20.

Kiczales, G. 1992. Towards a New Model of Abstraction in the Engineering of Software.

Minsky, M. 1970. Form and Content in Computer Science (1970 ACM turing lecture). *Journal of the ACM* 17:197–215.

Mooers, C. N. – Deutsch, L. P. 1965. Programming languages for non-numeric processing–1: TRAC, a text handling language. Teoksessa *Proceedings of the 1965 20th national conference* ACM '65 229–246. ACM, New York, NY, USA.

Queinnec, C. – Callaway, K. 2003. *Lisp in Small Pieces*. Cambridge University Press, New York, NY, USA.

Sebesta, R. W. 2008. *Concepts of Programming Languages*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 8. painos.

Steele, G. L., Jr. 1990. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA.

Taivalsaari, A. 1995. Delegation versus concatenation or cloning is inheritance too. *SIGPLAN OOPS Messenger* 6:20–49.

Taivalsaari, A. – Moore, I. – Noble, J. 1999. *Prototype-Based Programming: Concepts, Languages, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Weise, D. – Crew, R. 1993. Programmable syntax macros. Teoksessa *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* PLDI '93 156–165. ACM, New York, NY, USA.

Wirth, N. 1985. From programming language design to computer construction. *Communications of the ACM* 28(2):160–164.

Wirth, N. 2002. Computing science education: the road not taken. *SIGCSE Bulletins* 34(3):1–3.

Wirth, N. – Hoare, C. A. R. 1966. A contribution to the development of ALGOL. *Communications of the ACM* 9(6):413–432.

PYTHON-TOTEUTUS LIITE 1

```
#!/usr/bin/env python

import re,sys

object_ptree = None
symbol_ptree = None
number_ptree = None
tuple_ptree = None
string_ptree = None
stack_ptree = None
code_ptree = None
operator_ptree = None
file_ptree = None

class Input:
    def __init__ (self):
        self.peeked = None
    def next (self):
        if self.peeked == '':
            return ''
        self.peeked = sys.stdin.read(1)
        return self.peeked
    def get (self):
        if self.peeked is None:
            return self.next()
        return self.peeked

class Output:
    def write (self,x):
        print >>sys.stdout, x

class Object:
    def __init__ (self,ptree):
        self.ptree = ptree
    def __str__ (self):
        return '<?>'

class Symbol (Object):
    def __init__ (self,name):
        Object.__init__(self,symbol_ptree)
        self.name = name
    def __str__ (self):
        return self.name
S = Symbol

class Number (Object):
    def __init__ (self,value):
```

```
        Object.__init__(self,number_ptree)
        self.value = value
    def __str__ (self):
        return str(self.value)
N = Number

class Tuple (Object):
    def __init__ (self,*args):
        Object.__init__(self,tuple_ptree)
        self.tuple = list(args)
    def __str__ (self):
        return '( '+' '.join([str(x) for x in self.tuple])+' ) '
T = Tuple

class String (Object):
    def __init__ (self,str):
        Object.__init__(self,string_ptree)
        self.str = str
    def __str__ (self):
        return '" %s"' % (self.str)
R = String

class Stack (Object):
    def  __init__ (self):
        Object.__init__(self,stack_ptree)
        self.stack = []
    def __str__ (self):
        return 'S( '+' '.join([str(x) for x in self.stack])+' )S '
    def push (self,i):
        self.stack = [i]+self.stack
    def pop (self):
        i = self.stack[0]
        self.stack = self.stack[1:]
        return i
A = Stack

class Operator (Object):
    def __init__ (self,fn):
        Object.__init__(self,operator_ptree)
        self.fn = fn
    def __str__ (self):
        return "%d operator " % (id(self))
O = Operator

class Code (Tuple):
    def __init__ (self,l,o,n):
        Object.__init__(self,code_ptree)
        self.tuple = [l,o,n]
    def __str__ (self):
```

```
        if self.tuple[2] is none:
            return ''
        return "^%d %s %s" % (self.tuple[0],self.tuple[1],self.tuple[2])
C = Code

input = Input()
output = Output()

def _intern (str):
    global ST
    i = ST
    while i != none:
        if i.tuple[0].name == str:
            return i.tuple[0]
        i = i.tuple[1]
    i = S(str)
    ST = T(i,ST)
    return i
I = _intern

def _find (sym,pl):
    while True:
        if pl is none: return undefined
        elif len(pl.tuple) == 2:
            x = _find(sym,pl.tuple[0])
            if x != undefined: return x
            pl = pl.tuple[1]
        else: # len(pl.tuple) == 3
            if pl.tuple[0] is sym:
                return pl
            pl = pl.tuple[2]

def empty_word ():
    return C(0,none,none)

def isend (c):
    return re.match('^[ \r\n\t]?$',c) and True or False

def isspace (c):
    return re.match('^[ \r\n\t]$',c) and True or False

def read ():
    global LVL
    while isspace(input.get()): input.next()
    if input.get() == '':
        return PS.push(EOF)
    buf = ''
    while True:
        c = input.get()
```

```
        if isend(c):
            input.next()
            break
        buf += c
        input.next()
    if re.match('^\^\d+$',buf):
        LVL = int(buf[1:],10)
        return read()
    if re.match('^0[0-7]*$',buf):
        return PS.push(N(int(buf,8)))
    if re.match('^0b[01]+$',buf):
        return PS.push(N(int(buf[2:],2)))
    if re.match('^0x[0-9a-fA-F]+$',buf):
        return PS.push(N(int(buf[2:],16)))
    if re.match('^\d+$',buf):
        return PS.push(N(int(buf,10)))
    return PS.push(_intern(buf))

def ReadSentence ():
    while True:
        object = read_object()
        Compile(LVL,t)
        if t is EOF or t is semi:
            break

def get_token ():
    while True:
        if RP.tuple[2] != none: return RP
        read_sentence()

def next_token ():
    global RP
    RP = RP.tuple[2]

def get_next ():
    next_token()
    return get_token()

def _compile (l,o):
    global WP
    WP.tuple[0] = l
    WP.tuple[1] = o
    WP.tuple[2] = empty_word()
    WP = WP.tuple[2]

def is_a (o,p):
    b = _find(p,o.ptree)
    if b is undefined: return False
    return b.tuple[1] is true and True or False
```

```
def get_lvar (s):
    return _find(s,LE)

def get_dvar (s):
    return _find(s,DE)

def get_var (s):
    b = get_lvar(s)
    if b != undefined: return b
    return get_dvar(s)

def abort (msg):
    output.write(msg)
    sys.exit(-1)

def u_resolve ():
    o = PS.pop()
    if not (is_a(o,symbolp)): PS.push(o); return
    b = get_var(o)
    if b is undefined:
        abort("no binding for symbol %s" % (o))
    PS.push(b.tuple[1])

def _eval (o):
    b = _find(eval_sym,o.ptree)
    if b is undefined: abort("eval not defined for %s" % (o))
    b.tuple[1].fn()

def u_eval ():
    _eval(PS.stack[0])

def jsr ():
    global INC, RP
    INC = False
    CS.push(RP)
    RP = PS.pop()

def eval_operator ():
    #print 'eval op(%s)' % (PS.stack[0])
    PS.pop().fn()

def eval_code (self):
    #print 'eval code(%s,%s)' % (PS.stack[0].tuple[0],PS.stack[0].tuple
        [1])
    PS.push(self.tuple[1])
    jsr()

def _do_nothing ():
```

```
    #print 'doing nothing for %s' % (PS.stack[0])
    pass

def _end_of_file ():
    global quit
    quit = True

def main_loop ():
    global INC
    while True:
        t = get_token()
        print
        print PS,"!' PS ;"
        print CS,"!' CS ;"
        print t
        if t.tuple[0] > 0:
            _compile(t.tuple[0]-1,t.tuple[1])
        else:
            PS.push(t.tuple[1])
            u_resolve()
            u_eval()
        if quit: break
        if INC: next_token()
        else: INC = True

# setup global runtime structures

none = S('none')
ST = T(none,none)
true = I('true')
objectp = I('object?')
symbolp = I('symbol?')
tuplep = I('tuple?')
object_ptree = T(objectp,true,none)
tuple_ptree = T(tuplep,true,object_ptree)
symbol_ptree = T(symbolp,true,object_ptree)
object_ptree.ptree = tuple_ptree.ptree = symbol_ptree.ptree = tuple_ptree

i = ST
while i != none:
    i.ptree = tuple_ptree
    i.tuple[0].ptree = symbol_ptree
    i = i.tuple[1]

# bootstrap done

G = T(none,none,none)
def add (pl,k,v):
    pl.tuple[2] = T(k,v,pl.tuple[2])
```

```
undefined = I('?')
false = I('false')
numberp = I('number?')
stringp = I('string?')
stackp = I('stack?')
operatorp = I('operator?')
filep = I('file?')
codep = I('code?')

number_ptree = T(numberp,true,object_ptree)
string_ptree = T(stringp,true,object_ptree)
stack_ptree = T(stackp,true,object_ptree)
operator_ptree = T(operatorp,true,object_ptree)
file_ptree = T(filep,true,object_ptree)
code_ptree = T(codep,true,tuple_ptree)

do_nothing = O(_do_nothing)

eval_sym = I('eval')
add(object_ptree,eval_sym,do_nothing)
add(code_ptree,eval_sym,O(eval_code))
add(operator_ptree,eval_sym,O(eval_operator))

end_of_file = O(_end_of_file)
EOF = I('eof')
add(G,EOF,end_of_file)

semi = I(';')
add(G,semi,do_nothing)

quit = False
INC = True
PS = A()
CS = A()
RP = empty_word()
WP = RP
LE = none
DE = T(G,none)
LVL = 0

# add the meat on the bones here...

def number_add ():
    x = PS.pop()
    y = PS.pop()
    if not is_a(x,numberp): abort('x NaN')
    if not is_a(y,numberp): abort('y NaN')
    PS.push(N(x.value+y.value))
```

```
add(G,I('+'),O(number_add))

def pr ():
    output.write(PS.pop())
add(G,I('pr'),O(pr))

def tret ():
    global INC, RP
    if not (PS.pop() is true): return
    INC = False
    RP = CS.pop()
add(G,I('tret'),O(tret))

def tjsr ():
    global INC, RP
    if not (PS.pop() is true): PS.pop(); return
    INC = False
    CS.push(RP)
    RP = PS.pop()
add(G,I('tjsr'),O(tjsr))

def tjmp ():
    global INC, RP
    if not (PS.pop() is true): PS.pop(); return
    INC = False
    RP = PS.pop()
add(G,I('tjmp'),O(tjmp))

def doublequote ():
    PS.push(get_next().tuple[1])
add(G,I("''"),O(doublequote))

def quote ():
    PS.push(get_next().tuple[1])
    u_resolve()
add(G,I("'"),O(quote))

def _is ():
    PS.push((PS.pop() is PS.pop()) and true or false)
add(G,I('is'),O(_is))

add(G,I('$'),O(u_eval))

if __name__ == '__main__':
    main_loop()
```

Lähdekoodilistaus 24. phase.py