Timo Myyrä

Tietokoneshakki funktionaalisella ohjelmoinnilla

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tietotekniikan koulutusohjelma
Insinöörityö
16.3.2011

Metropolia

Tiivistelmä

Tekijä(t) Otsikko Sivumäärä Aika	Timo Myyrä Tietokoneshakki funktionaalisella ohjelmoinnilla 55 sivua + 1 liite 16.3.2011
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaaja(t)	lehtori Miikka Mäki-Uuro lehtori Vesa Ollikainen

Tämän opinnäytetyön aiheena on tutkia, kuinka tietokoneshakki voidaan toteuttaa funktionaalisella ohjelmoinnilla. Työssä käsitellään miten tietokoneshakit rakentuvat. Siinä tutkitaan pelipuun muodostamista, karsintaa ja hakualgoritmien toimintaa. Työ esittelee myös muutaman tunnetun shakkitietokoneen.

Työssä käytetään funktionaalista ohjelmointia. Työ tutkii hieman ohjelmointityylin pohjana olevaa lambdakalkyyliä ja kuvaa sen jälkeen funktionaalisen ohjelmoinnin ominaisuuksia. Näiden tietojen pohjalta on työssä kehitelty tietokoneshakki. Se on toteutettu Clojure-ohjelmointikielellä, joka perustuu vanhimpaan funktionaaliseen ohjelmointikieleen. Työ kuvaa, kuinka Clojurella voidaan esittää tietokoneshakki käyttäen funktionaalista ohjelmointitapaa.

Työn tuloksena toteutettu Tursas-shakkimoottori pelaa riittävän hyvin shakkia, mutta siihen jäi työn valmistumisen jälkeen vielä parantamisen varaa. Memoisaation ja laiskojen sekvenssien tehokkaalla käytöllä se saataisiin toimimaan tehokkaammin. Työssä huomattiin, kuinka hyvin funktionaalinen ohjelmointi tukee dynaamista ohjelmakehitystä. Siinä voidaan ohjelmoinnin abstraktiotaso nostaa korkeammalle, jolloin ei tarvitse kiinnittää paljoa huomiota toteutuksen yksityiskohtiin.

Avainsanat	shakki, clojure, lisp, funktionaalinen ohjelmointi

Metropolia

Abstract

Author(s) Title Number of Pages Date	Timo Myyrä Computer Chess in Functional Programming 55 pages + 1 appendix 16 March 2011
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer Vesa Ollikainen, Senior Lecturer

The purpose of this thesis was to study how computer chess can be implemented using functional programming. The thesis goes through how computer chess programs are constructed, it shows how chess game trees are created and pruned and it also shows a few common search algorithms. The work also covers the introduction to the most famous chess computers.

The study applied functional programming. It presents the basics of Lambda Calculus which is the foundation for functional programming languages. It then shows the most common characteristics of functional programming. A computer chess implementation was developed based on this knowledge. It was done using Clojure programming language which is based on the oldest functional programming language. Furthermore, the work describes how one can implement computer chess with Clojure by using functional programming style.

The Tursas chess engine which was developed as result of the thesis plays reasonable well game of chess but it leaves room for improvement. The engine could be made more efficient by using memoization and lazy sequences. It was also noted how well functional programming supports dynamic software development during the making of Tursas. The abstraction level of the programming could be made higher and as a result, one would not need to pay attention to the details of implementation.

Keywords	chess, clojure, lisp, functional programming

Sisällys

1 Johdanto

Opinnäytetyön tavoitteena on tutustua tietokoneshakin toteutukseen funktionaalisella ohjelmoinnilla. Funktionaalinen ohjelmointi eroaa monella tavalla yleisesti opetetusta olio-ohjelmoinnista. Olio-ohjelmoinnissa tieto kapseloidaan olioiden sisään. Siinä oliot kommunikoivat toistensa kanssa, ja toiminnallisuutta voidaan periyttää aliluokkiin. Funktionaalisessa ohjelmoinnissa funktiot ovat ensiarvoisia tietojäseniä. Siinä ohjelmat nähdään pienten funktioiden koosteena.

Opinnäytetyössä toteutettavaksi projektiksi valitsin tietokoneshakin, sillä se on kaikille tuttu, ja näin projekti on helppo määrittää. Se on myös riittävän monimutkainen projektikohde, jottei sen toteutus ole liian helppo. Funktionaalista ohjelmointikielistä olisi projektiin voinut valita lukuisista vaihtoehdoista. Valitsin ohjelmointikieleksi Clojuren. Se kuuluu Lisp-ohjelmointikieliin, jotka ovat kiehtoneet minua. Lisp-kielet ovat tulleet tunnetuksi tekoälyjen kehityksessä. Shakki oli ensimmäisiä tekoälyn tutkimuskohteita, joten niiden historia on kytköksissä toisiinsa.

Opinnäytetyön olen jakanut kolmeen osaan eri aihealueiden mukaan. Raportin ensimmäisessä osassa perehdytään tietokoneshakin taustoihin. Tutkin hieman, millainen historia tietokoneshakilla on, ja tutustun nykyaikaisiin tietokoneshakkiohjelmiin. Lopuksi esittelen shakkitekoälyissä yleisimmin käytettyjä algoritmeja ja tietorakenteita.

Tietokoneshakin taustojen jälkeen työssä tutustutaan hieman funktionaaliseen ohjelmointiin. Työssä käydään hieman funktionaalisen ohjelmoinnin matemaattista taustaa lambdakalkyylin perusteiden muodossa. Esittelen myös funktionaalisten ohjelmointikielten ominaisuuksia ja siirryn lopuksi esittelemään ohjelmointityössä käytetyn Clojure-ohjelmointikielen taustaa.

Opinnäytetyön viimeinen osa käy läpi työssä toteutettua Tursas-tietokoneshakkia. Sen ei ole tarkoitus tarjota kaikkea mahdollista toiminnallisuutta. Tarkoituksena on tutustua, kuinka tietokoneshakki ylipäätään rakentuu ja kehitellä se funktionaalisen ohjelmointitavan mukaisesti.

2 Tietokoneshakki

2.1 Historia

Shakilla on pitkät perinteet tietokonetekoälyn kehityksessä. Se oli ensimmäisiä kohteita, johon tekoälytutkijat perehtyivät. Alan Turing esitti ensimmäisen tietokoneella pelattavan shakin jo vuonna 1949 [1]. Hänellä ei tuolloin ollut käytössään tietokonetta, mutta hän kuvasi shakkitekoälyn kulun paperilla. Hän testasi ohjelmaansa pelaten shakkia ystäväänsä vastaan. Turing itse simuloi tietokonetekoälyn toimintaa.

Claude Shannon antoi tarkemman kuvauksen tietokoneshakin luomisesta muutamaa vuotta myöhemmin [2]. Hän kuvasi, kuinka tietokoneshakin tekoälyt voidaan jakaa kahteen eri kategoriaan, tyypin A ja B tekoälyihin. Tyypin A shakkitekoälyt laskevat kaikki mahdolliset siirrot ja niiden seuraukset ja valitsevat näin saaduista siirroista kaikkein parhaimman. Tyypin B tekoälyt tarkastelevat vain valikoidusti tiettyjen pelinappuloiden siirtoja hyvien siirtojen varalta. Tyypin B tekoälyt vastaavat, kuinka ihmispelaaja pelaa shakkia. Kokenut ihmispelaaja näkee pelitilanteista suoraan, mitkä pelinappuloiden siirrot vaikuttavat kaikkein eniten pelitilanteen kehittymiseen ja eivätkä he edes harkitse muita siirtoja. Tietokoneiden kannalta tämä on ongelmallista. Tietokoneen tulisi saada tietää, mitä siirtoja sen kannattaa harkita. Jos tietokone valitsee huonot siirrot, joita se lähtee tarkastelemaan, voi se jättää oikeasti merkittävät siirrot kokonaan tarkastelematta. Näin se voi tarjota vastustajalle suuren taktisen etulyöntiaseman. Tietokoneshakin alkuaikoina tietokoneiden laskentakapasiteetti ei riittänyt useiden siirtojen läpikäyntiin, joten ne pyrkivät noudattelemaan tyypin B tekoälyjen valikoitua siirtojen tarkastelua. Tietokoneiden laskentateho alkoi olla 70-luvulla riittävän suuri, että tyypin A tekoälyt alkoivat menestyä shakkiturnauksissa paremmin kuin B-tyypin. Nykyään kaikki huipputason shakkitekoälyt toimivat tyypin A mukaisesti, mutta sisältävät paljon tyypin B elementtejä tehostaen niiden toimintaa.

Shannonin teoksessa esiteltiin kuinka minimax-algoritmilla [3, s. 190] voi suorittaa pelipuuhun kohdistuvia siirtojen hakuja. Pelipuu on puurakenne, jossa yksittäistä solmua kuvaa pelitilanne. Juurisolmuna toimii nykyinen pelitilanne. Sen alla sijaitsevat yhdellä siirrolla saatavat pelitilanteet. Minimax-algoritmi vaatii toimiakseen funktion

pelitilanteen heuristiseen arviointiin. Heuristisen arviointi-funktion tarkoituksena on tarkastella, kuinka hyvän peliasemaan pelaaja voi saavuttaa tekemällä kyseisen siirron. Shakin luonne ja säännöt tekevät täydellisen siirron muodostamisesta mahdottoman. Yhden shakkipelin siirrosta aiheutuva haarautuminen on keskimäärin 35, ja yhdessä pelissä on tyypillisesti 50 siirtoa. Näin shakin pelipuun kooksi saadaan 35^{100} tai 10^{154} [3, s. 162]. Nykyään shakkitekoälyt pyrkivät tehokkaasti kuvaamaan shakkipelin tilanteen ja muodostamaan pelin sallitut siirrot mahdollisimman nopeasti. Siirtojen arviointiin tekoälyissä ei käytetä paljon resursseja: on huomattu, että riittävän hyvällä arviointi-funktiolla pärjää hyvin, jos tekoäly pystyy tarkastelemaan siirtojen seurauksia riittävän syvälle pelipuuhun.

Minimax-algoritmi on toimiva, mutta sen suoritus on hidas. Parantaakseen algoritmin toimintaa useat tutkijat päätyivät itsenäisesti alfabeta-algoritmin määrittelyyn 50-luvulla. Alfabeta jättää osan pelipuusta kokonaan tarkastelun ulkopuolelle, jos se ei vaikuta lopputulokseen. Yksi ensimmäisten joukossa olevista tutkijoista oli MIT-tekoälylaitoksella työskentelevä, tietokoneshakin pioneereihin lukeutuva John McCarthy [3, s. 191]. McCarthy ehdotti oppilaalleen Alan Kotokille alfabeta-algoritmin käyttämistä tämän suunnittelemassa shakkiohjelmassa. Ensimmäinen tietokoneiden välinen shakkiottelu oli juuri Kotok-McCarthy-shakkiohjelman ja moskovalaisen ITEP-ohjelman välillä [3, s. 192]. McCarthy myös loi vuonna 1958 tekoälyohjelmointikielenä tunnetuksi tulleen Lisp-ohjelmointikielen [4], johon opinnäytetyössä käytetty Clojure-ohjelmointikielikin perustuu.

Vuonna 1968 kansainvälinen shakkimestari David Levy pelasi leikkimielisen ottelun John McCartyä vastaan Edinburghissa, ja hän voitti McCarthyn pistein 4-0 [5]. McCarthy ennusti tuolloin, että tietokoneet päihittävät ihmispelaajan kymmenen vuoden kuluessa. Hän löi tästä vetoa Levyn kanssa. Levy voitti vedon 1978, kun hän voitti Chess 4.7 shakkiohjelman. Vuonna 1988 hän hävisi ottelun Deep Thought -shakkitietokoneelle. Deep Thought oli IBM:n kehittelemä shakkitietokone, joka edelsi kuuluisaa Deep Blueta [3, s. 185].

2.2 Modernit shakkitietokoneet

Vuosi 1997 oli merkittävä shakkitekoälyjen suhteen. Tuolloin IBM:n Deep Blue -shakkitietokone voitti silloisen shakin maailmanmestarin, Garry Kasparovin. Tämä oli ensimmäinen kerta, kun tietokoneshakki voitti hallitsevan maailmanmestarin [3, s. 192]. Deep Blue oli rinnakkainajettava shakkiohjelma. Siinä oli useita prosessoria pelkästään alfabeta-haun suoritukseen. Erikoisuutena se käytti mukautettuja shakkiprosessoreja, joiden tarkoituksena oli suorittaa siirtojen luontia, järjestelyä ja arviointia pelipuun häntäpäälle. Deep Blue käsitteli noin 30 miljardia pelitilannetta per siirto. Se tarkasteli siirtoja aina 14 siirron syvyyteen, mutta lisäksi se tarkasteli pelipuuta paikoin jopa 40 siirron syvyyteen asti niin sanotun yksittäislaajennuksen avulla.

Hydra-shakkitietokonetta voidaan pitää Deep Bluen jälkeläisenä. Se toimii 64 1GHz prosessorin klusterissa ja sisältää Deep Bluen tapaan mukautettua mikrosiruja, jotka on optimoitu shakin tarpeisiin. Hydra suorittaa noin 200 miljoonaa pelitilanteen arviointia per sekunti, mikä vastaa suurin piirtein Deep Bluen tasoa. Hydra eroaa Deep Bluesta siten, että se saavuttaa 18 siirron syvyyden käyttämällä tyhjän siirron heuristiikkaa ja esikarsintaa.

Rybkaa pidetään tämän hetken parhaimpana shakkiohjelmana. Rybka poikkeaa Hydrasta siinä, ettei sitä ajeta klusterissa. Se toimii normaalin 8-ytimen Xeon-prosessorin päällä. Rybkan rakenteesta ei tunneta paljon, sillä ohjelman lähdekoodia ei ole saatavilla. Rybkan etuna uskotaan olevan erityisen tarkka arviointifunktio. Sitä on kehittänyt Rybkan kehittäjä ja ainakin kolme shakin suurmestaria.

2.3 Esitysmuodot

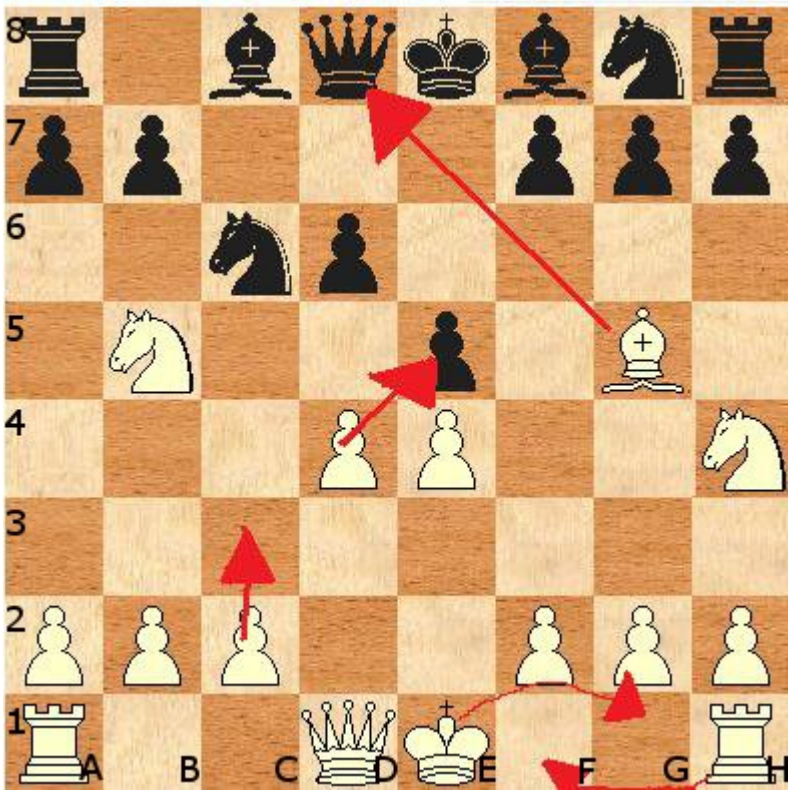
Shakin pelitilanteiden ja siirtojen esitykseen on kehitetty erilaisia esitysmuotoja. Karkeasti ottaen shakin esitysmuodot voidaan jakaa kahteen ryhmään, ihmisten ja tietokoneiden käyttöön soveltuviin esitysmuotoihin.

Yksittäisten siirtojen kuvaamiseen on käytössä kaksi tunnettua tapaa. Algebrallinen esitys (Standard Algebraic Notation) on nykyinen standardi shakkiturnausten

esitykseen. Sitä käytetään laajalti myös alan kirjallisuudessa. Esitysmuodossa siirrot kuvataan antamalla jokaisella pelinappulalle tunnusomainen kirjain yhdistettynä kohderuudun koordinaattiin. Sotilaan tapauksessa kirjainta ei tarvitse esittää. Lyönneissä kirjaimen ja ruudun koordinaatin väliin merkitään x kuvaamaan lyöntiä. Shakin aiheuttavat siirrot merkitään +-merkillä ja matin aiheuttaviin siirtoihin lisätään kohderuudun perään #-merkki. Linnoittaminen merkitään merkkijonolla 0-0-0, jos linnoittaminen tapahtuu kuningattaren puolelle, ja 0-0 kuninkaan puolelle. Sotilaan korotukset merkitään kohderuudun jälkeen annettavalla =-merkillä ja lisäämällä pelinappulan merkki. Ohestalyöntiä merkitään kuten normaalia lyöntiä. Jos esitystapa jättää epäselväksi, mikä pelinappula liikkeen suorittaa voidaan siirron kuvausta tarkentaa seuraavasti.

1. Jos liikkuvan pelinappulan linja yksilöi sen, voidaan linjaa vastaava kirjain lisätä heti liikkuvan pelinappulan merkin perään.
2. Jos vielä ei saada selvyyttä, tarkastellaan, yksilöikö rivi siirron tekevän pelinappulan, ja lisätään se vastaavasti siirtoon.
3. Jos vieläkään ei saada selville siirtyvää pelinappulaa, lisätään siirtoon kaksikirjaiminen kohderuudun koordinaatti.

On huomioitava, että liikkeen selkeyttämiseksi yritetään yllä olevia askeleita annetussa järjestyksessä. Esitystapaan voidaan myös lisätä siirtojen analysointia, kuten "!!" erityisen hyvän siirron merkiksi ja vastaavasti "??" erityisen huonon siirron merkiksi [6].



Kuvio 1. Shakkisiirtoja

Esimerkkisiirtoja vasemmalta oikealle: c3, dxe5, 0-0, Bxd8

Tietokoneelle yksinkertaisempi tapa ilmaista siirtoja on täydellinen koordinaattiesitys, jossa ei mainita liikkuvaa pelinappulaa. Siirrot kuvataan antamalla lähde- ja kohderuutu. Jos kyseessä on korotus, annetaan uuden pelinappulan kirjain siirron lopussa. Esimerkiksi ylläkuvatut siirrot olisivat koordinaattimuodossa: c2c3, d4e5, e1g1, g5d8.

Forsyth–Edwards Notation -esitys (FEN) on tietokoneille tarkoitettu, shakin tilaa kuvaava esitysmuoto [7]. Se koostuu yhdestä merkkijonosta, jossa on kuusi välilyönnillä erotettua kenttää. Kentät kuvaavat järjestyksessä alusta alkaen pelilaudan nappuloita, vuorossa olevan pelaaja, linnoittamismahdollisuutta, ohestalyönnin ruutua, puolisiirtojen ja täysien siirtojen määrä.

Esimerkiksi shakin alkutilanne on FEN-esityksellä:

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

Pelinappuloita kuvataan yksittäisellä kirjaimella. Valkean pelaajan nappulat kuvataan isoilla ja mustat pienillä kirjaimilla. Pelilaudan rivit erotellaan /-merkeillä. Jos rivillä on tyhjiä ruutuja, niiden määrä osoitetaan lukuarvolla. Vuorossa olevan pelaajan väri ilmoitetaan pienellä w- tai b-kirjaimella, jotka tulevat englannin kielen sanoista white (valkoinen) ja black (musta). Linnoittamisen mahdollisuus osoitetaan kirjaimilla. K-kirjain merkitsee mahdollisuutta linnoittaa kuninkaan puolelta ja q-kirjain kuningattaren puolelta. Isot kirjaimet osoittavat valkoisen ja pienet mustan pelaajan linnoittamismahdollisuuden. Jos kumpikaan pelaaja ei voi linnoittaa, kenttää kuvaa yksittäinen viiva (-). Ohestalyönnin mahdollisuus kuvataan joko sotilaan takana olevan ruudun koordinaatilla tai viivalla, jos mahdollisuutta ei ole. Puolisiirtojen määrä kuvaa, kuinka monta siirtoa on kulunut edellisestä sotilaan siirrosta tai pelinappulan lyönnistä. Tällä voidaan seurata 50-siirron säännön toteutusta. Viimeisenä on kaikkien siirtojen määrä, joka alkaa ykkösestä ja kasvaa yhdellä mustan pelaajan tekemän siirron jälkeen.

FEN- tai SAN-esitystavalla ei voi helposti kuvata pelin etenemistä. Sitä varten on kehitelty PGN tai Portable Game Notation [7]. Se on ihmisille sekä tietokoneille suunniteltu pelin etenemistä kuvaava esitystapa. Siinä listataan ensin pelin alkutiedot tageina. Jos pelissä pelataan jotain shakin variaatiota, pelin alkutilanne annetaan FEN-merkkijonona. Alkutietojen jälkeen PGN listaa pelaajien tekemät siirrot SAN-esitystavalla.

Esimerkkipeli PGN-esitystavalla:

[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]
1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7 11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5 Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6 23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5 hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5 35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6 Nf2 42. g4 Bd3 43. Re6 1/2-1/2

2.4 Protokollat

Shakin mallintaminen tietokoneella on yleistä. Tietokoneshakin ei tarvitse aina toteuttaa graafista käyttöliittymää. Protokollan avulla shakkimoottoriksi kutsuttu tietokoneshakki voi kommunikoida shakkikäyttöliittymän kanssa. Näin voidaan esimerkiksi kaksi erilaista shakkimoottoria laittaa pelaamaan toisiaan vastaan. Nykyään käytössä on kaksi tunnettua shakin tiedonsiirtoprotokollaa, Chess Engine Communication Protocol sekä Unified Chess Interface.

Chess Engine Communication Protocol on XBoard-shakkiohjelman tekijöiden kehittelemä ja käyttämä kommunikointiprotokolla [8]. Tim Mann kehitteli sen XBoard-käyttöliittymän ohessa. Sen alkuperäinen tarkoitus oli kommunikoida GNU Chess-shakkiohjelman kanssa, joka käsitteli vain tekstipohjaista syötettä. Protokollan ensimmäinen versio loi GNU Chess -ohjelman päälle oman protokollan. Tämän avulla Xboard-ohjelman graafisen käyttöliittymän tapahtumat pystyttiin kommunikoimaan GNU Chess -shakkiohjelmaan. Protokolla on laajentunut tukemaan normaalin shakin lisäksi erilaisia shakin variaatioita. Se tukee myös kolmea erilaista ajanhallintatapaa, perinteisiä kelloja, kasvavia kelloja ja tarkkoja sekuntia per vuoro ajoituksia. Protokollaa käyttäviä shakkitekoälyjä on arvioilta yli 300, ja yli 30 shakkikäyttöliittymää tukee sitä. Uusin protokollan versio laajensi protokollan tukemaan muistinkäytön ja säikeiden rajaamista.

Unified Chess Interface, UCI, on toinen yleisessä käytössä oleva protokolla. Stefan Meyer-Kahlen kehitteli ja julkaisi protokollan vaihtoehdoksi vanhemmalle Chess Engine Communication Protocol -protokollalle [9]. Hän on myös Shredder-shakkitekoälyn luoja, jossa osa perinteisesti pelimoottorin huoleksi jätetyt osat on siirretty käyttöliittymän puolelle. Merkittävimpänä pelin aloitussiirrot on jätetty käyttöliittymän puolelle. Pelin ensimmäiset siirrot tulevat avaussiirrot sisältävästä taulukosta. Kun peli on saatu näin avattua, alkaa shakkimoottori vasta suunnitella siirtojaan. Shakkimoottori voi myös jättää loppupelin siirrot käyttöliittymälle tai hoitaa ne itse. Shakin loppupelit on ratkaistu, jos pelilaudalla on vähän nappuloita. Kaikki mahdolliset siirrot loppupelistä on laskettu valmiiksi ja sijoitettu taulukkoon. Taulukosta voi lukea parhaan mahdollisen siirron loppupelissä. Jos loppupeli jätetään käyttöliittymän huoleksi, shakkimoottori ei osaa varautua loppupeliin. Esimerkiksi moottori voi lähellä loppupeliä tarkastella siirtoja

tiettyyn syvyyteen asti ja horisonttivaikutuksen vuoksi luulla, että saatu asema on hyvä, vaikka todellisuudessa se johtaisi loppupelissä aina mattiin. Tältä voidaan välttyä, jos shakkimoottori käsittelee loppupelitaulukkoa itse. Näin se voi tarkastella pelipuuta läpikäydessään, onko pelitilanteelle vastine loppupelitaulukosta. Jos on, voi shakkimoottori tarkastaa siirron koko vaikutuksen aina pelin loppuun asti, eikä näin kärsi horisonttivaikutuksesta turhaan. UCI-protokolla ei saanut paljoa kannatusta ennen kuin Fritziä markkinoiva Chessbase-yritys alkoi tukea sitä 2002. Nykyään protokolla tukee yli 100 pelimoottoria.

2.5 Pelilaudan tietorakenteet

Vuosien saatossa on tietokoneshakeissa käytetty monenlaisia tietorakenteita kuvaamaan shakkilautaa. Shakin mallintamiseen käytettävien tietorakenteiden tehokkuus on suoraan verrannollinen niiden monimutkaisuuteen. Helposti toteutettavat ja yksinkertaiset tietorakenteet eivät ole tehokkaita, ja tehokkaat tietorakenteet ovat hankalia toteuttaa ja hahmollistaa. Tietorakenteet perustuvat joko taulukon käyttöön tai bitti-lukuarvoihin talletettuihin tietoihin.

Kaksiulotteinen taulukko on kaikkein yksinkertaisin tietorakenne shakkilaudan esitykseen. Siinä taulukon yksi alkio vastaa yhtä shakkilaudan ruutua. Näin yhdellä 64-alkiosella taulukolla voidaan esittää koko shakkilauta. Jokaiseen alkioon voidaan tallentaa siinä olevan nappulan tiedot. Yksinkertaisimmillaan tämä voi olla lukuarvo. Esimerkiksi valkoisille nappuloille voidaan antaa arvoiksi numerot 1,2,3,4,5,6 merkitsemään sotilasta, tornia, ratsua, lähettiä, kuningatarta ja kuningasta. Mustalle pelaajalle voidaan antaa vastaavat mutta negatiiviset arvot. Tyhjät ruudut voidaan merkitä luvulla 0.

Taulukko 1. 8x8-taulukko

-2	-3	-4	-5	-6	-4	-3	-2
-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	3	4	5	6	4	3	2

Siirtojen muodostaminen 8x8-taulukossa ei ole tehokasta, sillä jokaista siirtoa tehdessä tulee aina tarkastaa, osuuko siirto pelilaudalle, mikä hidastaa sen käyttöä. Lisäksi taulukon käyttöä hidastaa sen sisällön tarkastelu. Kun pelilaudan pelinappulat halutaan listata, tulee kaikki taulukon arvot käydä läpi ainakin kertaalleen. Tämä hidastaa merkittävästi tekoälyn toimintaa, kun se hakee hyvää siirtoa laudalta.

Taulukko 2. 12x12-taulukko

99	99	99	99	99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99	99	99	99	99
99	99	-2	-3	-4	-5	-6	-4	-3	-2	99	99
99	99	-1	-1	-1	-1	-1	-1	-1	-1	99	99
99	99	0	0	0	0	0	0	0	0	99	99
99	99	0	0	0	0	0	0	0	0	99	99
99	99	0	0	0	0	0	0	0	0	99	99
99	99	0	0	0	0	0	0	0	0	99	99
99	99	1	1	1	1	1	1	1	1	99	99
99	99	2	3	4	5	6	4	3	2	99	99
99	99	99	99	99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99	99	99	99	99

Pienenä variaationa 64-alkioista taulukkoa voidaan laajentaa 12x12 taulukoksi. Siinä pelilautaa kuvaavien alkioiden ympärille jätetään kaksi tyhjää alkiota, mikä helpottaa laillisten siirtojen tarkastelua. Pelilaudan ulkopuolelle jääville alkiolle voidaan antaa arvoksi jokin suuri luku, esimerkiksi 99. Siirtoja muodostaessa voidaan tarkastella siirron päätepisteen arvoa. Jos siirto päätyisi ruutuun, jonka arvo on 99, tiedetään, ettei se ole sallittu, sillä se menisi pelilaudan reunojen yli. Pienempää taulukkoa käyttäen olisi kaikki nappuloiden siirtojen indeksiarvot tarkistettava ennen taulukkoon viittaamista. Jos annettu indeksiarvo ei osu taulukolle varattuun tilaan, on seurauksena ohjelman virhetilanne. 12x12-taulukossa nappuloiden siirrot päätyvät aina taulukkoon tai sen ulkoreunaa kiertäville alkioille.

Taulukkopohjaista ratkaisua voidaan parantaa antamalla taulukon kooksi 8x16 alkiota. Tätä kutsutaan 0x88-pelilaudaksi [10]. Taulukon alkuperäinen kehittäjä on tuntematon, mutta Bruce Moreland kertoo kuulleensa siitä Hong Kongin tietokoneshakin maailmanmestaruuskisoissa vuonna 1995. 0x88-taulukossa pelilaudalle varataan tilaa 128-alkioisen taulukon verran. Tämä vastaa sitä, että normaalin 8x8-taulukon oikealle puolelle asennetaan toinen samankokoinen taulukko. Taulukon edut tulevat paremmin ilmi, kun sitä tarkastelee sen alkioiden indeksien arvoja heksamuodossa.

Taulukko 3. 0x88-metodilla toteutettu taulukko

70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F

0x88-taulukosta saatava hyöty tulee sen rakenteesta. Taulukon vasemman puoleisiin alkioihin asetetaan pelinappuloita kuvaavat arvot aivan kuten 8x8-taulukon kanssa. Taulukon rakenne antaa suuren hyödyn siirtojen muodostamisessa. Kun halutaan

selvittää, osuuko annettu indeksi pelilaudalle, saadaan se nopeasti selville yksinkertaisella bittioperaatiota käyttävällä ehdolla.

```
(index & 0x88) == 0
```

Yllä oleva lauseke antaa tulokseksi nollan vain, kun indeksin arvo osoittaa taulukon pelialuetta vastaavaa aluetta. Testi on nopea suorittaa. Näin ei tule riskiä, että siirtoja muodostaessa viitattaisiin taulukon ulkopuolelle, mikä voisi johtaa ohjelman virhetilanteeseen. Pelilaudan indeksin sijainti pelilaudan riveihin ja sarakkeisiin nähden saadaan myös selville bittioperaatioilla. Bittioperaatioilla suoritettavien testien ensisijainen hyöty on niiden nopea suoritus. Tämän lisäksi taulukon indekseillä on selkeät suhteet toisiinsa. Esimerkiksi siirryttäessä ylöspäin pelilaudalla indeksin arvo kasvaa 16:sta ja vasemmalle siirryttäessä se pienenee yhdellä.

Bittilaudat eroavat siten, että taulukko ei enää kuvaa pelilautaa. Bittilaudoissa luodaan taulukko, johon tallennetaan 64-bittisiä lukuarvoja. Siitä voidaan käyttää myös termiä bittitaulukko tai bittivektori. Bittilauta perustuu siihen, että shakkilauta koostuu 64 ruudusta. Näin pelilaudan yksittäisen ruudun tilannetta voidaan kuvata yhdellä bitillä. Bittilautojen tehokas käyttö vaatii 64-bittisen prosessorin, jotta 64-bittisiä lukuarvoja voidaan käyttää tehokkaasti. Koko pelilaudan esitykseen tarvitaan useampi 64-bittinen lukuarvo, sillä yksittäinen bitti voi esittää vain arvoa 1 tai 0. Yksittäinen 64-bittinen lukuarvo kertoo tietyn tyyppisten pelinappuloiden sijainnit laudalla. Yksi lukuarvo voi kertoa valkoisten lähettien sijainnit laudalla, toinen valkoiset sotilaat ja niin edelleen. Kun näin kaikki kahdentoista eri pelinappulatyypin sijainnit sisältävät lukuarvot yhdistetään loogisella JA-operaatiolla, saadaan koko pelilaudan sisältö selville. Näin myös eri pelinappuloiden sijainnit on nopea selvittää kohdistamalla bitti-operaatioita vain haluttuihin lukuarvoihin.

Bittilautamenetelmää on käytetty jo 50-luvulta lähtien [11]. Sitä käytettiin alun perin tammessa. Shakkiin bittilaudat tulivat 70-alkupuolella. Bittilautoja alettiin käyttää kahdessa tunnetussa shakkiohjelmassa, eri puolella maailmaa, samoihin aikoihin. Chess-ohjelma Amerikassa ja Kaissa-ohjelma silloisessa Neuvostoliitosta käytti bittilautaa pelin esitykseen. Pyörivät bittilaudat on pieni optimointikeino bittilautoja käyttävän shakkimoottorin siirtojen muodostukseen. Niissä ”liukumalla” liikkuvien pelinappuloiden, kuten kuningattaren ja lähetin, pelinappuloiden bittilaudoista pidetään kopioita. Kopioiden bittejä pyöritetään jonkin tietyn asteluvun verran, jotta ne

vastaisivat paremmin pelinappuloiden liikkeitä. Näin lukuarvon bitit saadaan asettumaan vierekkäin, mikä nopeuttaa siirtojen muodostamista kyseisille pelinappuloille. Maagiset bittilaudat ovat uusin optimointikeino bittilautoihin. Ne käyttävät kerto- ja bittisiirto-operaatioita niin sanottujen täydellisten hajautustaulujen luontiin. Nykyisin tietokoneet osaavat suorittaa nämä operaatiot todella nopeasti, ja prosessoreiden välimuistit ovat kasvaneet niin isoiksi, jotta tehdyt operaatiot pystytään tallentamaan välimuisteihin niiden suorituksen ajaksi.

On huomioitavaa, että pelkästään pelilaudan tilanteen tallentaminen ei riitä. Vaikka kaikki ruudut ovat esitettyinä, shakissa pitää myös ylläpitää tietoa linnoittamisen, ohestalyönnin mahdollisuudesta ja shakkitilanteista.

2.6 Siirtojen muodostus

Siirtojen muodostamisen tavoitteena on muodostaa kaikki lailliset pelinappuloiden siirrot nykyisestä pelitilanteesta. Siirtojen muodostus jaetaan laillisiin ja pseudolaillisiin siirtoihin. Lailliset siirrot on tarkastettu mahdollisten shakkitilanteiden varalta, mutta pseudolaillisia ei tarkasteta. Siirtojen muodostaminen voidaan jakaa kolmeen eri ryhmään: valikoivaan, kasvavaan ja täydelliseen muodostukseen.

Valikoiva muodostus, tai esikarsinta, on Shannonin määrittelemä tyypin B tekoäly. Siinä valitaan muutama kiinnostava siirto ja tarkastellaan näiden seuraamuksia eikä välitetä muista vaihtoehdoista. Tämä vastaa melko tarkkaan, kuinka ihmispelaajat valitsevat siirtonsa shakissa. Metodi oli suosittu 1970-luvulle asti, kunnes tietokoneiden laskentatehot olivat kasvaneet niin suuriksi, että kaikkien siirtojen muodostaminen oli varmempi keino. Valikoivan muodostuksen ongelmana on oikeiden pelinappuloiden valinta. Shakkimoottori voi tehdä todella huonoja siirtoja valitsemalla tarkastelun kohteeksi väärät pelinappulat.

Jäljelle jäävät tekniikat ovat samanlaisia, ja ne eroavat lähinnä toteuttamistavaltaan. Täydellisessä muodostuksessa kaikki siirrot muodostetaan suoraan ja näistä valitaan haluttu. Se soveltuu peleihin, joissa siirtojen muodostus on helppoa, ja jossa sama pelitilanne voi toistua usein. Tekniikka käyttää yleisesti siirtovaihtotaulukoita, johon eri

pelitilanteet tallennetaan. Koska pelitilanteet toistuvat useasti, vältytään taulukoita käyttämällä laskemasta siirtoja moneen kertaan.

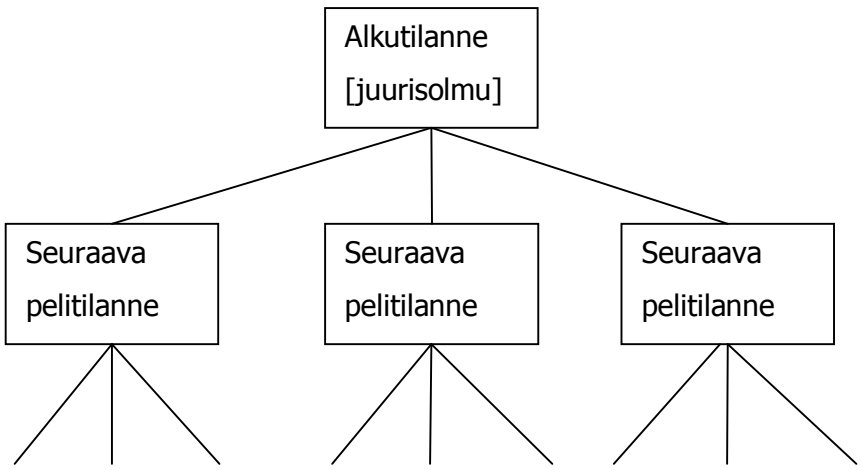
Shakin siirtojen muodostaminen on monimutkaista, joten kasvava muodostus sopii siihen paremmin. Siinä aletaan jostain tietystä pelitilanteesta alkaen muodostaa laillisia siirtoja seuraavan kaavan mukaan:

- Haetaan kaikki lailliset siirrot annetusta asemasta.
- Järjestetään löydetyt siirrot jonkin kaavan mukaan nopeuttaen näin hyvien siirtojen hakua.
- Hakien lisää siirtoja kaikista jo löydetyistä siirroista, kunnes kaikki vaihtoehdot on käyty läpi tai annettu hakusyvyys on saavutettu.

Kasvavan haun tarkoitus on hakea hyvää siirtoa kokeilemalla muutamaa. Näitä siirtoja tarkastellaan, ja jos vielä ei löydy tarpeeksi hyvää siirtoa, luodaan niitä lisää ja jatketaan, kunnes hyvä siirto on löytynyt. Tavoitteena on, että hyvä siirto löytyy mahdollisimman nopeasti, jolloin vältytään luomasta kaikkia siirtoja.

2.7 Pelipuun käsittely

Shakin pelipuun käsittely on shakkitekoälyn olennaisin toiminto. Shakin pelipuulla tarkoitetaan puurakennetta, jossa nykyinen pelitilanne on juurisolmuna. Tämän alla on kaikki pelitilanteesta mahdollisesti seuraavat pelitilanteet, ja näiden alla aina niistä seuraavat pelitilanteet.



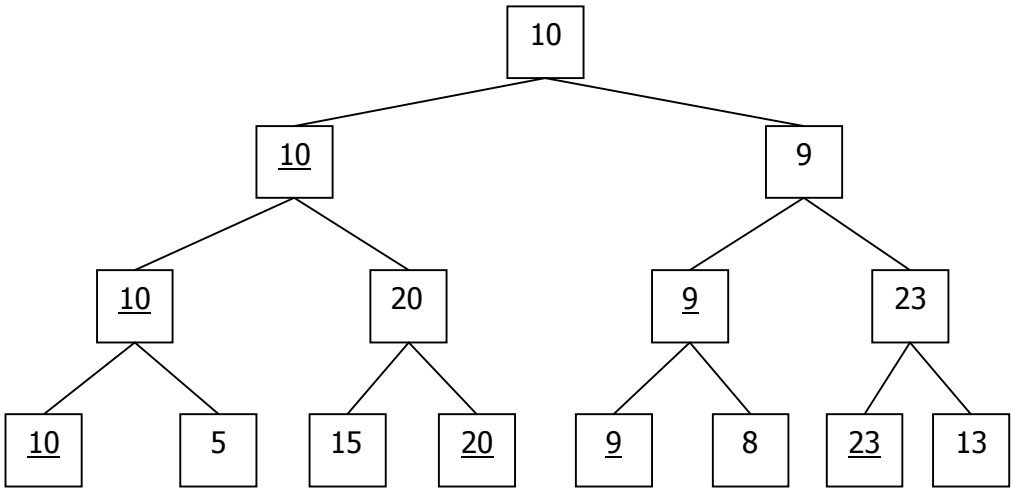
Kuvio 2. Pelipuu

Oikea shakin pelipuu on huomattavasti laajemmin haarautuva. Yllä olevan kuvan tarkoituksena on visualisoida puun rakennetta. Esimerkiksi shakin alkutilanteesta valkoisella pelaajalla on 20 laillista siirtoa. Näin puun juurisolmun alla olisi 20 solmua. Jokaisen tällaisen alisolmun alla olisi jälleen 20 uutta solmua, sillä mustalla on vastaavasti 20 siirtoa omasta alkuasetelmastaan.

Shakin säännöt määräävät, mitkä siirrot on sallittu missäkin tilanteessa. Tekoälyn on pystyttävä valitsemaan näistä vaihtoehdoista oma siirtonsa. Pelipuun käsittely voidaan jakaa eri osiin, kuten haku-algoritmeihin, pelipuun karsintatapoihin, laajennuksiin sekä puun päätepisteiden arviointiin.

Algoritmit

Shakin siirtojen hakemiseen pelipuusta on kehitetty useita erilaisia algoritmeja. Haun perusalgoritmina voidaan pitää minimax-algoritmia. Kaikki shakkialgoritmit perustuvat minimax-algoritmiin, mutta ne parantavat sen tehokkuutta eri tavoin. Minimax-algoritmi käy läpi kaikki pelitilanteet ja valitsee niistä aina parhaimman lopputuloksen antavan vaihtoehdon. Sen heikkoutena on hidas suoritusaika. Algoritmin hitaus johtuu siitä, että se käy läpi kaikki pelipuun solmut ilman minkäänlaista karsintaa. Päätesolmuista lähtien se alkaa nostaa arvoja ylöspäin puussa vuorotellen joko min-arvoa tai max-arvoa pelaajasta riippuen.

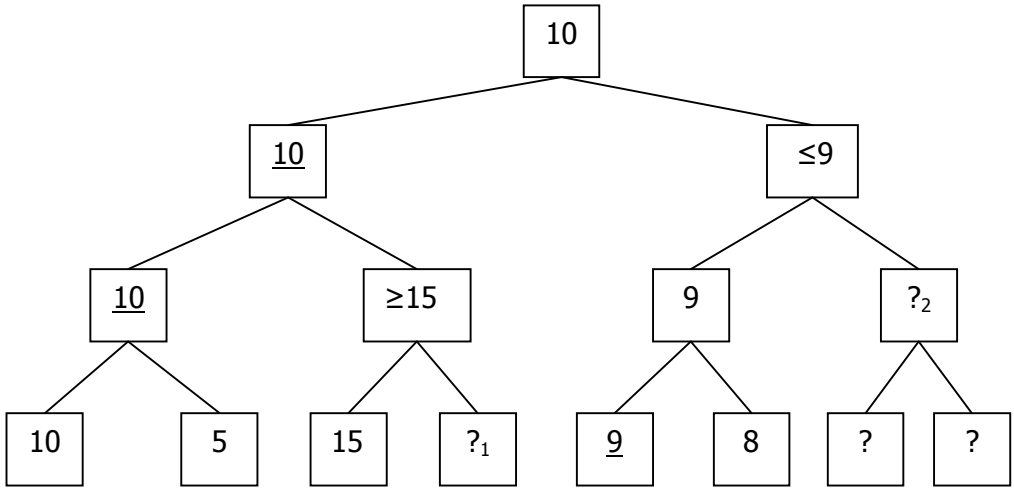


Kuvio 3. Minimax-algoritmin kulku neljän siirron hakusyvyydellä

Kuva on yksinkertaistettu versio minimax-algoritmin toiminnasta pelipuussa. Oikeassa pelipuussa olisi enemmän haarautumisia, mutta tämä sopii esimerkin tarkoitukseen hyvin. Pelipuun alimmat arvot on saatu arviointi-funktion tuloksena. Näistä arvoista valitaan pelaajan kannalta paras vaihtoehto ja nostetaan sen arvo puussa ylöspäin. Valittuja arvoja kuvaan alleviivatuilla luvuilla. Seuraavaksi on vastustajan vuoro valita. Oletetaan vastustajan pelaavan hyvin, joten hän valitsee itselleen parhaimman arvon, joka on vastaavasti pelaajan kannalta pienin arvo ja nostaa sen taas puun haaraa ylöspäin. Tässä tapauksessa vastustaja on valinnut arvot 9 ja 10. Viimeisenä valitaan paras arvo omalta kannalta ja nostetaan se koko puun arvoksi, tässä tapauksessa 10.

Useimmat shakin haussa käytettävät algoritmit toimivat minimax-algoritmin periaatteella, mutta pyrkivät tehostamaan sen toimintaa jättämällä joitain kohtia pelipuusta käsittelemättä. Minimax-algoritmista on olemassa pieni variaatio, negamax. Negamax-algoritmi perustuu siihen, mitä huonompi vastustajan siirto on, sen parempi se on itselle. Etuna minimax-algoritmiin nähden, arviointifunktio voidaan pitää samana negamaxia käyttäessä. Negamax-algoritmi kääntää arviointifunktion tuloksen etumerkin riippuen kumman pelaajan kannalta pelitilannetta tarkastellaan.

Alfabeta-algoritmi parantelee minimax-algoritmia karsimalla haun piiristä alipuita, jotka eivät tule vaikuttamaan haun lopputulokseen. Se toimii samalla tavalla kuten ylläkuvattu minimax-algoritmi. Jos se arvioi pelipuun solmua, joka on jo todettu huonommaksi kuin jo löydetty, se ohittaa koko haaran arvioinnin. Alfabeta-haulle annetaan tyypillisesti lähtöarvoiksi ∞ ja -∞, ja algoritmi alkaa etsiä lopputulosta näiden välistä. Alfabetalla päästään aina samaan lopputulokseen kuin minimax-algoritmillakin, mutta huomattavasti nopeammin. Tyypillisesti alfabeta jättää 40 % pelipuusta tarkastelematta.



Kuvio 4. Alfabeta-algoritmin kulku neljän siirron hakusyvyydellä

Kuva noudattelee samaa kaavaa kuin yllä oleva minimax-algoritmi. Tällä kertaa osa hakupuusta voidaan jättää arvioimatta kokonaan. Kuvassa on kysymysmerkein merkitty hakupuun kohdat, jotka eivät vaikuta lainkaan algoritmin lopputulokseen. Esimerkiksi solmua $?_1$ ei tarvitse käsitellä lainkaan, sillä vastustaja valitsee aina kohteekseen 10-merkityn alipuun välttääkseen mahdollisen 15 arvoisen päätesolmun. Tällaista katkaisua kutsutaan beta-katkaisuksi. Vastaavasti solmussa $?_2$ tapahtuu katkaisu. Riippumatta sen tai sen alla olevista arvoista, sitä ei koskaan valita, koska vasemmalla olevassa pelipuun haarassa on pelaajan kannalta parempi arvo. Tapahtuu alfa-katkaisu, joka karsii tässä tapauksessa kokonaisen alipuun turhana.

Algoritmien, kuten alfabetankin, tehokkuus riippuu, missä järjestyksessä se tarkastaa siirtoja. Jos algoritmi saa heti hyvän siirron tarkasteltavaksi, se saa rajattua alfa- ja beta-arvot hyvin pieneen ikkunaan aiheuttaen nopeasti alipuiden karsintaa. Jo yksinkertaisella siirtojen järjestämisellä, kuten kokeilemalla ensin lyöntejä, uhkauksia ja eteenpäinsiirtoja, päästään lähemmäksi algoritmin optimaalista suoritusta [3, s. 169].

Toinen tapa järjestää siirtoja on kokeilla entuudestaan tunnettuja hyviä siirtoja ensin. Esimerkiksi hakupuuta luodessa tarkastetaan edellinen siirto ensin. Vaihtoehtoisesti voidaan suorittaa iteratiivista pelipuun muodostusta ja hakea siirtoja ensin yhden siirron syvyyteen ja valita näistä paras siirto algoritmin ensimmäiseksi tarkastelukohteeksi. Tällaisia siirtoja kutsutaan mahtisiirroiksi (killer move) [3, s. 170].

Yleisin moderneissa shakkimoottoreissa käytetyistä algoritmeista tukeutuu siirtojen järjestelyyn. Negascout-algoritmi, päävariaatiohaku, toimii negamax-algoritmin tapaan käyttäen alfabeta-algoritmin katkaisukohtia. Se eroaa alfabetasta käyttämällä tyhjän ikkunan hakua ensimmäiseksi harkittuihin siirtoihin. Tyhjän ikkunan haussa alfabetalla annetaan alfa- ja beta-arvoiksi sama arvo. Kun pelipuuhun tehdään näin matalan syvyyden haku tyhjällä ikkunalla, saa algoritmi joko pelitilanteen arvon tai tiedon, että valittu siirto ei ollut kyseisessä pelitilanteessa paras, vaan vaatii koko alipuun normaalin käsittelyn. Jos algoritmin saama ensimmäinen siirto oli paras, sen ei tarvitse tarkastella alipuuta sen pidemmälle. Jos siirtojen järjestely ei toimi oikein, on negascout hitaampi kuin normaali alfabeta-algoritmi.

Shakissa voidaan eri siirtosarjoilla päästä samaan pelitilanteeseen. Tehostaakseen shakin siirtojen muodostusta tai pelitilanteen arviointia voidaan shakkitietokoneessa käyttää siirtovaihtotaulukoita. Niihin tallennetaan peliaseman heuristisen arvioinnin tulos. Samoin siihen voidaan tallentaa peliasemasta muodostettavat siirrot. Aina kun saavutaan pelitilanteeseen, tarkastetaan, onko kyseisen tilanne arvioitu tai siirrot luotu. Jos on, voidaan tiedot lukea suoraan taulukosta ja näin vältetään laskemasta niitä uudelleen. Taulukoiden käytössä muodostuu ongelmaksi niiden jatkuva kasvu pelin edetessä. Taulukon kanssa on kehitettävä jokin strategia, jonka perusteella arvoja poistetaan siirtovaihtotaulukosta. Näin taulukko saadaan pysymään käytettävissä mittapuissa. Siirtovaihtotaulukon tehokas käyttö voi tuplata shakkikoneen hakusyvyyden [3, s. 170-171.]

Karsinta

Shakin valtava pelipuu tekee koko pelipuun käsittelyn mahdottomaksi. Näin ollen jokainen shakkimoottori suorittaa jonkinlaista karsintaa. Yksinkertaisinta on karsia pelipuu tiettyyn syvyyteen. Näin pelipuuta tarkastellaan, ja kun annettu hakusyvyys on saavutettu, se katkaistaan. Vähemmin käytetty tapa on suorittaa esikarsintaa siirtoja luodessa. Tässä tavassa on riskinsä. Karsimalla väärät siirrot voi tekoäly jättää tärkeitä pelitilanteita kokonaan käsittelemättä. Kaikilla karsintatavoilla on riski joutua horisonttivaikutuksen uhriksi.

Horisonttivaikutuksella tarkoitetaan tilannetta, jossa tekoäly ei huomaa huonoa asemaansa karsinnan vuoksi katkaistun pelipuun takia. Esimerkiksi tekoälyn viimeinen siirto on lyödä kuningattarella vastustajan sotilas. Jos karsinta katkaisee pelipuun tarkastelun tähän siirtoon, jää tekoälylle luulo, että se on saanut sotilaan verran etumatkaa, vaikka vastustaja voisi seuraavalla siirrollaan lyödä kuningattaren. Tällaisia tilanteita voidaan yrittää paikata erilaisilla pelipuun laajennuksilla.

Laajennukset

Normaalisti kun hakualgoritmi saavuttaa annetun maksimisyvyyden, se arvioi tilanteen ja palaa, vaikka pelilaudan tilanne jättäisi pelinappulan uhatuksi. Tällöin voidaan pelipuun käsittelyä jatkaa erilaisilla laajennuksilla. Niiden tarkoituksena on tiettyjen kriteerien mukaisesti jatkaa pelipuun tarkastelua ongelmakohdista, jotta horisonttivaikutukselta vältyttäisiin.

Yksinkertainen laajennus on rauhallisuushaku (Quiescence Search). Siinä tarkastetaan, millaiseen pelitilanteeseen pelipuun lehtisolmu on jäänyt. Pelipuun käsittelyä jatketaan, jos viimeinen pelitilanne ei ole rauhallinen. Rauhallisella pelitilanteella tarkoitan pelitilannetta, jonka aiheuttava siirto ei uhkaa tai lyönyt vastustajan pelinappulaa tai ole korotus. Mikäli pelitilanne seuraa esimerkiksi pelinappulan lyöntiä, jatketaan pelipuun tarkastelua kyseisen pelitilanteen osasta vielä yhden siirron verran syvemmälle. Tätä jatketaan, kunnes saavutetaan rauhallinen pelitilanne, jolloin vältytään osittain horisonttivaikutukselta.

Yksittäislaajennus (Singular extension) on IBM:n Deep Thought- ja Deep Blue -shakkikoneiden käyttämä tekniikka. Sen ideana on pelipuuta läpikäydessä tarkkailla erityisen hyviä yksittäisiä siirtoja (singulars). Näiden yksittäisten siirtojen aiheuttamat alipuut tutkitaan muuta pelipuuta syvemmälle. Näin pyritään varmistumaan, ettei kyseissä siirrossa piile vaaroja syvemmällä pelipuussa. Kyseinen laajennus on erityisen raskas, sillä se lisää pelipuun hakusyvyyttä yhdellä jokaisen yksittäissiirron kohdalta. Deep Thought- ja Deep Blue -shakkikoneiden tehokkaat prosessorit tekivät tästä kannattavan taktiikan, mutta perinteisille tietokoneille tämä laajennus jää liian raskaaksi ollakseen hyödyllinen.

Pelitilanteen arviointi

Pelitilanteiden paremmuuden arviointi on olennainen osa shakin siirtojen valinnassa. Parhainkin hakualgoritmi on turha, ellei siirrosta seuraavaa pelin tilannetta osata arvioida oikein. Täydellistä algoritmia pelitilanteen paremmuuden arviointiin ei ole kehitelty, mutta useat shakkimestarit ovat antaneet oman näkemyksensä, kuinka niitä voidaan heuristisesti arvioida [6].

Yksi heuristiikka pelitilanteen arvioinnille on materiaalinen tasapaino. Se kuvaa kyseessä olevan pelaajan pelinappuloiden määrää ja laatua vastustajaan nähden. Shannon esitti seuraavat pistearvot pelinappuloille: kuningatar on 9, torni 5, lähetti ja ratsu 3 ja sotilas 1 pistettä. Kuninkaalle annetaan isoin arvo, tyypillisesti ääretön, sillä sen menetys katkaisee pelin. Pelin materiaalinen tasapaino saadaan selville laskemalla positiivisena kaikki vuorossa olevan pelaajan pelinappuloiden arvot ja vastaavasti negatiivisena kaikki vastustajan pelinappuloiden arvot. Näin saatu luku kertoo, kummalle puolelle saatu tulos on suosiollinen. Nolla tarkoittaa, että pelaajat ovat samanarvoisia, negatiivinen luku vastustajan ja positiivinen omaa etua. Pelinappulan arvo ei anna täyttä kuvaa sen hyödystä. Torni, joka on omien nappuloiden takana eikä pääse liikkumaan laudalla, on pelaajan kannalta hyödytön. Useasti pelitilanteen arvioinnissa tarkastellaan nappuloiden mahdollisia siirtoja ja pelinappuloiden asemaa materiaalisen tasapainon lisäksi.

Shakkilauta voidaan jakaa eri alueisiin. Se voidaan jakaa keskeltä kahtia mustan ja valkoisen pelaajan alueisiin. Pelin alkutilanteessa laudan tyhjää osaa kutsutaan keskikohdaksi. Laudan keskikohdan hallinta on edellytys pelissä menestymiselle. Mitä useampaa ruutua pelaaja uhkaa pelilaudan keskeltä, tai missä sijaitsee pelaajan nappula, sen parempi. Keskeltä lautaa pelaajan on helpompi suojata omaa puolta pelilaudasta ja hyökätä vastustajan puolelle. Samoin yksittäiset siirrot, kuten linnoittaminen ovat eduksi pelaajalle. Linnoittamisessa kuningas saadaan vietyä turvaan omien sotilaiden taakse ja vedettyä torni pois pelilaudan nurkasta suojaamaan kuningasta. Linnoittamisesta saadaan hyötyä vain pelin alkupuolella. Loppupuolella, kun pelinappuloita ei ole enää paljoa, voi vastustaja saada kuninkaan mattiin, jos kuningas on omien sotilaiden takana ilman pakoreittiä. Pelitilanteiden arvioinnissa täytyy ottaa huomioon, missä vaiheessa peliä ollaan.

Upseerien pelinappuloille voidaan antaa eri pisteytys riippuen pelilaudan tilanteesta. Lähettipari on parempi kuin yksittäinen lähetti. Jos pelilaudalla on paljon pelinappuloita valkeilla ruuduilla, on valkoisen ruudun lähetti huonompi kuin mustien ruutujen lähetti. Ratsut toimivat parhaiten, kun pelissä on vielä paljon pelinappuloita liikkeessä. Ratsu on ainut pelinappula, joka voi siirtyä muiden pelinappuloiden yli. Ratsulla voidaan taktikoida parhaiten pelilaudalla, jossa on paljon nappuloita. Pelin loppupuolella kun pelilaudalla on tilaa lähettien ja tornien liikkeille, ovat ratsut huonommassa asemassa.

2.8 Tehokkuuden arviointi

Shakkimoottorin tehokkuuden arviointi on helppoa, jos se tukee jotain olemassa olevista shakkiprotokollaa. Näin shakkimoottorin voi laittaa pelaamaan jo olemassa olevia moottoreita vastaan. Tällöin voidaan ajaa useita pelejä saaden tietoa shakkimoottorin todellisesta vahvuudesta.

Shakkimoottoreiden tehokkuutta arvioidaan samalla tavalla kuin ihmispelaajienkin tehokkuutta. Yleisin shakissa käytettävä arviointiperuste on Elo-luku. Se ilmaisee pelaajan tai shakkitekoälyn suhteellista tehokkuutta. Luku ilmaisee suhteellisen eron kahden pelaajan taidoissa. Elo-luku on nimetty sen kehittelijän, Arpad Elon mukaan [12]. Pääpiirteissään pelaajan Elo-luku kasvaa, jos hän pelaa odotettua paremmin ja tippuu, jos hän pelaa odotettua huonommin. Alla kuvattu Elo-lukujen skaalaa. FIDE on kansainvälinen shakkijärjestö, joka ylläpitää omaa Elo-arvojen listaa pelaajista.

Taulukko 4. Elo-lukujen arvoasteikko

Elo-luku alue	**Kategoria**
2600+	Maailmanmestaruuskisaajat
2400 – 2600	Suurin osa Suurmestari-tason pelaajista ja kansainvälisistä mestareista
2300 – 2400	FIDE-mestarit
2200 – 2300	FIDE-kanditaattimestarit, useimmat valtakunnalliset mestarit
2000 – 2200	Mestarikandidaatit, Ekspertit (USA)

1800 – 2000	A-luokka, Kategoria 1
1600 – 1800	B-luokka, Kategoria 2
1400 – 1600	C-luokka, Kategoria 3
1200 – 1400	D-luokka, Kategoria 4
Alle 1200	Noviisit

Elo-lukuja käytetään hieman muunneltuna monessa paikassa, joten niitä ei voida suoraan verrata keskenään. Vain saman paikan suorittamat arvioinnit tulisi huomioida. Pelimoottoreita arvioidessa tulee huomata, että testit ajetaan samalla raudalla. Esimerkiksi bittilautoja hyödyntävät ohjelmat saavat suuren hyödyn 64-bittisistä käyttöjärjestelmistä. Korkein mitattu Elo-luku on Kasparovin 2851 [3, s. 193]. Aiemmin esitellyn Hydra-shakkitietokoneen Elo-luvun arvioidaan olevan jossain 2850 ja 3000 välissä. Rybka on luokiteltu 2900–3100 välille. Tämä tulos ei ole kovin luotettava, sillä Rybkan otanta on melko suppea. Nykyaikaisia shakkitietokoneita tarkastellessa voidaan sanoa, että tietokoneet ovat ohittaneet vahvimmatkin ihmispelaajat.

Toinen tapa mitata shakkimoottorin tehokkuutta on tarkastella sen mahdollisten siirtojen muodostamiseen kuluvaa aikaa. Perft, eli performance test, funktion tarkoituksena on testata tätä. Siirtojen tehokkuuden mittaamisessa yksinkertaisesti ajastetaan, kuinka kauan perft-funktiolla kestää laskea kaikki siirrot jostain tietystä pelitilanteesta johonkin tiettyyn syvyyteen asti. Perft-funktiolla saatuja siirtojen lukumääriä voidaan verrata muiden shakkimoottoreiden saamiin tuloksiin. Näin voidaan varmistua, että shakkimoottori osaa muodostaa shakin siirrot oikein.

Yksinkertaisimmillaan perft-funktio laskee kaikki sallitut siirrot ottamatta kantaa, millainen siirto on. Funktiota voidaan laajentaa pitämään yllä siirtojen tyypeistä, kuten lyönneistä ja korotuksista. Näin voidaan paremmin rajata, missä pelimoottori tekee virheen, jos siirtojen määrät eivät täsmää laskettujen arvojen kanssa.

Taulukko 5. Shakin alkutilanteesta saatavat perft-tulokset

Syvyys	**Siirtoja**	**Lyöntejä**	**Ohesta-lyönnit**	**Linnoitukset**	**Korotukset**	**Shakit**	**Matit**
1	20	0	0	0	0	0	0
2	400	0	0	0	0	0	0
3	8902	34	0	0	0	12	0
4	197281	1576	0	0	0	469	8
5	4865609	82719	258	0	0	27351	347
6	119060324	2812008	5248	0	0	809099	10828

Taulukosta 5 käy ilmi shakin mahdollisten siirtojen eksponentiaalinen kasvu. Tehokas siirtojen muodostus on erityisen tärkeää, jos haluaa shakkimoottorinsa antavan hyvän vastuksen.

3 Funktionaalinen ohjelmointi

3.1 Lambdakalkyyli

Funktionaalinen ohjelmointi perustuu Alonzo Churchin lambdakalkyyliin (Lambda Calculus) [13]. Lambdakalkyylissa mallinnetaan abstrakti ohjelmointikielen malli, joka perustuu lambdalausekkeisiin. Se näyttää, kuinka ongelmat voidaan ratkaista vain luomalla lambdalausekkeita ja kutsumalla niitä. Tämä eroaa Alan Turingin kehittelemästä Turingin koneesta, jossa koneen tilaa muokataan annettujen käskyjen mukaan. Turingin kone ja lambdakalkyyli vastaavat toisiaan. Molemmilla saavutetaan sama lopputulos, mutta ainoastaan toteutukset eroavat.

Lambdakalkyyli on matematiikan logiikkaan perustuva funktioiden formaali esitysmuoto, joka koostuu lambdalausekkeista. Lambdalausekkeet kirjoitetaan prefix-muodossa ja lisäksi funktio ja sen parametrit kirjoitetaan peräkkäin ilman sulkeita. Esimerkiksi normaalisti matematiikassa ja yleisesti ohjelmointikielissä funktiokutsu kirjoitetaan sin(x), mutta lambdakalkyylissa se kirjoitetaan sin x. Jos funktio ottaa useamman parametrin, ne lisätään peräkkäin. Esimerkiksi 3 + 4 + 6 muuttuu muotoon + 3 4 6. Sulkeita voidaan käyttää ryhmittelyyn, jos suoritusjärjestys ei muuten ole selvä. Matematiikassa funktiot on tapana esittää yhtälöinä

f(x) = 3x^2 + 4

Lambdakalkyyli ei nimeä funktioita lainkaan. Yllä oleva funktio olisi lambdakalkyylin mukaisesti esitettynä:

λx.+ (* 3 x x) 4

Lausekkeen alussa oleva kreikkalainen lambda-merkki osoittaa, että kyseessä on funktion muodostus. Sitä seuraava x on funktion formaaliparametri. Piste erottaa funktion määritelmän funktion rungosta. Sulut ovat ryhmittelemässä kertolaskun suoritettavaksi ensin, ja lopuksi tulokseen lisätään 4.

Lambdaesitystavalla kirjoitettua funktiota voidaan kutsua seuraavasti.

(λx.+ (* 3 x x) 4) 2

Ylimääräiset sulut lisätty osoittamaan, ettei viimeinen 2 kuulu funktion runkoon. Lambdalausekkeita voidaan sitoa muuttujiin käyttämällä yhtäsuuruusmerkkiä, kuten matematiikassa on tapana.

F = λx.+ (* 3 x x) 4

Tämän jälkeen aiemmin esitetty lauseke voidaan ilmaista F 2. Funktiorunko voi sisältää myös uuden funktion.

N = λy.(λx.+ (* y x x) 4)

Kun funktiota N kutsutaan N 3, saadaan aiemmin esitelty F funktio paluuarvoksi. Se voidaan myös tulkita funktioksi, joka ottaa kaksi parametria eli N 3 2 tulisi palauttaa 16. Funktion luominen ja kutsuminen on ainut mitä tarvitaan. On huomioitavaa, että lambdakalkyylin mukaisesti määritetyt funktiot ottavat vain yhden parametrin.

Lambdakalkyylin funktion sanotaan sitovan sille annetun muuttujan funktiorungossa. Funktiossa λx.E, jossa E kuvaa lambdalauseketta, kaikki muuttujan x ilmentymät E:n sisällä ovat sidottu. Kaikki muut muuttujat E:n sisällä ovat vapaita muuttujia, elleivät ne ole muun lambdalausekkeen sitomia, kuten muuttuja y alla olevassa funktiossa.

λx.λy.xy

Lambdalausekkeen virallinen määritys on

M ::= c | x | M M | λx.M.

Sanallisesti ilmaistuna tämä tarkoittaa, että lambdalauseke muodostuu joko vakiosta c, muuttujasta x, funktiokutsusta tai funktion luomisesta. Lambdalauseke, jossa ei esiinny vakiotermiä, ja jossa ei ole vapaita muuttujia kutsutaan puhtaaksi lambdalausekkeeksi. Lambdakalkyylin mukaan vakioita ovat esimerkiksi numerot ja matemaattiset operaattorit kuten + ja *.

Lambdalausekkeilla laskeminen tapahtuu sieventämällä. Ensin funktion formaalit parametrit korvataan funktiolle annetuilla parametreilla ja sievennystä jatketaan matematiikasta tutuin säännöin.

(λx.+ (* 3 x x) 4) 2
+ (* 3 2 2) 4
+ 12 4
16

Kun sieventäminen ei enää onnistu, lauseke on saavuttanut normaalimuodon.

Tämä käsittää hieman yleistäen, kuinka tyypitön lambdakalkyyli toimii. Siinä funktiota ei voi rajata toimimaan vain numeroille, sillä nimensä mukaan se ei ota kantaa tyyppeihin. On myös tyypitetty lambdakalkyyli, joka laajentaa lambdakalkyylia lisäämällä siihen tyyppimäärittelyt. Tyypitön lambdakalkyyli voidaan nähdä tyypitetyn lambdakalkyylin erikoistapauksena, jossa käytössä on vain yksi tyyppi. Tyypitetty lambdakalkyyli on perusteena useille funktionaalisille kielille ja ohjelmointikielissä oleville tyyppijärjestelmille.

3.2 Funktionaalisten ohjelmointikielten ominaisuuksia

Funktionaalinen ohjelmointitapa eroaa AMK-tason oppilaitoksissa yleisesti opetetusta olio-ohjelmoinnista. Siinä missä olio-ohjelmointi keskittyy tilan ja tiedon kapselointiin olioiden sisälle, funktionaalinen ohjelmointi keskittyy nimensä mukaan funktioihin ja pyrkii tilattomaan ohjelmointiin. Funktionaaliset ohjelmointikielet muistuttavat yllä esitettyä lambdakalkyylia. Kuvaan seuraavaksi hieman yleisesti funktionaalisissa ohjelmointikielissä olevia ominaisuuksia. Pyrin listaamaan jokaisesta ominaisuudesta yksinkertaisen esimerkin, jotta niiden hahmottaminen on helpompaa. Käytän esimerkeissä Haskell-ohjelmointikieltä, joka tukee elegantisti kaikki listattuja ominaisuuksia. Esimerkeissä esiintyvä merkkijono ”ghci>” kuvaa Glasgow Haskell Compiler –kääntäjän komentotulkkia. Ilman komentotulkkia olevat rivit esittävät aikaisemman rivin suorituksesta seuraavaa arvoa. Haskell-tulkin toiminta on rajattua, joten funktiot tulee määrittää let-funktion avulla. Haskell-lähdekoodissa esimerkit olisivat selkeämpiä. Siinä let-funktio voidaan jättää pois ja puolipisteet voitaisiin korvata rivinvaihdoilla. Luvun lopussa käyn vielä isomman esimerkkifunktion Tursas-shakkimoottorista läpi yksityiskohtaisesti. Esimerkki käyttää Clojure-ohjelmointikieltä mutta siitä enemmin luvun loppupuolella.

Muuttumattomat tietojäsenet

Yksi ensimmäisistä erovaisuuksista on funktionaalisten ohjelmointikielten muuttujat. Yleisesti funktionaalisissa ohjelmointikielissä ei ole sijoitusoperaatiota, vaan niissä voidaan vain sitoa nimi jollekin tietylle arvolle. Kaikki muuttujat ovat siis vakioita. Yleisesti myös tietorakenteet pysyvät muuttumattomina. Tämä hidastaa

tietorakenteiden käsittelyä, sillä tietorakennetta ”muuttaessa” joudutaan siitä luomaan uusi kopio. Yleensä ohjelmointikielen kääntäjä ratkaisee tämän jakamalla tietorakenteen yhteiset osat. Esimerkiksi jos puu-rakenteeseen lisätään uusi arvo, ei koko rakennetta kopioida, vaan ainoastaan muutoksen sisältävä pelipuun haara luodaan uusiksi. Muut puu-rakenteen osat jaetaan uuden kopion ja vanhan puu-rakenteen välillä. Prosessi on edelleen hitaampi kuin puun suora muutos, mutta ero ei ole enää niin suuri.

Useat ohjelmointikielet käyttävät virheellisesti aliohjelmista tai proseduurikutsuista funktion nimeä. Funktionaalisissa ohjelmointikielissä funktiot noudattavat matemaattista funktion määritelmää. Funktion tulee palauttaa sama arvo, jos parametrit pysyvät samoina. Funktiot eivät myöskään aiheuta sivuvaikutuksia, vaan ne voivat vain palauttaa niille annettujen parametrien mukaiset tulokset. Näin funktion rajapinta muuhun ohjelmaan näkyy sen määritelmästä. Tästä on suuri etu ohjelman testaamisessa ja kehittelyssä. Jos funktio f ja funktio g saavat samat parametrit ja palauttavat saman tuloksen, voidaan niitä pitää samoina riippumatta niiden sisäisestä toteutuksesta. Funktiot ovat siis eri viittaustapoja samaan arvoon. Tätä ominaisuutta kutsutaan viittausten läpikuultavuudeksi.

Sivuvaikutusten hallinta

Matemaattinen funktioiden ja muuttujien malli sopii hyvin matematiikan piiriin, mutta oikeissa ohjelmissa vaaditaan muutakin. Esimerkiksi ohjelmalta saatu syöte pitää saada tulostettua näytöllä käyttäjää varten. Esimerkiksi Javan int read() –aliohjelma ei ole sallittu funktionaalisissa kielissä. Jotta read-aliohjelma toimisi kuten funktio, olisi sen palautettava sama arvo kutsusta toiseen. Tämä on ongelma funktionaalisissa ohjelmointikielissä. Niissä pyritään tilattomaan toimintaan, mutta ympäröivällä maailmalla on tila, esimerkiksi mitä kohtaa tiedostosta luetaan. Haskell-ohjelmointikieli on ratkaissut syötteen ja tulosteen kanssa toimimisen kertomalla kääntäjälle tarkasti, missä ohjelmakoodin kohdissa on sivuvaikutuksia. Näin ohjelmointikielen kääntäjä tietää, että kyseinen osa ohjelmaa käsittelee ulkopuolista maailmaa ja sisältää tilan käsitteen. Kääntäjä voi näin tarkistaa muun ohjelmakoodin, ettei sivuvaikutuksia sisältävää koodia ole merkittyjen alueiden ulkopuolella.

Funktio täysvaltaisena arvona

Funktionaalisissa ohjelmointikielissä funktiot ovat täysvaltaisia arvoja. Täysvaltaisen arvon tulee toteuttaa seuraavat ehdot:

- Niitä voi tallentaa muuttujiin ja tietorakenteisiin.
- Niitä voi välittää parametriksi funktioille.
- Niitä voi palauttaa funktion paluuarvona.
- Niitä voi luoda ajon-aikana.
- Niillä on oma identiteetti.

Muut funktioiden käsittelyyn liittyvät ominaisuudet riippuvat juuri tästä ominaisuudesta.

Anonyymit funktiot

Funktionaalisissa ohjelmointikielissä voidaan luoda funktioita ajon aikana lisää. Näin luodut funktiot ovat nimettömiä eli anonyymeja funktioita. Koska ne esiteltiin ensimmäisen kerran lambdakalkyylissä, yleensä niistä käytetään termiä lambdafunktio tai lambdalauseke.

```
ghci> (\x -> x + 2) 2
4
```

Haskell käyttää \-merkkiä kuvaamaan kreikan lambda-merkkiä. Kyseinen esimerkki luo anonyymin funktion, joka lisää sille välitettyyn parametriin 2. Esimerkissä funktiota kutsutaan arvolla 2 ja komentotulkki antaa tulokseksi 4.

Sulkeumat

Ajon aikana muodostettuihin funktioihin liittyy vahvasti sulkeumien käsite. Siinä muodostetaan uusi funktio, joka käyttää funktion luomisen ympäristössä määritettyä arvoa. Näin funktio muodostaa sulkeuman sille välitettyjen vapaiden arvojen ympärille ja muistaa kyseisen arvon, vaikka sitä kutsuttaisiin sen alkuperäisen luomisympäristön ulkopuolelta. Sulkeutumien avulla voidaan piilottaa tilaa funktioihin. Niiden avulla voidaan luoda kontrolli-rakenteita ja olio-ohjelmoinnista tuttuja olioita. Yksinkertainen esimerkki sulkeumasta on summain-funktion luominen.

```
ghci> let adder = \n -> (\x -> x + n)
ghci> let add1 = adder 1
add1 2
3
```

Yllä olevassa yksinkertaisessa esimerkissä luodaan adder-funktio. Se ottaa yhden parametrin, joka sidotaan n muuttujaan, ja se palauttaa uuden funktion, joka summaa sille annetun parametrin n muuttujan kanssa. Kun toisella rivillä kutsutaan adder-funktiota parametrilla 1, se välittyy uudelle funktion määritykselle ja uusi funktio sulkee n parametrin sisäänsä luoden sulkeuman. Normaalisti ilman sulkeumia adder-funktion kutsun jälkeen, n-muuttujan arvon elinkaari olisi päättynyt ja sen varaama muisti vapautettaisiin. Tällöin kutsuessa add1-funktiota sattuisi poikkeus, sillä n-muuttujan arvo olisi tuntematon. Sulkeuman takia add1-funktio muistaa, että sitä kutsuttiin arvolla 1. Olio-ohjelmointi pyrkii kapseloimaan tilan olioiden sisään, mutta funktionaalisissa kielissä tila voidaan kapseloida sulkeumien avulla.

Funktion osittainen soveltaminen

Yksi täysvaltaisilla funktioilla suoritettava ominaisuus on niiden osittainen soveltaminen. Siinä funktiolle annetaan vähemmin parametreja, kuin se ottaa vastaan. Näin funktio palauttaa uuden funktion, joka ottaa vähemmin parametreja. Esimerkiksi, jos useamman parametrin ottavalle funktiolle annetaan yksi parametri, palauttaa se uuden funktion, joka tarvitsee vain loput alkuperäisen funktion parametreista.

```
ghci> (dropWhile isSpace)
```

dropWhile –funktiota sovelletaan yllä olevassa esimerkissä osittain. Se pudottaa kaikki predikaatti-funktion toteuttavat merkit syötteestä mutta isSpace-funktiolle ei vielä välitetä parametreja. Predikaatti-funktio on funktio, joka palauttaa sille annetun parametrin tuloksena tosi tai epätosi arvon. Näin (dropWhile isSpace) –funktiokutsu palauttaa uuden funktion, joka ottaa yhden merkkijonon parametriksi ja palauttaa uuden merkkijonon, josta puuttuu alkuperäisen merkkijonon alussa olevat välilyönnit.

Curry-muunnos

Curry-muunnoksella tai ”kurituksella” voidaan funktiot, jotka ottavat useamman parametrin, ilmaista yhden parametrin ottavien funktioiden ketjuna. Ketjun jokainen

funktio ottaa vain yhden parametrin ja palauttaa uuden funktion, joka ottaa yhden parametrin. Kuritusta voidaan käyttää missä tahansa ohjelmointikielessä, joka tukee sulkeumia. Haskell-ohjelmointikielessä kaikki funktiot ovat lambdakalkyylin mukaisesti yhden parametrin funktioita ja useamman parametrin funktiot ovat curry-muunnoksen avulla kutsuttavia funktio-ketjuja.

Kompositio

Funktionaalisissa ohjelmointikielissä voidaan funktioita yhdistää matematiikasta tutun komposition mukaisesti. Esimerkiksi f ◦ g voidaan ilmasta funktiona, jonka parametri välitetään funktiolle g. Tämän lopputulos välitetään funktion f parametriksi, ja näin saatu arvo on koko muodon paluuarvo. Komposition avulla voidaan ajon aikana luoda uusia funktioita nopeasti, uudelleen käyttäen aiemmin määriteltyjä funktioita pohjana.

```
ghci> let capCount = length . filter (isUpper . head) . words
ghci> capCount "Hello there, Mom!"
2
```

Esimerkissä luodaan uusi funktio capCount, joka laskee, kuinka monta isolla kirjaimella alkavaa sanaa annetussa merkkijonossa on. Uusi funktio muodostetaan yhdistelemällä jo tunnettuja funktioita. Funktio sidotaan let-funktiolla capCount-nimeen, jottei sitä tarvitse kirjoittaa aina uudestaan.

Korkeamman kertaluvun funktiot

Täysvaltaisiin funktioihin liittyvät korkeamman kertaluvun funktiot. Korkeamman kertaluokan funktiot ovat funktioita jotka saavat parametriksi funktion ja/tai palauttavat funktion tuloksenaan. Yleisimmät korkeamman kertaluokan funktiot funktionaalisissa ohjelmointikielissä ovat map, filter ja fold. Näiden funktioiden nimet voivat hieman vaihdella ohjelmointikielestä riippuen, mutta toteutus on sama.

Ensimmäinen ja ehkä yleisin korkeamman kertaluvun funktio on map-funktio. Sen tarkoituksena on suorittaa muunnos sille välitetylle kokoelmalle. Se ottaa parametriksi funktion ja kokoelman. Se suorittaa sille annetun funktion kokoelman jokaiselle alkiolle ja palauttaa näin saadut tulokset listassa.

```
ghci> map (dropWhile isSpace) [" a", "f", "    e"]
["a","f","e"]
```

Filter-funktio karsii kokoelman alkioita. Sille annetaan predikaatti-funktio ja kokoelma. Filter-funktio käy kokoelman läpi, kutsuen predikaatti-funktiota kokoelman jokaiselle alkioille ja palauttaa listan, jossa on vain predikaatin toteuttavat alkiot.

```
ghci> filter odd [3,1,4,1,5,9,2,6,5]
[3,1,1,5,9,5]
```

Fold–funktio ottaa myös funktion ja kokoelman parametreikseen mutta niiden lisäksi sille tavanomaisesti annetaan alkuarvo. Fold-funktion tarkoituksena on ”sieventää” kokoelma johonkin muotoon. Funktio toimii siten, että se lomittaa sille annetun funktion sille annetun kokoelman alkioiden väliin. Siten se suorittaa sille annetun funktion annetulle alkuarvolle ja listan ensimmäisille alkioille. Se jatkaa kutsuen funktiota seuraavaksi edelliselle tulokselle ja listan seuraavalle alkiolle. Fold-funktio on yleisesti nimetty joko fold-left ja fold-right funktioiksi riippuen, kummasta suunnasta se alkaa suorittaa funktiokutsuja.

```
ghci> foldl (+) 0 [1,2,3]
6
```

Kyseinen funktiokutsu suorittaa siis (((0 + 1) + 2) + 3) laskutoimituksen.
Foldr aloittaa suorituksen toisesta päästä: (0 + (1 + (2 + 3))). Molemmilla päästään näin yksinkertaisessa esimerkissä samaan lopputulokseen. Monimutkaisimmissa funktiokutsuissa voi kutsujärjestys vaikuttaa lopputulokseen.

Rekursio

Koska funktionaalisissa ohjelmointikielissä ei ole sijoitusoperaatioita, ei siinä voida käyttää monista ohjelmointikielistä tuttuja iterointirakenteita kuten *for, while, do*. Sen sijaan funktionaaliset ohjelmointikielet käyttävät matematiikasta tuttua rekursiota saman lopputuloksen saavuttamiseksi. Rekursiivinen funktio on funktio, joka määritellään itsensä avulla. Perinteinen esimerkki rekursiosta on fibonaccin lukuja laskeva funktio.

```
ghci> let fib n = if n <= 1 then 1 else fib (n-2) + fib(n-1)
ghci> fib 10
89
```

Useimmat ei-funktionaaliset ohjelmointikielet eivät suosi rekursiivisia funktioita tai aliohjelmakutsuja, sillä jokainen aliohjelmakutsu vie muistia pinosta. Isoilla

syötemäärillä rekursiiviset funktiokutsut voivat aiheuttaa pinon ylivuodon. Funktionaaliset ohjelmointikielet ovat yleisesti toteutettu siten, että häntärekursiiviset funktiokutsut optimoidaan kääntäjän puolesta iteraatiorakenteeksi, joka ei kärsi tästä puutteesta. Häntärekursiivinen funktio on rekursiivinen funktio, joka kuljettaa rekursiivisissa funktiokutsuissa 'kerääjäparametria, joka palautetaan funktion lopussa. Näin rekursiivisen funktiokutsun viimeisessä vaiheessa ei tarvitse viitata aikaisempiin funktiokutsuihin ja rekursio voidaan toteuttaa hyppylauseena ohjelmointikielen kääntäjän toimesta kuluttamatta pinon muistia. Haskell-kielen kääntäjä ei välttämättä tee uutta funktioita pinoon aina funktiokutsun sattuessa, joten siinä tärkeämpää on vain saada päättymisehdot kohdalleen.

Hahmonsovitus

Monet funktionaaliset ohjelmointikielet tukevat hahmontunnistusta. Sen avulla voidaan funktion määritelmä kuvata hyvin matemaattiseen tapaan. Hahmonsovituksen ideana on sovittaa parametrina saatua tietoa hahmoon. Jos sopiva hahmo löytyy, etenee ohjelman suoritus hahmon määrittelemään haaraan. Tekniikka vähentää ohjelmassa käytettävien ehtorakenteiden määrää. Esimerkiksi yllä oleva fib-funktio voidaan kirjoittaa hahmontunnistusta käyttäen seuraavasti

```
ghci> let fib 0 = 1; fib 1 = 1;  fib n = fib(n-2) + fib(n-1)
```

Nyt fib-funktiota kutsuttaessa hahmontunnistus tarkistaa ensin funktion parametrin ja sen perusteella valitsee funktiorungon, jota se kutsuu. Jos parametri on 0 tai 1, palautetaan 1, ja muussa tapauksessa parametri sidotaan n muuttujaan ja suoritetaan sitä vastaava funktiorunko.

Suoritusmekanismit

Funktionaaliset ohjelmointikielet voivat erota ohjelmakoodin suoritusjärjestyksen suhteen muista ohjelmointikielistä. Suurin osa nykyään yleisistä ohjelmointikielistä käyttää ahkeraa suoritusmekanismia. Siinä aliohjelman parametrien arvot lasketaan ja niiden lopputulokset välitetään aliohjelmalle ennen niihin siirtymistä. Tämän vastakohta on laiska suoritusmekanismi. Siinä funktion parametrit välitetään suorittamatta funktiolle. Parametrit suoritetaan vain, jos niitä tarvitaan kyseisen funktion toiminnassa ja muuten ne jäävät suorittamatta kokonaan. Laiska suoritusmekanismi mahdollistaa

päättymättömien tietorakenteiden luomisen ja käsittelyn. Näitä voidaan käyttää, kunhan vain yksittäistä osaa tietorakenteesta tarkastellaan kerrallaan.

```
ghci> let countFrom n = n : countFrom (n + 1)
ghci> sum (take 10 (countFrom 1))
55
```

Esimerkissä lasketaan kymmenen ensimmäisen luonnollisen luvun summa luomalla kaikki kokonaisluvut numerosta 1 eteenpäin ja ottamalla tästä päättymättömästä lukujonosta 10 ensimmäistä alkioita ja summaamalla ne. Laiskan suoritusmekanismin takia CountFrom-funktion tuottamaa päättymätöntä lukujonoa ei suoriteta kuin vasta take-funktiota kutsuessa ja tällöin siitä pilkotaan vain 10 ensimmäistä alkiota.

Memoisaatio

Annetaan hieman monimutkaisempi esimerkki, josta saa ehkä paremmin selville mihin laiskaa suoritusmekanismia voidaan käyttää. Aiemmin esitetyt fibbonaccin lukujonoa laskevat funktiot käyttivät rekursiota, mutta niiden ongelmana on hidas laskeminen. Suurin ongelma on, että siinä lasketaan samoja tuloksia uudestaan ja uudestaan. Kutsu fib 4 laskee fib 3 ja fib 2 tulokset, jotka puolestaan laskevat aina aiemmat fibbonaccin lukujonon luvut. Laiskalla suoritusmekanismilla voidaan aiemmin lasketut fibbonaccin lukujonon tulokset tallentaa listaan, ja lukea ne sen sijaan, että ne laskettaisiin aina uudestaan. Tätä tapaa kutsutaan memoisaatioksi. Siinä siis tallennetaan funktion parametreilla saatava arvo, jottei myöhemmin samoilla parametreilla funktioita kutsuttaessa tarvitse laskea sen tulosta uusiksi.

```
ghci> let mfib = let fib 0 = 1; fib 1 = 1; fib n = mfib(n-2) + mfib(n-1) in (map
fib [0..] !!)
```

Edellinen esimerkki ei enää ole niin selkeä esitys kuin rekursiiviset esitystavat. Sen etuna on huomattavasti nopeampi suoritusaika eikä sitä uhkaa pinon ylivuoto.

3.3 Edut

Kaksi funktionaalisten kielten ominaisuutta voidaan nostaa muiden edelle niiden etuja tarkastellessa. Nämä ovat laiska suoritusmekanismi ja korkeamman kertaluvun funktiot [14]. Korkean kertaluvun funktiot mahdollistavat isojenkin ongelmien ratkaisun selkeästi. Iso ongelma voidaan jakaa pienempiin ongelmiin ja nämä vielä pienempiin,

kunnes kaikkein pienimmät osaongelmat osataan ratkaista. Korkeamman kertaluokan funktioilla nämä osaongelmien ratkaisut voidaan taas liimata takaisin yhteen ratkaisten selkeästi isommatkin ongelmat.

Matemaattisen luonteensa takia funktionaalisten ohjelmien toiminta voidaan todistaa matemaattisesti oikeaksi induktion avulla. Funktionaaliset ohjelmointikielet eivät ole sidottuja ympäristöön kuten käännettävät ohjelmointikielet. Funktionaalisia ohjelmointikielten lausekkeita voidaan tulkata suoraan ja useat kielet tarjoavat tulkin tähän dynaamisten ohjelmointikielten tapaan. Funktionaaliset ohjelmointikielet eroavat dynaamisista ohjelmointikielistä siinä, että niiden tuottama ohjelmakoodi on tehokkaampaa.

Koska funktioiden tulokset riippuvat vain niille annetuista parametreista, voidaan niitä suorittaa rinnan ilman murhetta. Tämä ja muut tietojen muuttumattomuuden ominaisuudet tekevät funktionaalisista ohjelmointikielistä varsin kiehtovia ajatellen nykyaikaista suuntausta, jossa tietokoneiden prosessoreiden kellotaajuuksien nostamisen sijaan, nostetaan niiden sisältämien prosessoriytimien määrää. Funktionaaliset ohjelmointikielet soveltuvat erinomaisesti rinnanajettaviksi ilman yleisiä rinnanajettavat koodin murheita kuten lukituksia. Lukitukset ovat turhia, sillä mikään muuttujan arvo ei voi muuttua ohjelman aikana.

Funktionaalisen ohjelmoinnin edut eivät ole jääneet huomaamatta yleisempien ohjelmointikielien keskuudessa. Yhä useammat funktionaalisten ohjelmointikielten ominaisuudet alkavat ilmestyä muihin ohjelmointikieliin. Esimerkiksi Java-ohjelmointikielen JDK8-versioon on suunniteltu tuki sulkeumille [15]. Nykyään on olemassa lukuisia erilaisia funktionaalisia ohjelmointikieliä kuten Haskell, Erlang ja ML-kielet. Ensimmäinen funktionaalinen ohjelmointikieli oli John McCarthyn Lisp vuodelta 1958.

3.4 Lisp

Lisp on toiseksi vanhin edelleen käytössä oleva ohjelmointikieli ja vanhin funktionaalinen ohjelmointikieli. Nykyiset Lisp-perheen ohjelmointikielet eivät eroa merkittävästi alkuperäisestä Lisp-ohjelmointikielestä. Lisp ja sen ominaisuudet ovat

olleet esikuvina monille, nykyään itsestään selville ohjelmointikielten ominaisuuksille, kuten ehtolauseelle. Muita esimerkkejä, joissa Lisp on ollut edelläkävijä, ovat Puu-tietorakenne, automaattinen muistinhallinta, dynaaminen tyypitys ja itsensä kääntävä kääntäjä. Sen lähdekoodi muodostuu symbolisista lausekkeista tai s-lausekkeista. Tästä syystä Lisp käyttää lambdakalkyylistä tuttua prefix-esitystapaa. Siinä jokaisen s-lausekkeen ensimmäinen alkio on operaattori, ja loput alkiot tulkitaan operaattorin parametreiksi. Lisp on myös homoikoninen (homoiconic) kieli. Tällä tarkoitetaan, että Lisp-lähdekoodi muodostuu Lisp-tietorakenteista, tässä tapauksessa listoista. Esimerkiksi (A B C) voidaan tulkita joko listana jossa alkiot A, B ja C tai funktion A kutsuna parametreilla B ja C. Esimerkki Lisp-lähdekoodista Scheme-murteella:

```
(define fib
  (lambda (n)
    (if (<= n 1)
      1
      (+ (fib (- n 2))
         (fib (- n 1))))))
```

Lisp-ohjelmat eivät olleet kovin tehokkaita alunperin, ja yksi tätä puutetta korjaamaan tehdyistä asioista oli Lisp-tietokoneiden luominen. Nämä koneet oli suunniteltu ja optimoitu Lisp-lähdekoodin suoritukseen. Yleiskäyttöisten tietokoneiden kehitys teki erikoistuneet Lisp-tietokoneet nopeasti turhiksi. Nykyään Lisp-kielillä toteutetut ohjelmat ovat yhtä nopeita kuin mitkä tahansa muut ohjelmat [16].

Lisp-kieliä ei usein mielletä funktionaalisiksi kieliksi. Ne eivät tue kaikkia funktionaalisille kielille tyypillisiä ominaisuuksia. Esimerkiksi hahmonsovitusta, funktioiden osittaista soveltamista ja kompositiota ei ole tavanomaisessa Lisp-ohjelmakoodissa. Lisp suosii funktionaalista ohjelmointia, sillä funktiot ovat ensisijaisia arvoja ja korkeamman kertaluvun funktioita käytetään ahkerasti. Lisp-kielistä puuttuu osa funktionaalisten kielten ominaisuuksista, mutta nämä ominaisuudet voidaan lisätä kieleen makrojen avulla.

Lisp-ohjelmointikielen tulkki on hyvin yksinkertainen luoda. Tästä syystä alkoi erilaisia Lisp-murteita syntyä eri puolilla heti kielen yleistymisen jälkeen. Scheme oli yksi merkittävistä Lisp-murteista, joka kehiteltiin 70-luvulla. Se yhdisti LISP 1.5 -kielen syntaksin ja semantiikan, Algol-kielen lohkorakenteiden ja leksikaalisen näkyvyysalueiden kanssa. Scheme on myös ensimmäinen ohjelmointikieli, joka tukee jatkumien käyttöä. Jatkumoilla ohjelma voi tallentaa kyseessä olevan ohjelmantilanteen

ja myöhemmin palata tallennettuun tilaan. Toinen merkittävä Lisp-murre tuli seurauksena suurimpien Lisp-toteutuksien yhdistämisestä. Lisp-kielten heikkous oli pitkään useiden eri versioiden olemassa olo ilman yhteistä standardia. Jotta Lisp-ohjelmat saataisiin toimimaan jatkossakin, alkoi kielen standardoiminen. Näin muodostui Common Lisp, joka yhdisti ominaisuuksia kaikista valloillaan olevista Lisp-toteutuksista. Hieman myöhemmin Common Lisp standardoitiin amerikkalaisen standardointijärjestö ANSI:n puolesta ANSI Common Lispiksi. Edelleen tunnetuimmat Lisp-kielet ovat Scheme ja ANSI Common Lisp, mutta uusia Lisp-murteitakin on kehitelty, kuten Paul Grahamin Arc ja Rich Hickeyn Clojure.

3.4.1 Makrot

Yksi Lisp-kielten tärkeimmistä ominaisuuksista on makrot. Makrot mahdollistavat Lisp-kielen lähes rajattoman laajentamisen. Makrojen avulla voi luoda uusia toimintoja ohjelmointikieleen, mikä mahdollistaa erilaisen lähtökannan ongelmien ratkaisuun kuin muilla ohjelmointikielillä olisi mahdollista. Siinä missä muilla ohjelmointikielillä ohjelmoija miettii, kuinka hän voi ratkaista ongelman ohjelmointikielensä avulla, miettii Lisp-ohjelmoija, kuinka hän voi muuttaa kieltänsä, jotta ongelma ratkeaa. Tehokas makrojen käyttö on edellytys sovelluskohtaisten kielien luomiselle. Makrojen avulla ohjelmoija voi kiertää funktioiden rajoituksia. Pääasiallisesti tämä tarkoittaa suoritusjärjestyksen muuttamista. Esimerkiksi yksinkertaista if-muotoa ei voida esittää funktiona, sillä funktiot evaluoivat parametrinsa ennen funktion rungon suoritusta. Näin if-funktion else-haara tulisi evaluoitua ennen testitapausta.

Ennen Lisp-lähdekoodin varsinaista kääntämistä suoritetaan makrojen laajentaminen. Tällöin kaikki lähdekoodin makrot muutetaan niiden kuvaamaan muotoon ja seuraava koodi ei sisällä enää lainkaan makroja. Lisp-makrot eroavat muista, esimerkiksi C-kielen esikääntäjä makroista. Siinä missä C-kielessä makrot ovat yksinkertaisia tekstin korvaamisia, Lisp-makrot ovat Lisp-koodia. Tämä mahdollistaa niiden tehokkaan käytön. Lisp-makrojen avulla itse ohjelmointikielen ydin voidaan pitää minimaalisena. Alkuperäinen Lisp-kieli sisälsi vain 7 ydinfunktioita joiden avulla loput ohjelmointikielirakenteet määriteltiin. Esimerkkinä tästä tekniikasta on let-makro. Makron tarkoituksena on nimetä sille annetut muuttujat sen rungossa.

```
(let ((x 1)
      (y 2))
  (+ x y))
```

Yllä oleva esimerkki voidaan myös esittää käyttämällä lambda-funktioita.

```
((lambda (x)
  ((lambda (y)
    (+ x y)) 2)) 1)
```

Tämän avulla voidaan kielen kääntäjä pitää yksinkertaisena. Ennen ohjelman käännöstä kaikki ohjelmakoodissa olevat makrot laajennetaan. Näin kääntäjän ei tarvitse tuntea monia erilaisia ohjelmointirakenteita tai funktioita, vaan sen tarvitsee hallita muutama ydinfunktio. Makrojen tehokasta hyödyntämistä on tutkittu pitkään. Monet teokset ovat pyrkineet kuvaamaan makrojen hyötyä, kuten Paul Grahamin *On Lisp.*

3.4.2 Sovelluskohtaiset kielet

Lisp-kielten tehokas makrojen tuki mahdollistaa ohjelmointikielen vaivattoman laajentamisen. Näin käytössä oleva kieli voidaan muuttaa paremmaksi ratkaistavan ongelman kannalta. Kyseisistä kielen muutoksista puhutaan sovelluskohtaisen kielen luomisesta. Esimerkiksi dynaamisesti HTML-merkkauskielen mukaisia sivuja luodessa voi olla kannattavaa luoda makro HTML-kielen rakenteiden luomiselle. Alla olevalla makrolla on helppo luoda XML-tyylisiä tag-rakenteita, joista esimerkiksi HTML-merkkauskieli koostuu.

```
(defmacro tag (name atts &body body)
  `(progn (print-tag `,name
                     (list ,@(mapcar (lambda (x)
                                       `(cons `,(car x) ,(cdr x)))
                                     (pairs atts)))
                     nil)
          ,@body
          (print-tag `,name nil t)))
```

Edellisen makron avulla voidaan HTML-kielen mukaisia ilmauksia luoda seuraavasti.

```
(tag html ()
     (tag body ()
          (printc "Hello World!")))
```

<html><body>Hello World!</body></html>

Tag-makroa voidaan erikoistaa lähemmäksi HTML-merkkauskielen rakenteita luomalla sen rakenteille omat makrot.

```
(defmacro html (&body body)
  `(tag html ()
        ,@body))

(defmacro body (&body body)
  `(tag body ()
        ,@body))
```

Nyt Hello World! –sivun luominen on selkeämpää.

```
(html
  (body
    (princ ”Hello World!”)))
```

<html><body>Hello World!</body></html>

Näin kieltä voidaan laajentaa lähemmäksi kyseistä sovellusta.

3.5 Clojure

Clojure on uusi tulokas Lisp-kielten joukkoon. Clojure saavutti vakaan 1.0 version keväällä 2009 [17]. Se on Rich Hickeyn kehittelemä, Java-virtuaalikoneessa ajettava Lisp-ohjelmointikieli. Clojure on myös vahvasti funktionaalinen kieli muuttumattomien tietorakenteidensa kanssa. Se on suunniteltu nykyaikaisia koneita varten. Paitsi että se suosii muuttumattomia tietorakenteita, se tarjoaa tehokkaat työkalut säkeistettyjen ohjelmien luomiseen. Yksi Clojuren suurimmista eduista on sen Java-sidonnaisuus. Yleisesti Lisp-kielien ongelmana on ollut valmiiden kirjastojen puute ja niiden vaikea asennus. Clojure on ratkaissut tämän toimimalla vaivatta Java-kielen kanssa, jolloin kaikki jo olemassa olevat Java-kirjastot ovat sen käytössä. Java-sidonnaisuudella on myös haittapuolensa: Java-virtuaalikonetta ei ole suunniteltu funktionaalisille ohjelmointikielille, joten siitä puuttuu esimerkiksi häntärekursiivisten funktiokutsujen optimointi. Clojureen on tätä ongelmaa paikkaamaan jouduttu luomaan recur- ja trampoline-makrot. Javaan nähden Clojure tarjoaa elegantin, ilmaisukykyisen kielen.
Alla on yksinkertaisen Java-metodin lähdekoodi.

```
public class StringUtils {
    public static boolean isBlank(String str) {
        int strLen;
        if (str == null || (strLen = str.length()) == 0) {
            return true;
        }
        for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
                return false;
            }
        }
        return true;
    }
}
```

Semanttisesti vastaava esimerkki Clojurella:

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

Kuten edellä tulee ilmi, on Clojuren versio huomattavasti selkeämpi kuin Javan. Sen lähdekoodissa ei ole haaroja ja haluttu ongelman ratkaisu on selkeämmin esillä ilman ylimääräistä syntaksia. Java-version toiminta on piilotettuna for-silmukan sisälle.

Käydään Clojure-lähdekoodi läpi hieman syvällisemmin. Lähdekoodi on listamuodossa, jota sulkeet erottavat. Listaa ei ole aloitettu ’-merkillä, joten Lisp-lukija suorittaa sen. Suoritus tapahtuu tulkitsemalla listan ensimmäinen alkio operaattoriksi ja loput sen parametreiksi. Tässä tapauksessa listan operaattori on defn. Defn on Clojure-makro, jolla voidaan määrittää uusi funktio. Kyseessä on makro, joten se suoritetaan makron tahtiin. Makro tulkitsee seuraavan parametrin funktion nimeksi, joka on blank?. Nimi noudattaa Clojuren nimeämiskäytäntöä, jossa predikaatti-funktioiden nimet päättyvät kysymysmerkkiin. Seuraavan parametrin arvo tulkitaan uuden funktion parametrivektoriksi. Hakasulkumerkit kuuluvat Clojuren syntaksiin, ja ne tulkitaan vektoriksi. Tässä tapauksessa funktio saa yhden parametrin, jonka arvo sidotaan muuttujaan s. Kuten huomataan, Clojure on Lisp-kielten tapaan heikosti tyypitetty, joten sille ei tarvitse sanoa, mitä tyyppiä parametri s on. Loput parametrit defn-makro tulkitsee funktion rungoksi. Funktion runko on lista, jonka ensimmäinen alkio tulkitaan operaattoriksi. Tässä tapauksessa kyseessä on every?-niminen funktio. Kyseessä on predikaatti-funktio, sillä sen nimi päättyy kysymysmerkkiin. Clojuren dokumentaatio kertoo, että every?-funktio ottaa parametriksi predikaatin ja sekvenssin ja palauttaa tosi arvon, jos kaikki sekvenssin alkiot palauttavat tosi-arvon annetulle predikaatille. Annettu predikaatti on tässä tapauksessa ”#(Character/isWhitespace %)” ja sekvenssi

s. Predikaatin #(...) rakenne on Clojuren syntaksia anonyymin funktion luomiselle, ja se voitaisiin myös kirjoittaa muodossa (fn [x] (Character/isWhitespace x). Anonyymien funktioiden luominen on yleinen toiminto Lisp-kielissä, joten sille on tehty oma syntaksinsa. %-merkki kuvaa siis anonyymin funktion parametria. Character/isWhitespace kuuluu Clojuren Java-sidonnaiseen syntaksiin, ja se kutsuu java.lang.Character –luokan staattista isWhitespace() –metodia, joka palauttaa tosi arvon, jos sille annettu merkki on ”tyhjä” kuten välilyönti.

Otetaan seuraavaksi tarkasteluun hieman isompi funktio, josta saa hieman paremman käsityksen funktionaalisesta ohjelmoinnista Clojurella.

```
(defn fen-board->0x88board [s]
  (reduce (fn [board [index piece]]
            (fill-square board index (piece-value piece)))
      (init-game-board)
      (indexed
        (map-str
          #(str % "EEEEEEEE")
          (->> (replace-by #"\d" #(str (repeat (Integer/parseInt %) \E)) s)
               (split #"/+")
               reverse)))))
```

Funktio on Tursas-shakkimoottorin osa joka muuttaa FEN-esitystavalla annetun merkkijonon, kuten ”rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR” ja muuttaa sen 128-alkioiseksi vektoriksi yhdistämällä funktioita. Kuten aiemmin käytiin läpi, muodostetaan tässä fen-board->0x88board -niminen funktio. Funktio ottaa vain yhden parametrin, joka sidotaan nimeen s. Funktion runko alkaa reduce-funktiolla, joka vastaa aiemmin kuvattua foldl-funktiota. Käydään ensin läpi kuinka reduce-funktio luo kokoelman, jota se käy läpi.

```
(indexed
  (map-str
    #(str % "EEEEEEEE")
    (->> (replace-by #"\d" #(str (repeat (Integer/parseInt %) \E)) s)
         (split #"/+")
         reverse)))
```

Lisp-kielissä funktiot käydään yleensä sisältä ulospäin läpi. Tässä tapauksessa suoritus alkaa ->> -makron sisältä. ->> on makro, jolla voidaan muuttaa funktion rakennetta ottamalla listan eri muotoja parametriksi ja suorittamalla ensin ensimmäisen muodon ja välittämällä sen tuloksen seuraavan muodon viimeiseksi parametriksi. Makro siis muuttaa annetun koodin muotoon:

```
(reverse (split #”/+” (replace-by #"\d" #(str (repeat (Integer/parseInt %) \E)) s)))
```

Sen tarkoituksena on vain selkeyttää funktion eri suoritusvaiheita.
Ensimmäinen funktiosuoritus makron sisällä on:

```
(replace-by #"\d" #(str (repeat (Integer/parseInt %) \E)) s)
```

Funktio replace-by korvaa kaikki säännöllistä lauseketta (#”\d”) vastaavat kohdat merkkijonosta s funktion (#(str (repeat (Integer/parseInt %) \E))) tuloksella. Tässä tapauksessa se korvaa jokaisen merkkijonosta löydetyn numeron numeroa vastaavalla määrällä E-kirjaimia. Seuraavaksi ->>-makro välittää tämän tuloksen funktiolle:

```
(split #"/+")
```

Funktio pilkkoo annetun merkkijonon /-merkin kohdalta listaksi. Viimeisenä vaiheena lista välittyy reverse-funktiolle, joka kääntää listassa olevien merkkijonojen järjestyksen. Seuraavaksi jatketaan alla olevaan vaiheeseen.

```
(indexed
  (map-str #(str % "EEEEEEEE") ....))
```

Map-str vastaa aiemmin esiteltyä map-funktio, paitsi että se palauttaa tuloksensa merkkijonona eikä listana. Str-funktio palauttaa sille annetut parametrit merkkijonona. Map-str-funktio lisää jokaiseen listassa olevaan merkkijonoon kahdeksan E-kirjainta ja palauttaa sen merkkijonona. Välituloksena saatava merkkijono vastaa 0x88-taulukon sisältöä. Indexed-funktio palauttaa sille annetun sekvenssin pareina, joista ensimmäinen on sen indeksi sekvenssissä ja seuraava alkio indeksin kohdalla oleva pari kuten [0 R] [1 N] [2 B] ...

Tästä pääsemme vihdoin alkuperäisen reduce-funktion piiriin, joka käy tämän uuden sekvenssin läpi ja lisää alkuarvona muodostettuun pelilautaan, indeksin mukaiseen kohtaan, merkkijonossa olevan kirjaimen mukaisen arvon. Edellinen funktio kuvaa hyvin, kuinka funktionaaliset ohjelmat kasaantuvat eri funktioiden yhdistämisenä.

Clojure ei oletuksena tue laiskaa suoritusmekanismia. Tätä varten Clojure tarjoaa laiskat sekvenssit. Laiskat sekvenssit vastaavat funktionaalisen ohjelmoinnin laiskoja virtoja, jotka ovat esitetty esimerkiksi tietotekniikan klassikkoteoksessa, Structure and Interpretation of Computer Programs. Laiskojen sekvenssien ajatuksena on että koko sekvenssiä ei evaluoida sitä luodessa, vaan siitä käsitellään vain sitä tarvittaessa. Näin funktio, joka saa laiskan sekvenssin parametriksi, ei evaluoi koko sekvenssiä vaan

parametrin evaluointi palauttaa viittauksen laiskan sekvenssin alkuun. Funktiorungossa voidaan sekvenssiä käsitellä halutulla tavalla. Tekniikka mahdollistaa päättymättömät listat ja muut laiskan suoritusmekanismin hyödyt. Lukuisat Clojuren mukana tulevista funktioista palauttavat tuloksenaan laiskan sekvenssin, joten niiden käyttö on suositeltua.

4 Tursas-shakkimoottori

Opinnäytetyön tavoite on toteuttaa toimiva shakkimoottori Clojure-ohjelmointikielellä. Nimesin toteutuneen shakkimoottorin Tursaaksi. Se on saatavilla GitHub-versiohallintaportaalista osoitteessa https://github.com/zmyrgel/tursas. Kattavan shakkimoottorin rakentaminen alusta asti olisi liian iso projekti jopa opinnäytetyöksi. Tästä syystä Tursaan tarkoituksena ei ole kattaa koko shakkiprotokollan tarjoamaa toiminnallisuutta, vaan keskittyä pieneen osaan siitä. Tavoitteena on tuottaa tietokonetta vastaan pelattava shakki, jossa on sääntöjä noudattava tekoäly.

Tietokoneshakin kehittäminen funktionaalisella ohjelmointitavalla poikkeaa monista jo olemassa olevista shakkimoottoreista. Useimmissa shakkimoottoreissa funktiot muokkaavat muistissa pidettävää tietorakennetta suoraan. Funktionaalisessa ohjelmoinnissa tämä ei ole mahdollista, sillä se perustuu yleensä muuttumattomien arvojen käsittelyyn. Tämän takia on muodostettava joka vuoron jälkeen shakkilaudan tilasta uusi ilmentymä ja korvattava jo olemassa oleva tila uudemmalla. Shakki on pelinä hyvä kandidaatti funktionaaliselle ohjelmoinnille, sillä se ei sisällä kovin paljoa tilaa, jonka muutoksia valvoa. Shakkipeli voidaan esittää ilman tilan käsitettä poislukien puolisiirtojen määrää, linnoituksen ja ohestalyönnin mahdollisuutta. Tilalla tarkoitan kaikkea tietoa, joka ei näy pelilautaa katsomalla.

Tursaan lähdekoodi pyrkii erottelemaan puhtaan funktionaalisen ja sivuvaikutuksia sisältävät osat. Ainoastaan yhdessä ohjelman lähdekooditiedostossa pidetään yllä muuttuvaa tietoa tai ohjelman tilaa. Muut osat shakkimoottorista on pyritty toteuttamaan ilman sivuvaikutuksia. Näin ohjelman eri osia voidaan testata riippumatta toisistaan.

4.1 Komentotulkki

Komentotulkki toimii Tursaan ytimessä. Sen tarkoituksena on vastaanottaa tekstipohjaista syötettä ja ohjata shakkimoottorin toimintaa saadun syötteen mukaan. Samoin se tulostaa vastauksia moottorilta.
Komentotulkin lähdekoodi on yksinkertainen:

```
(loop []
  (-> (game-read)
      game-eval
      game-print)
  (recur))
```

Komentotulkin osat on erotettu yksittäisiin funktioihin. Silmukan sisässä kutsutaan game-read -funktiota. Sen ainut toiminto on lukea standardisyötevirrasta syötettä yhden rivin verran. Luettu merkkijono välitetään parametrina game-eval -funktiolle. Game-eval -funktio tulkitsee annetusta merkkijonosta shakkimoottorin komentoja. Tulkkaus tarkastaa ensin yleiset tursaan komennot ja tämän jälkeen siirtää merkkijonon tulkkauksen aktiivisen protokollan mukaiselle tulkille. Tulkkaus suorittaa toimintoja annetun komennon mukaan ja palauttaa merkkijonon tai listan merkkijonoista, jos sen seurauksena tarvitsee tulostaa jotain. Game-print funktion ainut tarkoitus on tulostaa sille annettu merkkijono tai merkkijonolista.

Tursaksen komentotulkki tukee vain suppeaa osaa Chess Engine Communication Protocol -protokollasta. Tarkoituksena on, että se tukee riittävästi, jotta valkoinen ihmispelaaja voi pelata mustaa tekoälyvastustajaa vastaan. Tursas tukee myös muutamia omia komentoja, jotka mahdollistavat shakin pelaamisen suoraan konsolissa tai auttavat Tursaan kehityksessä.

4.2 Tietorakenteet

Tursas-shakkimoottorissa käytetään kahta tietorakennetta. Olennaisin tietorakenne on kulloisenkin pelitilanteen sisältävä State. Tursas pitää muistissa listaa kaikista pelitilanteista pelin alusta lähtien. State käsittää kaiken tiedon kyseisestä pelin tilanteesta. Tästä on se etu, että shakin pelipuuta voidaan käydä läpi ilman sivuvaikutuksia.

```
(defprotocol State
  (occupied? [state index])
  (black? [state index])
  (white? [state index])
  (check? [state])
  (mate? [state])
  (draw? [state])
  (state->fen [state])
  (result [state])
  (legal-states [state])
  (legal-moves [state])
```

```
(apply-move [state move])
(turn [state])
(perft [state depth])
(dynamic? [state])
(full-moves [state])
(evaluate [state])
(last-move [state])
(game-score [state])
(game-end? [state]))
```

State-protokolla listaa monia funktioita pelitilanteen käsittelyyn. Se määrittää rajapinnan, jota voidaan käyttää State-protokollan toteuttajasta riippumatta.

```
(defrecord State0x88 [board black-pieces white-pieces])
```

State0x88-tietue toteuttaa State-protokollan. Se koostuu 0x88-mallin mukaisesta, 128 elementtiä sisältävästä vektorista, sekä erillisistä kentistä, jotka pitävät yllä tiedon kummankin pelaajan pelinappuloiden sijainnista. Pelinappuloiden erikseen tallentaminen nopeuttaa pelin käsittelyä, sillä siirtoja muodostaessa ei tarvitse iteroida koko 128 alkiota sisältävää vektoria läpi. Sen lisäksi se sisältää assosiaatiotauluissa pelinappuloiden sijainnit laudalla. Näin koko lautaa ei tarvitse käydä läpi pelinappuloita etsiessä. Muut shakin tiedot, kuten linnoittamisen mahdollisuus, on tallennettuna pelilautaa kuvaavan vektorin alkioihin, jotka jäävät pelilautaa kuvaavan osan ulkopuolelle.

Pelilautaa kuvaava vektori voidaan jakaa riveihin heksamuodossa. Näin jokaisen vektorin rivien alkiot, joiden indeksin arvo on välillä 0x00 - 0x07 käsittelee itse pelilautaa ja arvot välillä 0x08 - 0x0F jäävät pelilaudan ulkopuolelle. Näin pelilaudan lisäksi jää paljon tyhjiä alkioita, joihin tallentaa pelin kannalta oleellisia tietoja. Tursas tallentaa tyhjiin alkioihin State0x88-tietueen tarvitsemia tietoja. Haittapuolena laudalle tallentamisessa on vektorin rajallisuus. Vektori sisältää vain tavun kokoisia alkioita pitääkseen pelilaudan vaatiman tilan mahdollisimman pienenä. Näin sinne tallentavan tiedon tulee mahtua yhteen tavuun. Toinen vajaus on tiedon talletuksen haun kanssa. Jokainen tietopalanen tulee hakea vektorista, mikä tekee lähdekoodista monimutkaisempaa kuin se olisi jos itse tietue tallentaisi tiedon. Hyötyinä saadaan tehokkaampi tilan käyttö ja ohjelman tehokkuus.

Toinen Tursaan tietorakenne on yksittäistä siirtoa kuvaava Move-protokolla. Se noudattaa samaa kaavaa kuten State eli siinä on itse protokolla ja sen toteuttava Move0x88-tietue.

```
(defprotocol Move
  (move->coord [move])
  (from [move])
  (to [move])
  (promotion [move]))
```

Protokolla määrittää rajapintafunktiot siirron käsittelyyn.

```
(defrecord Move0x88 [from to promotion])
```

Move0x88-tietue toteuttaa Move-protokollan ja kuvaa siirtoa 0x88-laudalla. Se sisältää tiedon pelinappulan lähtö- ja kohdeindeksistä sekä korotukseen valitusta pelinappulasta. Indeksit tallentuvat 0x88-pelilaudan alkioita vastaavina lukuarvoina. Näin 0x88-pelilautaa käyttävien funktioiden ei tarvitse muuntaa lähde- ja kohdeindeksien arvoja tarpeettomasti. Apufunktio move->coord muuntaa sille välitetyn siirron koordinaattiesitykseen, kuten "e1g1". Myös muut funktiot palauttavat arvonsa koordinaattiesityksessä.

4.3 Siirtojen muodostaminen

Toisin kuin useimmat shakkiohjelmat, Tursas ei luo siirtoja pelaajalle vaan pelaajan tekemän siirron seurauksena muodostuvia pelitilanteita. Uusia pelitiloja luodessa luodaan lista kaikista pseudolaillisista siirroista, joita peliasemassa voidaan tehdä. Pseudolailliset siirrot eivät ota kantaa, jättääkö tehty siirto pelaajan shakkiin.

Pelinappuloiden siirrot luodaan seuraavasti. Kuningattaren, lähetin ja tornin siirrot luodaan samalla tavalla. Nämä pelinappulat liikkuvat samalla tavalla, ainoastaan siirtojen suunta muuttuu. Siirtojen muodostamisesta vastaava shakkimoottorin funktio alkaa tarkastaa ruutu kerrallaan pelinappulasta yhteen suuntaan, ja luo siirtoja vastaavia Move0x88-tietueita jokaista siirtoa vastaan, joka päätyy tyhjään ruutuun. Liikkeiden tarkistus päätyy joko omaan tai vastustajan pelinappulaan tai jos seuraava tarkasteltava indeksi sattuisi pelilaudan ulkopuolelle.

Toinen siirtotyyppi on ratsun siirto. Siirtogeneraattori tarkistaa ennalta määrätyt ruudut nykyisen pelinappulan sijainnin ympäriltä, jotka ovat pelilaudalla ja jossa ei ole omaa pelinappulaa ja luo näitä siirtoja vastaavat siirto-tietueet.

Kolmantena ovat kuninkaan siirrot. Siirtogeneraattori etenee aluksi kuten ratsukin, tarkistaen ennalta määrätyt ruudut kuninkaan ympäriltä, joihin se voi siirtyä. Sen lisäksi, jos linnoittamisen mahdollisuus on vielä olemassa, tarkistetaan linnoitussiirron mahdollisuus.

Viimeisenä on hankalin siirrettävä eli sotilas. Sotilaan epäsäännöllinen liikkumistapa tekee siitä hankalan mallinnettavan. Tursas tarkistaa sotilaan siirtoja muodostaessa, onko se sotilaan alkuaseman rivillä. Jos on, sotilas voi liikkua yhden tai kahden siirron verran eteenpäin, jos edessä ovat tyhjät ruudut. Lisäksi tarkistetaan, onko sotilaan etusivuilla vastustajan pelinappuloita, jolloin sotilaalle voidaan luoda lyöntisiirto kyseiseen ruutuun. Jos sotilaan siirto päättyy pelilaudan päätyyn, korotetaan sotilas pelaajan valitsemaksi upseeriksi, oletuksena kuningattareksi. Sotilaan lyöntisiirtoja tarkistaessa, Tursas lukee State-tietueen pelilaudan vektorista ohestalyönnin arvon. Tämä arvo kertoo, onko ohestalyönti mahdollista. Arvo on joko -1, jos ohestalyönti ei ole mahdollinen tai se on pelilaudan indeksilukuarvo. Jos sotilas voi suorittaa lyönnin ohestalyönnin osoittamaan vektorin alkioon, lisätään sitä vastaava siirto mahdollisiin sotilaan siirtoihin.

Shakkitilanteiden tarkistuksessa tarkistetaan, ettei kuninkaan sisältämä indeksi ole vastustajan pelinappulan uhkaama. Uhkatarkistus suoritetaan käänteisellä logiikalla siirtojen muodostamiseen nähden. Halutusta indeksistä lähdetään tarkastelemaan, onko vastustajan liukuvaan pelinappulaan suoraa linjaa, tai onko indeksin vieressä sotilasta tai uhkaavaa ratsua. Uhkaavan kuninkaan tarkistus etenee monimutkaisemmin. Siinä tarkastetaan ensin, onko yhden ruudun päässä vastustajan kuningasta, ja jos on, niin se siirretään uhan alaiseen ruutuun. Tämän jälkeen tarkistetaan jääkö vastustajan kuningas pelaajan uhkaamaksi. Näin varmistetaan, ettei kuningas voi jättää itseään shakkiin.

4.4 Pelitilanteen muodostus

Uuden pelitilanteen muodostus suoritetaan jokaisen shakin siirron aikana. Pelitilanteen muodostus suorittaa tarkistukset, onko kyseinen pelitilanne sallittu sääntöjen mukaan.

```
(defn- update-state [state move]
  (when-not (nil? State)
    (-> state
      (update-move move)
      (update-half-moves move)
      (update-board move)
      (update-castling move)
      (update-en-passant move)
      update-full-move
      update-player-check
      update-opponent-check
      update-turn)))
```

Uuden pelitilan muodostus tallentaa ensin annetun siirron tiedot pelilaudan vektoriin ja päivittää puolisiirtojen määrän. Pelilaudan päivitys suorittaa pelinappuloiden liikkeet pelilaudalla ja päivittää mustien ja valkoisten pelinappuloiden listaa State0x88 -tietueessa. Linnoittamisen päivitys tarkastaa, onko kyseessä kuninkaan siirto, tornin siirto tai tornin lyönti. Tiedon perusteella se päivittää linnoittamista kuvaavaa arvoa. Ohestalyönnin päivitys tarkistaa, oliko edeltävä siirto sotilaan siirto ja liikkuiko sotilas kaksi siirtoa eteenpäin. Tuplaliikkumisen tapauksessa tarkastus jatkaa tarkistamalla, jääkö sotilaan siirto vastustajan sotilaan viereen sallien näin ohestalyönnin. Tämän perusteella ohestalyönniksi tallennetaan joko -1, jos se ei ole mahdollista tai liikkuvan sotilaan taakse jäävän ruudun indeksin arvo. Kokonaisten siirtojen määrää kasvatetaan, jos kyseessä oli mustan pelaajan siirto. Pelaajan shakkipäivitys tarkistaa, ettei pelaajan tekemä siirto jätä pelitilaa shakkiin. Jos näin käy, funktio palauttaa tyhjän arvon ja loput pelin tarkistuksista ohitetaan. Vastustajan shakinpäivitys tarkistaa vastaavasti jättääkö pelaajan siirto vastustajan shakkiin ja tallentaa tämän tiedon pelilaudan vektoriin. Tällä saadaan pieni tehokkuusetu, sillä pelin loppuehtoja tarkistaessa tätä arvoa luetaan monta kertaa ja tallentamalla arvo pelitilaa luodessa se voidaan lukea ilman uudelleen laskemista. Lopuksi vaihdetaan pelaajan vuoroa ja palautetaan uusi pelitila.

4.5 Siirtojen haku

Tursas hakee parhaan siirron käyttäen alfabeta-algoritmia. Tursaan käyttämä algoritmin toteutus pohjautuu Peter Norvigin esittämään versioon [18], mutta sitä on muokattu funktionaalisempaan tyyliin.

```
(defn alpha-beta [state alpha beta depth]
  (let [f (fn [states best-state ac depth]
    (if (empty? states)
      [ac best-state]
      (let [[val _] (alpha-beta (first states)
                                (- beta) (- ac) depth)
            value (int (- val))]
        (cond (>= ac beta) [ac best-state]
              (> value ac) (recur (rest states) (first states) value depth)
              :else (recur (rest states) best-state ac depth))))))]
  (cond (game-end? state) [(game-score state) nil]
        (zero? depth) [(evaluate state) nil]
        :else (let [children (legal-states state)]
                (if (empty? children)
                  (f nil nil (evaluate state) 0)
                  (f children (first children) alpha (dec depth)))))))
```

Algoritmille annetaan pelitilanne, josta se lähtee luomaan pelipuuta, alfa- ja beta-arvot rajaamaan hakutuloksia sekä hakusyvyys, johon algoritmin suorittaminen loppuu, jos peli ei muuten ole päättynyt ennen sen saavuttamista. Algoritmin suoritus alkaa tarkastamalla annetun pelitilanteen tila.

```
(cond (game-end? state) [(game-score state) nil]
      (zero? depth) [(evaluate state) nil]
      :else (let [children (legal-states state)]
              (if (empty? children)
                (f nil nil (evaluate state) 0)
                (f children (first children) alpha (dec depth)))))))
```

Jos peli lopussa, palautetaan pelituloksen arvo. Jos peli ei päättynyt, tarkastetaan onko päädytty maksimihakusyvyyteen, jolloin depth-arvo on nolla. Siinä tapauksessa palautetaan heuristinen arvio annetulle pelitilanteelle. Muutoin algoritmi muodostaa listan kaikista mahdollisista yhden siirron päässä olevista pelitilanteista, ja sitoo sen muuttujaan children. Ennen apufunktiokutsua tarkistetaan, onko pelitilanteella mahdollisia jatkotilanteita. Jos jatkotilanteita ei ole, kutsutaan apufunktiota tyhjillä arvoilla, jotta se palauttaa kyseisen pelitilanteen heuristisen arvioinnin vektorimuodossa. Aina apufunktiota kutsuessa hakupuun syvyyttä vähennetään yhdellä.

```
(fn [states best-state ac depth]
   (if (empty? states)
     [ac best-state]
     (let [[val _] (alpha-beta (first states)
                               (- beta) (- ac) depth)
           value (int (- val))]
       (cond (>= ac beta) [ac best-state]
             (> value ac) (recur (rest states) (first states) value depth)
             :else (recur (rest states) best-state ac depth)))))
```

Apufunktio saa parametreiksi listan pelitilanteista, joista se valitsee parhaimman perustuen heuristisen arviointifunktion arvoon. Se saa myös tällä hetkellä parhaimman pelitilan erikseen, parhaimman mahdollisen arvon suuruuden sekä hakusyvyyden.

Ensimmäiseksi apufunktio tarkistaa, onko kaikki pelitilanteet käyty läpi. Jos pelitilanteet on käyty läpi, on states-muuttuja tyhjä ja algoritmi palauttaa kaksialkioisen vektorin. Vektorin ensimmäisenä alkiona on parhaimman mahdollisen saavutettavissa olevan arvon suuruus ja seuraavana pelitilanne, josta kyseiseen arvoon päästään. Jos mahdollisia pelitilanteita ei ole, arvioidaan kyseinen pelitilanne ja palautetaan tulos. Muussa tapauksessa algoritmi jatkaa pelitilanteen tarkastelua kutsumalla itseään rekursiivisesti, kääntäen alfa ja beta-arvot pelaajan vaihdon takia. Algoritmi jatkaa alipuun tarkastelua, kunnes saavutettava arvo on suurempi tai yhtä suuri kuin beta-arvo. Muuten se päivittää saavutettavaa arvoa ja parhaan pelitilan, jos alipuun arvoksi saadaan isompi kuin saavutettavasta arvosta. Algoritmin arvo palautetaan kahden alkion vektorissa, jotta paras pelitilanne, että sen arvo saadaan palautettua.

Tursas sisältää myös minimax-algoritmin. Tursas muodostaa pelipuun funktionaalisen ohjelmoinnin tapaan. Se käyttää hyväkseen Clojuren laiskoja sekvenssejä muodostaakseen koko pelipuun annetusta pelitilanteesta. Muuten algoritmi on pilkottu pieniin osiin, joista yksi suorittaa pelipuun arvioinnin ja toinen suorittaa karsinnan annettuun syvyyteen. Hiljaisuushaku on myös toteutettu minimax-algoritmilla. Minimax-algoritmi ei ole käytössä, sillä se on erittäin hidas algoritmi, se on jäljellä mallina ja tarkoitus olisi muokata siitä funktionaalisen ohjelmoinnin mukainen versio alfabeta-algoritmista.

4.6 Aseman arviointi

Pelitilanteen arviointi on yksi shakkimoottorin tärkeimmistä funktioista. Sen perusteella tekoäly suorittaa siirtonsa. Se tarkoitus on selvittää, kuinka hyvä pelitilanne on, mutta samalla se tulisi tehdä mahdollisemman tehokkaasti, sillä arviointifunktiota kutsutaan useasti pelipuuta läpikäydessä. Hidas arviointifunktio voi pilata muuten hyvän shakkimoottorin. Täydellistä arviointifunktiota ei ole, vaan shakin jokaiselle asemalle voidaan antaa heuristinen arvo. Tämä arvo kuvaa, kuinka edullinen asema on kulloinkin vuorossa olevan pelaajan kannalta.

Tursaan arviointifunktio pisteyttää pelitilanteen materiaalisen tasapainon ja pelinappuloiden sijaintien mukaan. Tursas käyttää materiaalisessa arvioinnissa Shannonin esittelemiä pelinappuloiden arvoja, mutta kertoo ne kymmenellä. Pelinappuloiden arvot ovat siis seuraavia: sotilas 10, lähetti ja ratsu 30, torni 50 ja kuningatar 90 pistettä. Näin shakkimoottorissa voidaan antaa 1/10 sotilaan arvoisia pisteitä pysyen laskuissa edelleen kokonaisluvuissa. Tällä hetkellä Tursas antaa tarkimmillaan puolen sotilaan arvoisia pisteitä tietyille asemille, jotka eivät vastaa koko sotilaan menetystä. Pisteytys jättää näin tarkentamisen varaa, jos sille tulee tarvetta.

Pelinappuloiden sijainti vaikuttaa myös pelaajan aseman edullisuuteen. Pelaajan kannalta edulliset asemat saavat positiivisia arvoja. Vastaavasti haitalliset asemat saavat negatiivisen arvon. Arvo vaihtelee -15 ja +15 välillä. Näin hyvä asema on puolentoista sotilaan arvoinen. Se palkitsee pelilaudan keskellä sijaitsevien sotilaita, sillä pelilaudan keskustan hallitseminen on tärkeää. Ratsuille annetaan negatiivisia arvoja pelilaudan reunoilla olemisesta. Reunalla olevan ratsun liikkuminen on rajoittunut, joten se voi jäädä siellä jumiin. Samoin ratsulle annetaan positiivisia arvoja pelilaudan keskustan lähistöllä, sillä siellä siitä on eniten hyötyä. Lähetti käyttäytyy hieman samoin kuin ratsu. Kuninkaan tapauksissa aseman pisteytys riippuu pelitilanteesta. Alkupelissä kuninkaan on kannattavampaa olla omalla rivillään ja linnoittautua. Loppupelissä, kun pelinappuloita ei ole paljoa jäljellä, kuningas jää helposti mattiin nurkkiin ja omien pelinappuloiden taakse. Tällöin kuninkaan kannattaa hakeutua keskemmälle pelilautaa, jossa se voi välttää matti-tilanteita helpommin. Pelin päättymiseen johtavat pelitilat käsitellään erikseen. Niiden arvoksi tulee Matti-tilanteessa suuri negatiivinen arvo kuninkaan menetyksestä johtuen ja tasapelissä 0.

5 Yhteenveto

Nyt on karkeasti ottaen käyty läpi tietokoneshakin toteuttaminen funktionaalisella ohjelmoinnilla. Tietokoneshakki koostuu pelilaudan tietorakenteesta, pelipuun muodostuksesta, arviointi-funktiosta sekä haku-algoritmista. Tämän mallintaminen funktionaalisella ohjelmoinnilla onnistuu jakamalla yllä olevat vaiheet pieniin funktioihin ja yhdistämällä ne kokonaisuudeksi.

Funktionaalinen ohjelmointi oli minulle pintapuolisesti tuttu ennen opinnäytetyön aloittamista. Olin lukenut aiheen kirjallisuutta, mutta en ollut kokeillut käytännössä toteuttaa isompaa projektia. Funktionaalinen ohjelmointityyli oli hankala hahmottaa aluksi. Se vaati täysin uuden lähestymiskannan opettelun ja se poikkeaa tyystin muusta ohjelmointitavasta. Yleisesti jokainen shakkiohjelman funktioista kävi läpi monta eri versioita ennen lopulliseen versioonsa päätymistä. Projektin edetessä joko oppi uuden asian tai keksi uuden lähestymiskannan ongelmaan. Onneksi funktionaalisen ohjelmoinnin luonne auttoi tässä. Koska yksittäiset funktiot eivät vaikuta ulkopuoliseen ohjelmaan, oli ohjelman keskeltä helppo korvata funktioita uusilla versioilla.

Projektin suurin haaste oli sen yleinen rakenne ja toimintaan saattaminen. Shakkitietokoneiden erilaisiin toteutuksiin löytyi paljon materiaalia, mutta mikään niistä ei käsitellyt asiaa funktionaalisen ohjelmoinnin kautta. Projekti edistyi perehtymällä ensin imperatiiviseen malliin shakkitietokoneen rakenteesta ja tämän jälkeen soveltaen tätä funktionaalisempaan suuntaan. Haasteita projektissa on ollut riittävästi, mutta projektin kautta on opittu paljon uutta.

Näin jälkikäteen opinnäytetyön olisi voinut rajata pienemmäksi kokonaisuudeksi. Sekä tietokoneshakista, että funktionaalisesta ohjelmoinnista, olisi riittänyt aihetta opinnäytetyöksi. Näin aiheeseen olisi voinut tutustua hieman pintaa syvemmälle. Jouduin karsimaan paljon asioita sekä tietokoneshakin että funktionaalisen ohjelmoinnin puolelta, jotta sain oleelliset asiat mahtumaan selkeästi paperille.

Yllätyin varsin positiivisesti funktionaalisen ohjelmoinnin tarjoamista eduista. Funktionaalisella ohjelmoinnilla ohjelmoinnin abstraktiotason sai nostettua

korkeammalle, jolloin pystyy keskittymään mitä haluaa, ei miten haluaa ratkaista kulloisenkin ongelman. Näin siis funktionaalisessa ohjelmoinnissa paljon asioita jätetään kääntäjän huoleksi. Suurin osa ohjelmoinnista tapahtui Lisp-tulkin kanssa, kokeillen saada jotain yksittäistä funktiota toimimaan. Kun funktion oli saanut toimimaan halutulla tavalla, sen lisäsi muun lähdekoodin sekaan. Samoin yksittäisten ohjelmaosien testaus oli helpompaa, kun kyseisen funktion pystyi nappaamaan tulkkiin, jossa sen toimintaa pystyi tarkastelemaan interaktiivisesti.

Lisp-ohjelmoinnin dynaamisella luonteella oli myös haittansa. Lähinnä sen tyypittömyys haittasi projektin läpivientiä, sillä käännösaikana virheitä ei käydä läpi vaan ongelmat yleensä sattuivat ajon aikana. Näin niiden huomaaminen viivästyi ja aikaa kului vikojen etsimiseen. Tämän olisi voinut välttää hyvien testitapausten kirjoittamisella, johon en käyttänyt projektin aikana aikaa. Toinen tapa ratkaista tämä ongelma olisi ollut ohjelmointikielen vaihto esimerkiksi Haskell-ohjelmointikieleen. Siinä jokainen funktio ja tietojäsen on tyypitetty ja kääntäjä tarkistaa jokaisen funktiokutsun, että sen tyyppi on oikein.

Moniin shakkitekoälyn kohtiin jäi vielä huomattavasti parantamisen varaa. Esimerkiksi Tursaan käyttämä alfabeta-haku voisi hyödyntää Clojuren laiskoja sekvenssejä. Samoin algoritmin eri vaiheet voitaisiin erottaa erillisiksi funktioiksi, kuten samassa lähdekooditiedostossa määritelty minimax-algoritmi on toteutettu. Samoin algoritmiin voisi liittää rauhallisuushaun, jottei se kärsisi niin paljon horisonttivaikutuksesta. Nämä kuitenkin vaatisivat siirtojen muodostamisen uudelleen kirjoittamista, jotta se saataisiin toimimaan käytännöllisellä nopeudella.

Toinen optimoinnin aihe olisi useasti kutsuttujen funktioiden kutsujen ja tulosten tallentaminen muistiin memoisaatiolla. Memoisaatiosta voisi hyödyntää esimerkiksi pelitilanteen arvioinnissa.

Kaikin puolin olen tyytyväinen projektin lopputulokseen. Opin paljon funktionaalisesta ohjelmoinnista ja aion jatkaa aiheen opiskelua.

Lähteet

1 Alan Turing. 1953. Kappale 'Digital computers applied to games' teoksessa Faster than thought. Verkkodokumentti. <http://www.turingarchive.org/browse.php/B/7>. Luettu 19.9.2010.

2 Claude E. Shannon. 1953. XXII. Programming a Computer for Playing Chess. Tekstidokumentti. <http://www.pi.infn.it/~carosi/chess/shannon.txt>. Luettu 20.9.2010.

3 Stuart Russell, Peter Norvig. 2010. Artificial Intelligence - A Modern Approach, Third Edition. Pearson Education.

4 John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. PDF-dokumentti. <http://www-formal.stanford.edu/jmc/recursive.html>. Luettu 2.10.2010.

5 David Levy. 2010. Verkkodokumentti. <http://chessprogramming.wikispaces.com/David+Levy>. Luettu 13.10.2010.

6 James Eade. 2006. Shakinpelaajan käsikirja - Alkuasemasta mattiin 2. painos. Karisto Oy.

7 Steven J. Edwards. Standard: Portable Game Notation Specification and Implementation Guide. Verkkodokumentti. <http://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm>. Luettu 2.2.2011.

8 Tim Mann, H.G. Muller. 2009. Chess Engine Communication Protocol. Verkkodokumentti. <http://www.gnu.org/software/xboard/engine-intf.html>. Luettu 25.9.2010.

9 Rudolf Huber, Stefan Meyer-Kahlen. Huhtikuu 2006. Description of the universal chess interface (UCI). Zip-pakattu tekstidokumentti. <http://download.shredderchess.com/div/uci.zip>. Luettu 10.10.2010.

10 Bruce Molendar, 24.1.2003. 0x88 Move Generation. Verkkodokumentti. <http://web.archive.org/web/20070716111804/www.brucemo.com/compchess/programming/0x88.htm>. Luettu 20.9 2010.

11 Bitboards. 2010. Verkkodokumentti. <http://chessprogramming.wikispaces.com/Bitboards>. Luettu 3.11.2010.

12 Glickman, Mark E., and Jones, Albyn C. 1999. Rating the chess rating system. PDF-dokumentti. <http://www.glicko.net/research/chance.pdf>. Luettu 6.1.2011.

13 Achim Jung. 2004. A short introduction to the Lambda Calculus. PDF-dokumentti. <http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf>. Luettu 10.2.2011.

14 John Hughes. 1990. Why Functional Programming Matters. PDF-dokumentti. <http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf>. Luettu 27.11.2010.

15 OpenJDK Project Lambdas. 9.2.2011. Verkkodokumentti. <http://openjdk.java.net/projects/lambda/>. Luettu 26.1.2011.

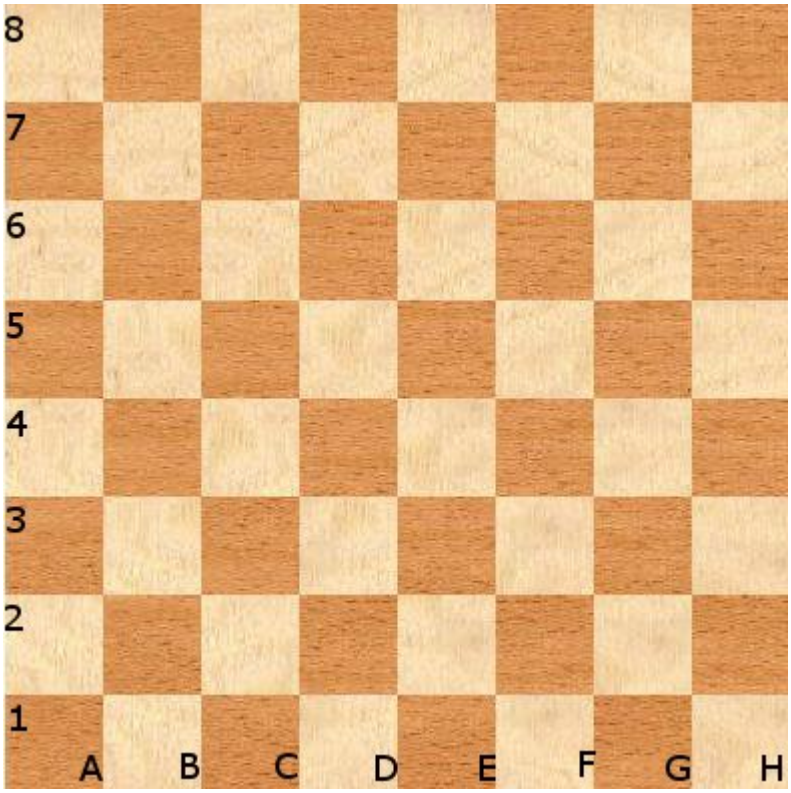
16 Børge Svingen. 2006. When Lisp is faster than C. PDF-dokumentti <http://www.cs.bham.ac.uk/~wbl/biblio/gecco2006/docs/p957.pdf>. Luettu 13.1.2011.

17 Rich Hickey. 2009. Clojure. Verkkodokumentti. <http://clojure.blogspot.com/2009/05/clojure-10.html>. Luettu 19.11.2010.

18 Peter Norvig. 1992. Paradigms of Artificial Intelligence Programming - Case Studies in Common Lisp. Morgan Kaufmann Publishing.

Shakin säännöt

Shakin säännöt ovat varsin yksinkertaiset, joten ne on helppo oppia. Sääntöihin on alla kuvattujen perussääntöjen lisäksi olemassa lisäyksiä, joita käytetään shakkiturnauksissa, mutta ne eivät ole olennaisia opinnäytetyön kannalta.



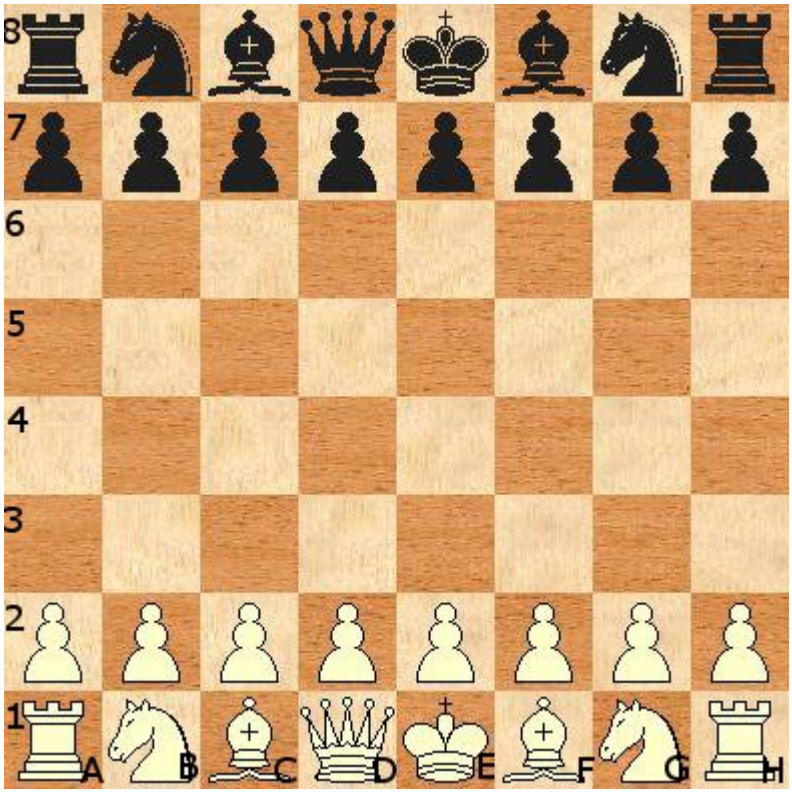
Kuvio 5. Shakkilauta

Shakkilauta on 64 ruutua käsittävä lauta. Lauta jaetaan vaakariveihin ja pystyriveihin. Jokaiselle shakkilaudan ruudulle on oma koordinaatti. Koordinaatit määräytyvät ruudun pysty- ja vaakarivin mukaan. Laudan pystyrivit on numeroitu alhaalta katsoen 1 - 8 ja vaakarivit ovat merkitty kirjaimilla vasemmalta oikealle a - h. Shakkilaudan ruudut ovat joko mustia tai valkoisia. Shakkilauta tulee asettaa siten, että ruutu koordinaatissa h1, eli oikeassa alareunassa, on valkoinen.

Shakissa on kaksi pelaajaa, musta ja valkoinen. Pelin aloittaa aina valkoinen pelaaja. Pelissä pelaajat siirtävät pelinappuloitaan vuorotelleen pelilaudalla tavoitteenaan lyödä vastapuolen kuningas. Kun pelaajan kuningas on vastapuolen pelinappulan uhkaamana, tilanne on shakissa. Tällöin ainut laillinen siirto on lyödä uhkaava nappi,

estää sen uhka kuninkaalle tai siirtää oma kuningas suojaan. Pelaaja ei saa tehdä siirtoa, jossa hänen oma kuninkaansa jäisi uhatuksi. Tilannetta, jossa vastapuolella ei ole yhtään laillista siirtoa estää kuninkaansa menetystä, kutsutaan ”matiksi”. Matin tehnyt pelaaja voittanut pelin. Jos peli päätyy tilanteeseen, jossa kumpikaan pelaaja ei voi tehdä mattia tai on kulunut yli 50 vuoroa ilman sotilaan siirtoa tai lyöntiä, on seurauksena tasapeli, patti. Peli päätyy myös silloin tasapeliin, jos pelaaja ei ole shakissa, mutta ei voi myöskään tehdä yhtään siirtoa joutumatta shakkiin.

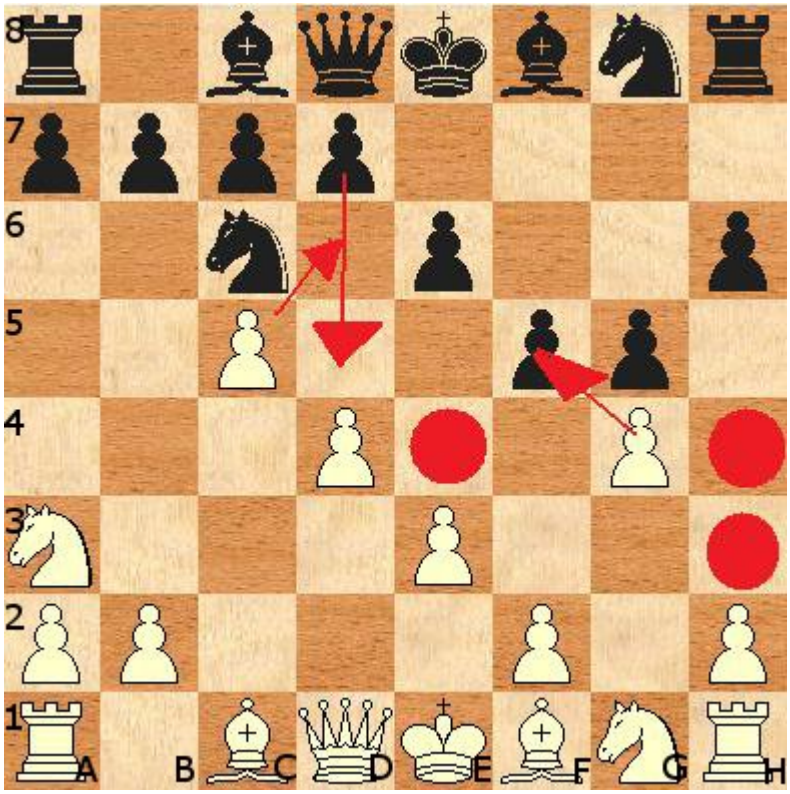
Shakin alkuasetelmassa molemmilla puolilla on 16 nappulaa: kahdeksan sotilasta, kaksi tornia, kaksi ratsua, kaksi lähettiä sekä kuningas ja kuningatar. Pelinappulat ovat aseteltu lähtöasemiinsa niiden ollessa alla olevan kuvan mukaisesti pelilaudalla.



Kuvio 6. Pelinappuloiden alkuasetelma

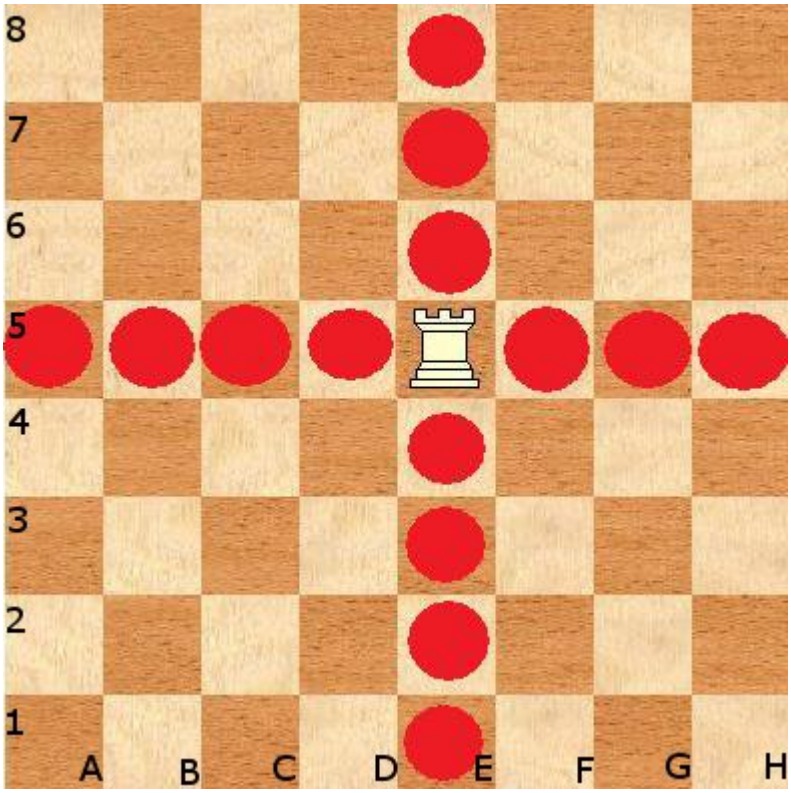
Pelaaja ei saa siirtää nappuloitaan ruutuihin, joissa on jo oma nappula. Jos siirto tapahtuu ruutuun jossa on vastapuolen pelinappula, se lyödään ja poistetaan laudalta siirron seurauksena.

Sotilas on pelin yleisin pelinappula. Se voi liikkua yhden ruudun eteenpäin, jos edessä oleva ruutu on tyhjä. Sotilas ei voi lyödä liikkumalla suoraan eteenpäin. Se voi lyödä vastustajan nappuloita liikkumalla viistosti eteenpäin yhden ruudun. Alkutilanteesta sotilas voi liikkua kaksi askelta kerrallaan. Kun se liikkuu kaksi ruutua alkuasetelmastaan ja päättää siirtonsa vastustajan sotilaan viereen, on vastustajan mahdollista suorittaa ohestalyönti. Ohestalyönti voi tapahtua vain heti sotilaan kahden ruudun siirron jälkeen. Jos tilaisuutta ei käytä hyödyksi, on se hukattu. Vastustaja voi tällöin lyödä kyseisen sotilaan liikuttamalla oman sotilaan sen taakse jäämään tilaan ikään kuin lyöden sen, aivan kuin sotilas olisi vain liikkunut vain yhden ruudun. Sotilas voidaan myös korottaa, jos se pääsee vastustajan päätyyn. Tällöin sotilasnappula voidaan vaihtaa torniksi, ratsuksi, lähetiksi tai kuningattareksi. Tavanomaisesti korotus tapahtuu kuningattareksi, mutta harvoissa tilanteissa jokin muu nappula on tarpeellinen. Esimerkiksi tilanne, jossa kuningattareksi korottaminen johtaisi tilanteen päätymisen pattiin.



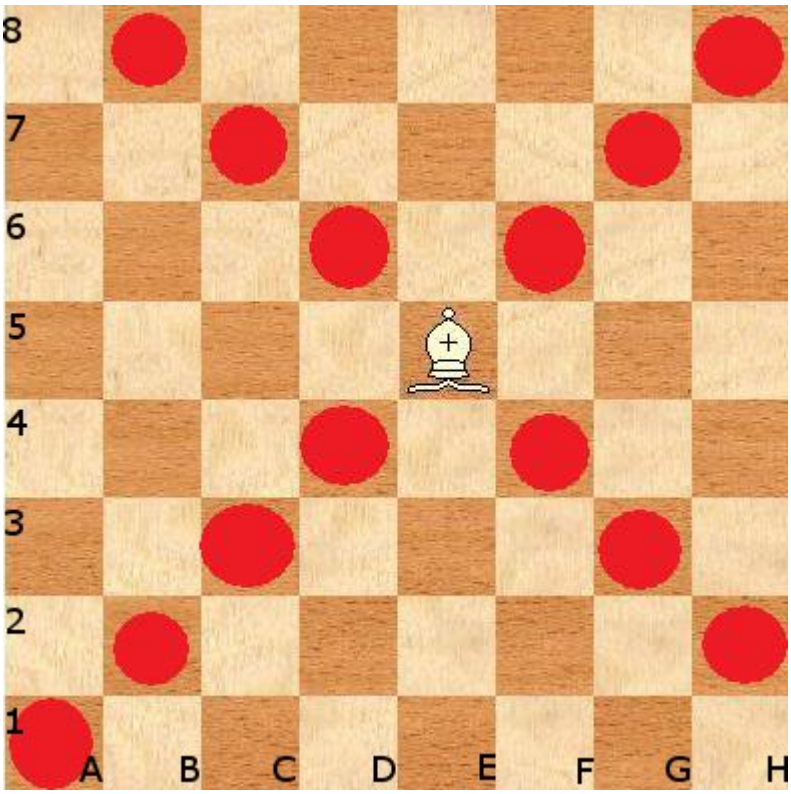
Kuvio 7. Sotilaan liikkeet

Torni voi liikkua vain vaaka- tai pystysuoraan pelilaudalla kuinka monta ruutua tahansa. Se ei saa liikkua omien pelinappuloiden yli. Torni kaappaa liikkumalla vastustajan pelinappulan sisältämään ruutuun. Se osallistuu linnoittamiseen kuninkaan kanssa.



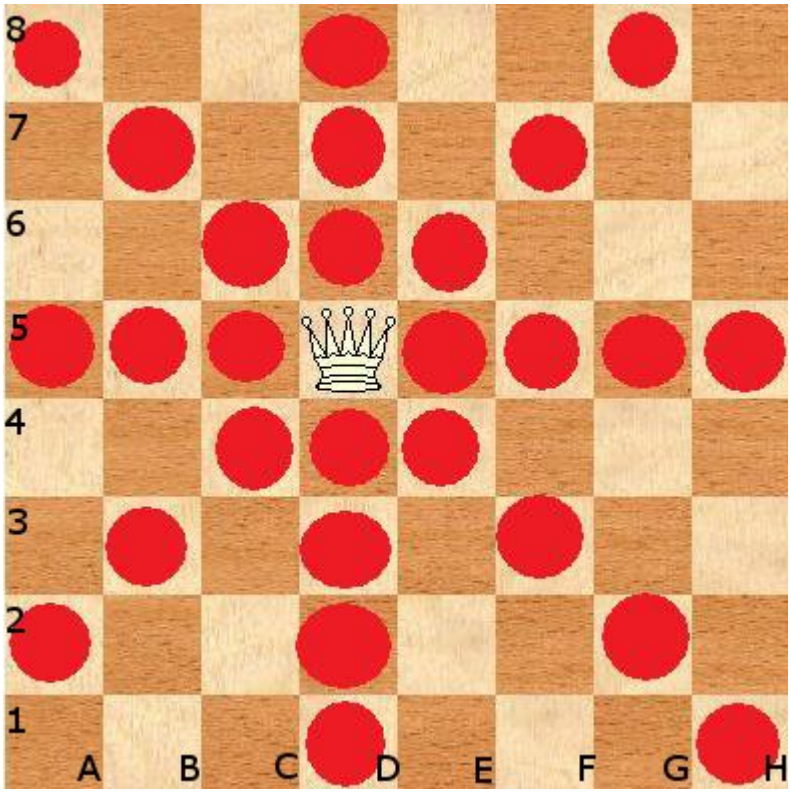
Kuvio 8. Tornin liikkeet

Lähetti liikkuu kuten torni, mutta vain viistosti laudalla. Lähetti voi kulkea vain oman värisiä ruutuja pitkin. Alkuasetelmassa valkoisella ruudulla oleva lähetti voi kulkea vain valkoisia ruutuja pitkin koko pelin ajan.



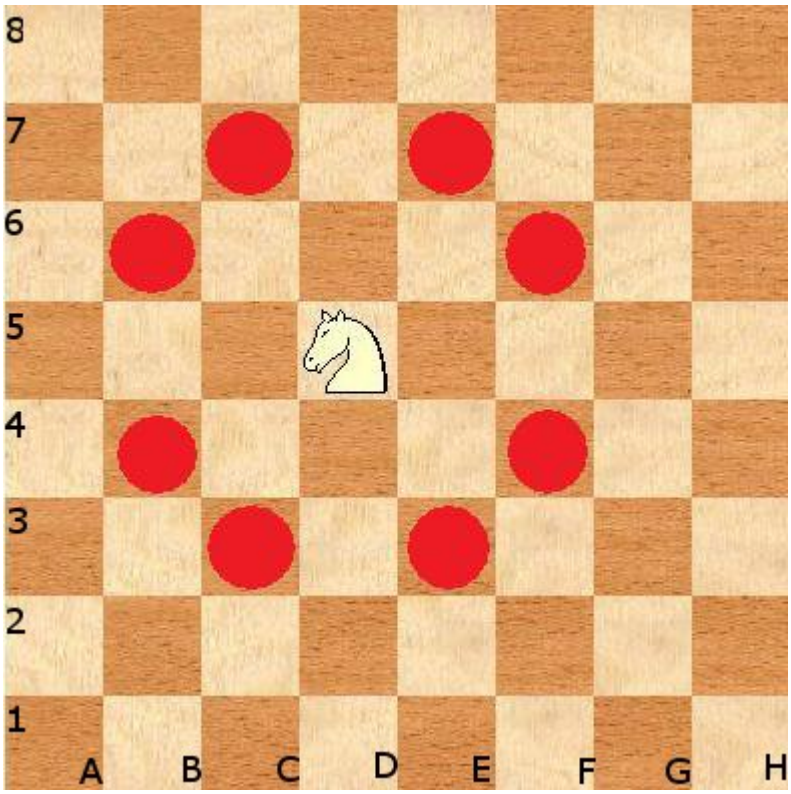
Kuvio 9. Lähetin liikkeet

Kuningatar on shakin voimakkain pelinappula. Se voi liikkua joko vaaka- tai pystysuoraan, kuten torni, tai viistosti kuten lähetti.



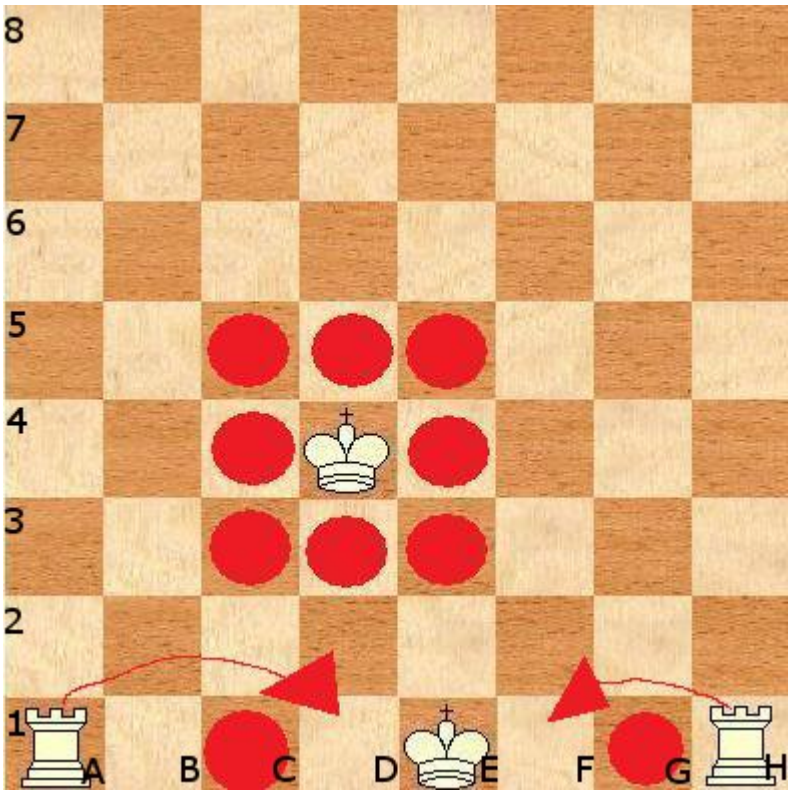
Kuvio 10. Kuningattaren liikkeet

Ratsu on ainut pelinappula, joka voi kulkea muiden pelinappuloiden yli. Ratsu liikkuu kaksi ruutua vaaka- tai pystysuorassa ja yhden ruudun jommalle kummalle sivulle liikuttuun suuntaan nähden.



Kuvio 11. Ratsun liikkeet

Kuningas voi liikkua yhden ruudun mihin tahansa suuntaan, poikkeuksena linnoittaminen tornin kanssa. Kuninkaan suorittama siirto ei saa päättyä shakkiin. Linnoittaminen on kuninkaan suorittama siirto, jossa se voi liikkua kaksi ruutua sivusuunnassa. Linnoittaminen on mahdollista, kun kuningas ja linnoitukseen osallistuva torni ei ole liikkunut pelin aikana ja näiden nappuloiden välissä ei ole yhtään muuta pelinappulaa. Myöskään ruutu, jossa kuningas on, jonka kautta se kulkee tai mihin se päätyy, ei saa olla uhattuna vastustajan puolesta. Linnoituksessa kuningas liikkuu kaksi ruutua jommalle kummalle puolella ja vastaavan puolen torni liikkuu kuninkaan yli ja ottaa paikan kuninkaan vierestä.



Kuvio 12. Kuninkaan liikkeet