OHJ-2100 OHJELMISTOTIETEEN PERUSTYÖKALUJA

5 op

http://www.cs.tut.fi/kurssit/OHJ-2100/

Antti Valmari

Tampereen teknillinen korkeakoulu
Ohjelmistotekniikan laitos
PL 553, 33101 TAMPERE

huone TF 209
puh. 3115 4321
sähköposti Antti.Valmari@tut.fi
seittisivu http://www.cs.tut.fi/~ava/

13.1.2011

1 JOHDANTO

Kurssin tavoitteena on

> *opettaa niitä matemaattisia menetelmiä ja ajattelutapaa, joista on eniten hyötyä ohjelmistotyössä*

- ohjelmistotyö = vaatimusten tunnistaminen, määrittely, suunnittelu, toteutus, testaus, ylläpito, …

Ohjelmistotyö on ollut kriisissä melkein niin kauan kuin ohjelmia on tehty

- ohjelmistohankkeet myöhästelevät ja ylittävät kustannusarvionsa pahasti
- ohjelmat ovat epäluotettavia
- “Software Hall of Shame” 1992–2005
  - IEEE Spectrum Sept. 2005
  - useita satojen miljoonien dollareiden möhläyksiä
- Pentium liukulukuvirhe 1995: 475 miljoonaa $

Vaatimustaso on jatkanut nousua

- ohjelmistojen sovellusalat ovat laajentuneet
  - langattomat, internet-sovellukset, pelit, …
- ohjelmistojen koot ovat kasvaneet rajusti
- ryhmätyö, jopa monikansallinen
- suorituksen rinnakkaisuus
- työmäärä ei kasva lineaarisesti ohjelman koon mukana
  - eräs tutkimus: työn määrä = $\Theta($ rivien määrä^{1,43} $)$
  $\Rightarrow$ kun koko viisinkertaistuu, työ kymmenkertaistuu

Ohjelmistotyön menetelmät ovat muuttuneet

- kielten kehitys
  - helppolukuinen, kattava *Algol 60 report* 1960: 15 s.
  - hyvä Pascal-kirja n. 1975: < 200 s.
  - Kernighan, Ritchie C 1978: ≈ 230 s.
  - Stroustrup C++ 1998: > 900 s.

⇒ ohjelmointikielen osaaminen vaatii nykyisin paljon enemmän kuin vielä vuonna 1990

- korkeammalla tasolla ohjelmistosuunnittelijan on nykyisin hallittava
  - oliomenetelmiä
  - UML-notaatiota
  - asiakas-palvelin-ohjelmointia
  - …
- miten vuonna 2010? 2020?

Siis tarvitaan uudentyyppistä osaamista

- monella päättäjällä on väärä käsitys ohjelmistotyön luonteesta
  - “elektroniikkainsinööri + Fortran-kurssi”

⇒ lyhytnäköisiä mielipiteitä ja päätöksiä

Useimmat tekniikan alat hyödyntävät tehokkaasti matemaattisia teorioita

- ovat yleensä alkaneet käsityötaitona ja perimätietona
- vähitellen on kehitetty teorioita tärkeiden ilmiöiden ymmärtämiseksi
  - esim. Maxwellin yhtälöt sähkömagnetismissa
- suunnittelun helpottamiseksi on kehitetty erilaisia laskenta- yms. menetelmiä
  - esim. Laplace- ja Fourier-muunnokset

- nykyisin menetelmiä on toteutettu tietokoneohjelmina
  - jopa niin, että käyttäjän ei tarvitse ymmärtää menetelmän toimintaa kovin syvällisesti

nikkarointi → tiede → automatisointi

Matematiikan nykytilanne ohjelmistotyössä

- ohjelmistotyön uskotaan vaativan matemaattista lahjakkuutta ja koulutusta
- kuitenkin todellisuudessa ohjelmistoala käyttää matematiikkaa varsin vähän
  - esim. Lethbridgen selvitys 1998 / 2000
    http://www.site.uottawa.ca/%7etcl/edrel/
    (pienehkö ⇒ vain suuntaa antava)
- havainto

Perinteisestä insinöörimatematiikasta on niukasti hyötyä ohjelmistotyössä.

Miksi perinteinen insinöörimatematiikka ei juurikaan auta ohjelmistotyössä, vaikka se on toiminut loistavasti monessa muussa tehtävässä?

- perinteisissä insinööritieteissä voi usein käyttää likiarvoja ja varmuuskertoimia
  - pannaan paksumpi tukipalkki
  - ohjelmistotekniikassa sama onnistuu harvoin
- taustalla vaikuttava syy: perinteiset insinööritieteet käsittelevät jatkuvan maailman ilmiöitä, ohjelmistotekniikan kohteet ovat diskreettejä
- ehkä vielä olennaisempi näkökulma: perinteiset insinööritieteet käsittelevät aineellisia kohteita

*Ihmiskunta ei ole tehnyt **isoja täsmällisiä aineettomia** kohteita ennen kuin ohjelmistot paisuivat isoiksi.*

- vielä yksi näkökulma
  - fysikaaliset luonnonlait asettavat rajoja sille miten ainetta voidaan muokata
  - ⇒ ideat eivät voi räjähtää kovin monimutkaisiksi kovin nopeasti, toisin kuin ohjelmistotekniikassa
- väärä käsitys ohjelmistotyön luonteesta
  - ⇒ sekoitetaan sovellusalan ja ohjelmistotyön tarpeet
- “yleisen matemaattisen kypsyyden” ajatus ei ole toiminut

Toinen havainto

> *Diskreetti matematiikka on pärjännyt vain keskinkertaisesti, vaikka sen uskotaan olevan ohjemoinnin matematiikkaa.*

- syitä
  - vääriä aiheita (esim. kombinatoriikka)
  - opetetaan liiaksi puhtaan matematiikan kaltaisesti, irrallaan sovelluksista
  - rutiinikoodari tarvitsee vähemmän kuin strateginen tuotearkkitehtuurisuunnittelija

Merkittävimmät keinot vähentää ohjelmistovirheitä ohjelmisto*tuotannon* asiantuntijan mukaan

- lähde: B. Boehm & V. R. Basili: *Software Defect Reduction Top 10 List*. IEEE Computer January 2001
  - http://www.cebase.org/

1. Ohjelmisto-ongelman löytäminen ja korjaaminen asiakkaalle lähettämisen jälkeen maksaa usein 100-kertaisesti verrattuna samaan tehtynä määrittely- ja suunnitteluvaiheessa.
2. Ohjelmistoprojektien voimavaroista 40 tai 50 % menee vältettävissä olevaan hukkatyöhön.

6. Vertaiskatselmointi paljastaa 60 % vioista.
   – eri tutkimuksissa 31 … 93 %
8. Henkilökohtaiset kurinalaiset toimintatavat vähentävät virheiden syntyä jopa 75 %.
   – useiden tutkimusten mukaan toiset tekevät virheitä 10-kertaisella tiheydellä toisiin verrattuna
   – kovalla koulutuksella on mahdollista pienentää omaa virhetiheyttään 10-kertaisesti

Mitä matematiikkaa ohjelmistotyö tarvitsee?

- uskon, että matematiikka voisi olla ohjelmistotyölle parhaiten avuksi tarjoamalla työkaluja
  – abstraktioiden tunnistamiseen ja muotoiluun
  – abstraktioista päättelemiseen
- tietokone on saivartelija ja organisaatiokin saattaa olla, mutta järjestelmät tulee tehdä ihmisille

⇒ *Ohjelmistoammattilaisen on rakennettava siltaa asiakkaan inhimillisen maailman ja koneen “yliloogisen” maailman välille.*

⇒ hallittava *molemmat* maailmat

Abstraktioiden tunnistaminen

- ohjelmistotyön pääpaino ei aikoihin ole ollut jokaiselle tuttujen asioiden koodaamisessa ohjelmointikielelle
- suuri osa työstä on erilaisten abstraktioiden muodostamista ja käsittelyä
  – käyttäjän tarpeet: esimerkeistä yleiseen
  – ohjelman osien väliset rajapinnat — niitä on paljon
  – tietovarastot ja tiedostomuodot
  – ohjelmakomponenttikirjastot
  – kansainväliset standardit esim. tietoliikenteessä
  – …

- vaatimusmäärittelykin vaatii hyvää abstraktion tajua
  - tilaajien esittämät toiveet ovat hajanaisia, puutteellisia ja ristiriitaisiakin

  ⇒ yleistäminen ja yhdistely jää ohjelmistoammattilaisten tehtäväksi
- esimerkki: mitä teksturiohjelman leikepöydälle tulee jäädä, jos heti leikkauskomennon jälkeen annetaan komento “peruuta”?
- esimerkki: sähköpostiviestien säilytysaika on 3 kk
  - lomailevan opiskelijan vai tietosuojavaltuutetun näkökanta?
- tilaajat eivät usein edes huomaa sanovansa toisin kuin tarkoittavat
  - ohjelmistotekniikan projektityön loppuraportista: “Asiakkaan löytämät viat ovat johtuneet siitä, että asiakas ei ole tajunnut mitä on pyytänyt.”
- moni yksityiskohta tuntuu saivartelulta siihen asti, kunnes se aiheuttaa virhetoiminnon

Abstraktion muotoilu

- abstraktio on esitettävä siten, että kohderyhmä ymmärtää sen, ja vieläpä oikein
- esitystavan pitää olla riittävän täsmällinen
  - muutoin osapuolet ymmärtävät jonkin tärkeän yksityiskohdan eri tavalla

  ⇒ osat eivät lopulta pelaa kunnolla yhteen
- määritelmän täytyy esittää haluttu käsite, eikä vain jotakin samalta vaikuttavaa
- kuitenkin täytyy osata jättää asioita poiskin, jotta määritelmä ei paisuisi valtavaksi

⇒ on opittava tunnistamaan olennainen

Abstrahoinnin luonnottomuus

- osata tehdä ja osata kertoa miten jokin tehdään ovat aivan eri asioita
  - vrt. polkupyörällä ajo
  - (myös kuunnella sujuvasti on eri asia $\Rightarrow$ laskarit)
- ihminen ajattelee luonnostaan kokonaisuutta ja usein muuttaa viestin järkeväksi, jopa huomaamattaan
  - “se on siinä siellä hyllyllä sen toisen vieressä”
- ihminen ei ole kovin hyvä esittämään asioita pilkuntarkasti
  - monet kokevat sen vastenmielisenä saivarteluna
- ikävä kyllä tietokone tekee juuri niin kuin käsketään, vaikka se ei olisi järkevää
- itse asiassa ihminenkin saattaa tehdä yhtä typeriä ja raivostuttavia tekoja kuin tietokoneet, jos hän ei ymmärrä tarpeeksi hyvin kokonaisuutta, jossa toimii
  - vrt. isot inhimilliset organisaatiot
  - eräs byrokratian selitys?
- isoissa ohjelmistohankkeissa on paljon abstraktioita, joita ei ymmärretä tarpeeksi hyvin tarpeeksi aikaisin

Kunnollinen abstrahointi ei silti ole mahdotonta

- harjoittelemalla ja tekemällä oppii
- ei ole kohtuutonta toivoa, että yliopistotasoisen tutkinnon suorittanut osaa muodostaa ja analysoida käsiterakennelmia omalla alallaan
- tietysti täytyy myös ymmärtää sovellusalue kunnolla
  - muuten saattaa tulla muodollisesti oikeita määritelmiä, joiden sisällössä ei ole järjen häivää

Lisää aihepiiristä:

- “Matematiikan tarve ohjelmistotyössä”, Arkhimedes 2/2001 ss. 18–22, http://www.cs.tut.fi/~ava/kirjoitelmia/Arkhimedes01.html
- “Kun ohjelmistotuotanto ei riitä”, Tietojenkäsittelytiede, kesäkuu 2000, ss. 10–14, http://www.cs.tut.fi/~ava/kirjoitelmia/kunoteiriita.html

Vastaväitteitä

- teollisuudessa ei käytetä tällaisia menetelmiä
  - kukaan ei tiedä mitä käytetään 20 v. päästä
  - yliopistojen pitää yrittää luoda uutta
  - vain rutiinimenetelmiä $\Rightarrow$ vain rutiinituotteita
- matemaattinen täsmällisyys on liian hankalaa ja kallista
  - monet virheetkin ovat todella kalliita
  - sitä ei pidä käyttää kaikkialla, vaan vain siellä ja siinä määrin, missä siitä on riittävästi hyötyä
  - jos ei osaa menetelmiä, ei myöskään tunnista niiden soveltamiskohteita
  - huipputuotteen jujun keksiminen vaatii huippuosaajan

    Jotta osaisi soveltaa sopivan epätäsmällisesti, on osattava soveltaa hyvin täsmällisesti.
- muita parempi osaaminen tuo kilpailuetua

Kurssin teemat

1. Täsmällinen määritteleminen

- ohjelman tilasta puhuminen logiikalla
- matemaattisen määrittelemisen perusidea ja tekniikoita

- sovelluksia:
  - ohjelman katselmointi
  - pysyväisväittämät (eli assertiot) toimivuuden tarkastajina
  - ohjelmanpätkän, algoritmin tms. määrittely
  - täsmällinen määrittely yleensä
- vanhaa asiaa, mutta yritetään
  - ottaa uusi, ohjelmistoläheinen näkökulma
  - korostaa määrittelmisen käytännön taitoja

2. Lausekkeiden, kielten ja automaattien teoria

- mitä asioita ja miten voi esittää rakenteeltaan yksinkertaisilla merkkijonoilla
- millä ehdoilla ja miten rakenteeltaan yksinkertaisia merkkijonoja voi tehokkaasti käsitellä koneella
- sovelluksia:
  - ohjelmointikielet (sekä ohjelmoijan että kääntäjän tekijän näkökulmasta)
  - rakenteellisen tiedon esittäminen, kuten nuottikirjoitus
- vanhin ja ehkä hienoin esimerkki loistavasti onnistuneesta tietojenkäsittelyteorian soveltamisesta

Kurssin näkökulma on ohjelmoijan ja spesifioijan

- ei matemaatikon, loogikon eikä tietojenkäsittelyteoreetikon

$\Rightarrow$ moni asia esitetään toisin kuin matematiikan kursseilla

2 LOGIIKKA JA OHJELMAN MÄÄRITTELY

Tässä luvussa

- kerrataan propositio- ja predikaattilogiikkaa
  - prujussa on paljon aineistoa, mutta luennoilla se käydään läpi valikoiden
- opetellaan puhumaan ohjelman tilojen ominaisuuksista tilapredikaateilla
- kootaan ohjelman spesifikaatio tilapredikaateista
- pohditaan, miten tilapredikaatin ja spesifikaation saa sanomaan oikeat asiat

Vaikka sovellusalueena on ohjelman tila, monet esitetyistä asioista pätevät laajemminkin

Kirjallisuutta

- luvun sisältö ei vastaa suoraan mitään kirjaa, mutta seuraavista opuksista saattaa olla hyötyä:
  - R. C. Backhouse: *Program Construction: Calculating Implementations from Specifications*, Wiley 2003
  - D. Gries: *The Science of Programming*, Springer-Verlag 1981
- seuraavassa kirjassa matematiikkaa opetetaan poikkeavalla, tietojenkäsittelijöille sopivalla tavalla:
  - D. Gries, F. B. Schneider: *A Logical Approach to Discrete Math*, Springer-Verlag 1993

2.1 Ohjelman tila

Tässä alaluvussa

- tarkastellaan ohjelman tilan käsitettä
- perustellaan, miksi siitä on järkevää puhua logiikan ja joukko-opin avulla

Määritelmä

ohjelman tila = muuttujien arvot + suorituksen sijainti

Esimerkkejä: alkuarvot, ohjelma ja sen kaikki tilat

- aluksi $x = y = 0$

```
1:  x := 1
2:  y := 2
3:
```

suoritus	x	y
1	0	0
2	1	0
3	1	2

- aluksi $x = 10$
  - $\lfloor x \rfloor$ on suurin kokonaisluku, joka on enintään x

```
1:  while x > 1 do
2:      x := ⌊ x / 2 ⌋
3:  endwhile
4:
```

suoritus	1	2	3	1	2	3	1	2	3	1	4
x	10	10	5	5	5	2	2	2	1	1	1

- aluksi $i = -5$ ja $A[1] = A[2] = A[3] = 0$

```
1: for i := 1 to 3 do
2:     A[i] := 2·i
3: endfor
4:
```

suoritus	1	2	3	1	2	3	1	2	3	1	4
i	–5	1	1	1	2	2	2	3	3	3	??
$A[1]$	0	0	2	2	2	2	2	2	2	2	2
$A[2]$	0	0	0	0	0	4	4	4	4	4	4
$A[3]$	0	0	0	0	0	0	0	0	6	6	6

Tilapredikaatit

- kaikkien muuttujien arvojen luetteleminen on usein työlästä ja tarpeetonta
- haluamme puhua tilojen ominaisuuksista määrittelemättä kaikkien muuttujien arvoja

$\Rightarrow$ otamme käyttöön *tilapredikaatteja*

- sanomme, että lauseke on *looginen lauseke*, jos se on tarkoitettu tulkittavaksi väittämäksi, joka pitää tai ei pidä paikkaansa
- määritelmä

  Tilapredikaatti = ohjelman muuttujien arvoista puhuva looginen lauseke.

- esimerkkejä
  - $x > 0$
  - $y = 2 \cdot x + 1$
  - $\forall\, i\, ;\, 1 \le i \le n\colon A[i] = -1 \vee A[i] \ge x$

Tilapredikaatit ja totuusarvot

- useimmissa ohjelmointikielissä on jokin tapa esittää “kyllä” ja “ei”
  - tarvitaan mm. **if**-lauseen ehdossa
- esimerkkejä
  - Pascal: tyyppi Boolean: **false** = ei, **true** = kyllä
  - C: `int` 0 = ei, muut `int` ovat kyllä
  - Lisp: Nil = ei, muut arvot = kyllä
- ohjelmointikielen lauseketta sanotaan *totuusarvoiseksi*, jos sen tulos on joko “kyllä” tai “ei”
  - C: tulos “kyllä” = 1
  - Lisp: tulos “kyllä” = “T”
- ohjelmointikielen totuusarvoinen lauseke **ei ole** tilapredikaatti
  - totuusarvoinen lauseke on ohjelma*ssa*
  - tilapredikaatti puhuu ohjelma*sta*, mutta on itse sen ulkopuolella

  $\Rightarrow$ ne asuvat eri maailmoissa
- esimerkki

```
1:  löytyi := false; i := 1
2:  while i ≤ n ∧ ¬ löytyi do
3:      if A[i] = avain then löytyi := true
4:      else i := i + 1
5:      endif
6:  endwhile
7:
```

  - rivillä 2 “$i \le n \wedge \neg$ *löytyi*” on totuusarvoinen lauseke
  - “$i \le n \wedge$ (*löytyi* = **false**)” on tilapredikaatti, joka pitää paikkansa rivin 3 alussa, mutta ei pidä rivin 7 alussa

- totuusarvoinen lauseke on haluttaessa helppo ja luonteva *tulkita* tilapredikaatiksi
  - lauseke e tilapredikaatiksi tulkittuna tarkoittaa samaa kuin tilapredikaatti "e = kyllä"

  $\Rightarrow$ kyllä ~ pitää paikkansa, ja
  ei ~ ei pidä paikkaansa
- esimerkki: tilapredikaatin "$i \leq n \wedge (löytyi$ = **false**$)$" saa nyt lyhentää muotoon "$i \leq n \wedge \neg\, löytyi$"

$\Rightarrow$ totuusarvoisen lausekkeen ja tilapredikaatin välillä
- on periaatteellinen ero
- useimmiten erosta ei tarvitse välittää, vaan totuusarvoisen lausekkeen saa tulkita tilapredikaatin erikoistapaukseksi

Tilapredikaatit ja suorituksen sijainti

- tavallisesti tilapredikaatit eivät puhu suorituksen sijainnista

$\Rightarrow$ tilapredikaatin paikkansapitävyys riippuu siitä, missä kohti ohjelmaa se tarkastetaan

- tärkeitä tulkintakohtia
  - "alussa" = juuri ennen ohjelman suorituksen aloittamista
  - "lopussa" = ohjelman lopetettua normaalisti (ei siis esim. ohjelman kaaduttua)
  - "rivillä n" = **joka kerta** kun ohjelma on valmis suorittamaan rivin n
- lopussa tulkittava tilapredikaatti sanoo jotain ohjelman tuloksista
  - tulokset jäävät joihinkin muuttujiin

- alussa tulkittava tilapredikaatti tulkitaan eri tavalla kuin muualla esiintyvät!
  - esittää muuttujissa oleville arvoille asetettua *alkuehtoa*
  - ko. muuttujat sisältävät ko. ohjelmanpätkän syötteet
- usein on kätevää kirjoittaa tilapredikaatti siihen kohti ohjelmaa, jossa se halutaan tulkita
  - aaltosulkeisiin “{” ja “}” (myöhemmin myös sulkeisiin “⟨” ja “⟩”)
  - tällöin ei välttämättä tarvita rivinumeroita
- esimerkki

  if $x < 0$ **then**
  { $x < 0$ }
  $x := -x$
  { $x > 0$ }
  endif { $x \geq 0$ }

  - jos **then**-haaraa ei suoriteta, niin lopussa $x \geq 0$
  - jos suoritetaan, niin lopussa $x > 0$

  $\Rightarrow$ joka tapauksessa lopussa $x \geq 0$

- esimerkki
  - kaikki muuttujat kokonaislukutyyppiä
  - käsitellään taulukkoa $A[1\ldots n]$

```
 1:    { n ≥ 1 }
 2:    a := 1; y := n
 3:    { a = 1 ∧ y = n ≥ 1 }  (* seuraus: a ≤ y *)
 4:    while a < y do
 5:        { a < y }
 6:        v := ⌊ (a+y) / 2 ⌋
 7:        { a ≤ v < y }
 8:        if A[v] < avain then
 9:            { v < y }
10:            a := v+1
11:            { a ≤ y }
12:        else
13:            { a ≤ v }
14:            y := v
15:            { a ≤ y }
16:        endif
17:        { a ≤ y }
18:    endwhile
19:    { a = y }
```

  - syötteitä koskeva vaatimus: $n \geq 1$
  - rivi 7 voidaan päätellä rivistä 5 ja keskiarvon ja alaspäin pyöristyksen ominaisuuksista
  - rivit 9 ja 13 seuraavat rivistä 7
  - rivi 17 seuraa riveistä 11 ja 15
  - rivillä 19 $a \geq y$ rivin 4 ehdon vuoksi; lisäksi $a \leq y$ rivien 3 ja 17 vuoksi; näistä yhdessä seuraa $a = y$

Pysyväisväittämät

- *pysyväisväittämä* eli *assertio* on ohjelman sisään tarkastuksen vuoksi kirjoitettu lause, joka kaataa ohjelman, jos annettu ehto ei päde
- pysyväisväittämistä on **erittäin** paljon hyötyä varsinkin monimutkaisten tietorakenteiden ja algoritmien virheiden jäljittämisessä
- C++:

```
#include <cassert>
   ...
assert( i >= 0 && 1 < n );
```

- AV:
  - parametrit lasketaan joka tapauksessa
  - ⇒ ei tehokkain mahdollinen ratkaisu, mutta …
  - yksinkertainen ja informatiivinen
  - näin pientä tehohävikkiä kannattaa murehtia vasta kun ohjelmointitaidot ovat hyvät

```
inline void tarkasta(
   bool ehto, const char *v0,
   int i1 = 0, const char *v1 = 0,
   int i2 = 0, const char *v2 = 0
){
   if( !ehto ){
      std::cout.flush();  // kyllä, cout!
      std::cerr << "\n??? Ohjelman "
         "sisäinen virhe:\n??? " << v0;
      if( v1 ){ std::cerr << i1 << v1; }
      if( v2 ){ std::cerr << i2 << v2; }
      std::cerr << std::endl; exit( 1 );
   }
}
```

- käyttöesimerkki

```
 1:   tarkasta( n ≥ 1, “liian pieni n” )
 2:   a := 1; y := n
 4:   while a < y do
 6:       v := ⌊ (a+y) / 2 ⌋
 8:       if A[v] < avain then
10:           a := v+1
12:       else
14:           y := v
16:       endif
17:       tarkasta( a ≤ y, “liian iso a ”, a, “ ”, y, “” )
18:   endwhile
19:   tarkasta( a = y, “a ≠ y ”, a, “ ”, y, “” )
```

- jos tilapredikaatin totuusarvo on helppo laskea, siitä saa kätevästi pysyväisväittämän
- monimutkaisemmin laskettavan tilapredikaatin tarkastuksesta voi tehdä aliohjelman, jota kutsutaan silloin tällöin
  - esim. pääsilmukan joka sadannella kierroksella
  - virhettä jäljitettäessä kutsujen tiheyttä voi nostaa
  - lopullisesta ohjelmasta kutsun voi poistaa esim. muuttamalla kommentiksi tai `#ifdef`illä
- esimerkki: keko
  $\forall\, i;\ 2 \le i \le n:\ A[\lfloor i/2 \rfloor] \ge A[i]$
- esimerkki: kaksisuuntainen linkitetty lista
  x^.next^.prev = $x \wedge x$^.prev^.next = x

$\Rightarrow$ tilapredikaattien avulla voi testata ohjelmia

Jatkossa tulemme laskemaan paljon tilapredikaateilla

$\Rightarrow$ kannattaa kerrata predikaateilla laskemisen säännöt eli logiikan perusteet

2.2 Propositiologiikan kertausta

Propositiologiikan kieli koostuu seuraavista osista:

- käyttäjän valitsema joukko *propositiosymboleja*
  - esim. *Sataa*, *Tuulee*, *Paistaa*, *Kenkuttaa*, *Hymyilyttää*
  - usein kun ei ajatella mitään erityistä sovellusta P, Q, R, P_1, P_2, … tai p, q, r, p_1, p_2, …
  - alaluvussa 2.3 propositiosymbolit korvataan sovellusalueen relaatioilla, esim. $A[i] < x$
- *loogiset operaattorit* eli *konnektiivit*, joilla propositiosymboleita yhdistetään lausekkeiksi (eli *kaavoiksi*)
  - "$\wedge$" *konjunktio* eli "ja": $P \wedge Q$
  - "$\vee$" *disjunktio* eli "tai": $P \vee Q$
  - "$\neg$" *negaatio* eli "ei": $\neg P$
  - "$\rightarrow$" *implikaatio* eli "jos … niin": $P \rightarrow Q$
  - "$\leftrightarrow$" *ekvivalenssi* eli "jos ja vain jos": $P \leftrightarrow Q$
  - (merkinnät "$\Rightarrow$" ja "$\Leftrightarrow$" esitellään myöhemmin)
- *sulut* "(" ja ")", joilla ohjataan laskentajärjestystä
- usein mukaan otetaan myös totuusarvovakiot
  - "**false**" tai "**F**" tai "$\perp$" ("epätosi", "vale") ja
  - "**true**" tai "**T**" ("tosi")
- joskus mukaan otetaan myös konnektiivi "**xor**"
  - meille tarpeeton, koska P **xor** Q on sama kuin $\neg(P \leftrightarrow Q)$

Propositiosymboleiden tulkinta

- propositiosymbolit edustavat väittämiä, jotka joko pitävät tai eivät pidä paikkaansa
  - esim. "*Sataa*" tulkittuna TTY:n parkkipaikalla 13.1.2011 klo. 12:00
  - esim. ohjelman tilaa koskeva väittämä "*x-on-0*"

- jos P on propositiosymboli, niin P:n *totuusarvo* on
  - "**true**" eli "**T**", kun P pitää paikkansa
  - "**false**" eli "**F**" eli "$\perp$", kun P ei pidä paikkaansa
- propositiologiikka ei määrää symbolien tulkintaa eikä totuusarvoja
  - se on käyttäjän asia
- propositiologiikan tehtävänä on tarjota keinoja
  - yhdistää väittämiä monimutkaisemmiksi
  - johtaa uusia tosia väittämiä
  - tarkastaa väittämien totuus tai yhtäpitävyys

Kaavat

- propositiosymboleita voi yhdistää konnektiiveilla lausekkeiksi, joita kutsutaan *(hyvin muodostetuiksi) kaavoiksi*
  - esim. $(Sataa \wedge Tuulee) \rightarrow Kenkuttaa$
- kaavojen totuusarvot saadaan osien totuusarvojen funktiona
  - $\neg$**T** = **F** ja $\neg$**F** = **T**
  - $P\ op\ Q$ taulukon mukaan, missä P:n totuusarvo vasemmalla ja Q:n ylhäällä:

$\wedge$	**F**	**T**
F	**F**	**F**
T	**F**	**T**

$\vee$	**F**	**T**
F	**F**	**T**
T	**T**	**T**

$\rightarrow$	**F**	**T**
F	**T**	**T**
T	**F**	**T**

$\leftrightarrow$	**F**	**T**
F	**T**	**F**
T	**F**	**T**

- esimerkiksi jos *Sataa* = **F**, *Tuulee* = **T** ja *Kenkuttaa* = **T**, niin
  - $Sataa \wedge Tuulee$ = **F**
  - $(Sataa \wedge Tuulee) \rightarrow Kenkuttaa$ = **T**
- (myöhemmin pohdimme miksi on päätetty, että **F** $\rightarrow$ **T** = **T**)

- kuten tavallista, sulkujen vähentämiseksi sovimme operaattoreille laskentajärjestyksen
  - onko $S \wedge T \rightarrow K$ tulkittava kuten $(S \wedge T) \rightarrow K$ vai $S \wedge (T \rightarrow K)$?
- tällä kurssilla käytetään alla esitettävää järjestystä
  - melko yleinen
  - järjestys vaihtelee hieman eri kirjoissa
  - $\Rightarrow$ usein on hyvä käyttää “ylimääräisiä” sulkuja, varsinkin “$\rightarrow$”:n ja “$\leftrightarrow$”:n yhteydessä
- operaattorien sitovuusjärjestys:

korkein				matalin
$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$

- ilman sulkuja laskenta etenee seuraavasti:
  - korkeamman sitovuuden omaavat ensin
  - saman sitovuuden omaavat vasemmalta oikealle
- esimerkki:

  $P \rightarrow Q \leftrightarrow \neg P \vee Q \wedge R$

  lasketaan kuten

  $(P \rightarrow Q) \leftrightarrow ((\neg P) \vee (Q \wedge R))$
- tämän alaluvun ajan
  - symbolit φ, η ja γ tarkoittavat kaavoja
  - jos näitä symboleita käytetään osana isompaa kaavaa (esim. $\varphi \wedge \eta$), kaikki tarpeelliset sulut kuvitellaan lisätyiksi
  - siis jos esim. $\varphi = P$ ja $\eta = Q \vee R$, niin $\varphi \wedge \eta = P \wedge (Q \vee R)$ eikä $P \wedge Q \vee R$

Loogiset ja muut totuudet

- usein kaavan totuusarvo riippuu propositiosymbolien tulkinnasta
  - esim. $Sataa \vee Tuulee$

- jos tällaisen kaavan väitetään olevan totta, silloin väitetään jotain sovellusalueesta
- on myös kaavoja, joiden totuusarvo ei riipu propositiosymboleiden totuusarvoista
  ⇒ ei riipu myöskään niiden tulkinnoista
  – esim. $Sataa \vee \neg Sataa$ on aina **T**
  – esim. $Sataa \wedge \neg Sataa$ on aina **F**
- kaava on
  – *tautologia*, jos se on tosi kaikilla tulkinnoilla
  – *ristiriita*, jos se ei ole tosi millään tulkinnalla

- siis esimerkiksi
  – $Sataa \wedge \neg Sataa$ on ristiriita
  – $Sataa \vee Tuulee$ ei ole tautologia eikä ristiriita
- kaava φ on tautologia jos ja vain jos ¬φ on ristiriita
- tautologiaa sanotaan myös *loogiseksi* totuudeksi
  – looginen totuus on voimassa puhtaasti muodollisista syistä
  ⇒ se ei “oikeasti” sano mitään sovellusalueesta

Premissit

- jos kaikkien propositiosymboleiden totuusarvot tiedetään, minkä tahansa kaavan totuusarvo on helppo laskea
- usein niitä ei tiedetä, mutta tiedetään tai oletetaan joukko niiden välisiä riippuvuuksia
- nämä riippuvuudet voidaan esittää joukkona kaavoja
  – *premissit*

- esimerkki:

  $Sataa \rightarrow Kenkuttaa$
  $Tuulee \rightarrow Kenkuttaa$
  $Kenkuttaa \leftrightarrow \neg Hymyilyttää$
  $\neg(Sataa \wedge Paistaa)$

- jatkossa oletamme, että premissejä on äärellinen määrä
  - propositiologiikan voi rakentaa myös sallien äärettömästi premissejä
  - silloin osa alla sanotusta ei pidä paikkaansa

Päätteleminen propositiologiikassa

- premisseistä voi johtaa muita kaavoja tai tarkastaa annettujen kaavojen totuudet suhteessa niihin
  $\Rightarrow$ "lisää" sovellusaluetta koskevaa tietoa
  - esim. $Sataa \vee Tuulee \rightarrow \neg Hymyilyttää$
- seuraava periaate formalisoi "oikean johtopäätöksen" käsitteen propositiologiikassa:

  Kaava φ *seuraa premisseistä* $\varphi_1, \ldots, \varphi_n$
  jos ja vain jos
  $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \varphi$ *on tautologia.*

- tämä periaate yhdistää kaksi tasoa:
  - formaalin logiikan kielen taso: "$\rightarrow$"
  - merkityksen eli semantiikan taso: "seuraa"
- johtopäätöksen φ oikeellisuus voidaan siis periaatteessa tarkastaa kokeilemalla kaikilla propositiosymboleiden arvoyhdistelmillä, tuleeko kaavan $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \varphi$ totuusarvoksi **T**
  - *totuustaulu*menetelmä
  - käytännössä yleensä liian työläs

- kaavojen ja päättelyiden käsin tai koneelliseen johtamiseen ja tarkastamiseen on kehitetty helpompia keinoja
  - resoluutio
  - “natural deduction”
  - …
- käsin laskijalle lienee helpointa käyttää jäljempänä lueteltavia lakeja ja päättelyoperaattoreita

Edellä käytettiin sanaa “formaali”; mitä se tarkoittaa?

- sanakirja: “formal” = muodollinen, kaavamainen, virallinen
- matematiikassa: merkitykseen nojautumaton
  - formaali laskeminen on vain sääntöjen soveltamista kaavoihin, eräänlainen merkkipeli
  - kaavan sisältöä ei ajatella: säännön mukaan saa laskea, vaikka se johtaisi ilmeisen “väärään” tulokseen
  - edes selvästi “oikeaa” laskua ei saa tehdä, jollei löydy sääntöä, joka antaa siihen luvan
- siis lasketaan teorian sisällä ja vain teorian tarjoamin keinoin, unohtaen teorian tulkinta

$\Rightarrow$ jotta vältettäisiin älyttömyydet, säännöstö on suunniteltava huolella

- säännöstö ei yleensä määrittele, mitä sääntöä ja mihin on laskettaessa sovellettava seuraavaksi
  - tilaa luovuudelle
  - vaikea pulma laskennan automatisoinnille

- koska sääntöjä sovelletaan merkitystä ajattelematta, soveltaminen on mekaanista työtä
  - vaatii suurta täsmällisyyttä
  - ihmiselle kurjaa
  - tietokoneelle helppoa
- koska merkitykseen ei saa vedota, sääntöjen on oltava hyvin täsmällisiä ja niitä on noudatettava hyvin täsmällisesti

$\Rightarrow$ **ohjelmistotyön näkökulmasta**

formaali = matemaattisen täsmällinen

Miksi periaate “seuraa” ~ “… $\rightarrow$ … on tautologia” pätee?

- se voidaan perustella tai todistaa tai “todistaa” päteväksi esimerkiksi tarkastelemalla totuusarvoja
- toisaalta jokainen perustelu tai todistus nojaa jollain tavalla operaattorien merkitystä ja päättelysääntöjä koskeviin epäformaaleihin mielikuviin
- viime kädessä kyse on siitä, haluammeko uskoa, että rakentamamme päättelyn formalisointi todella vastaa “oikeita” päättelyn lakeja — mitä tahansa “oikeat” päättelyn lait sitten ovatkin ja mikäli niitä edes on

$\Rightarrow$ kyse on formaalin teorian vertaamisesta ihmisten päissä asuviin mielikuviin

Vertailu “vastaako teoria todellisuutta” on aina pohjimmiltaan epäformaali.

Mistä propositiologiikka sitten saa oikeutuksensa, kun väitettä “formalisointi vastaa todellisuutta” ei voi todistaa?

- se vain jotenkin tuntuu oikealta melkein kaikista niistä, jotka ovat ymmärtäneet sen merkinnät ja säännöt

- se toimii!
  - ts. se on osoittautunut tehokkaaksi ja erittäin luotettavaksi työkaluksi

Miksi on päätetty, että **F** $\rightarrow$ **T** = **T** ?

- mitä mahdollisia totuusarvoyhdistelmiä φ ja η voivat saada, jos η seuraa (sanan intuitiivisessa epäformaalissa merkityksessä) φ:stä?
- kokeillaan esimerkillä (x > 2) $\rightarrow$ (x > 0), x on luku
  - x = 3: **T** $\rightarrow$ **T**
  - x = 1: **F** $\rightarrow$ **T**
  - x = 0: **F** $\rightarrow$ **F**
  - mitenkään ei saada aikaan tilannetta **T** $\rightarrow$ **F**
- yleisesti
  - silloin kun φ pätee, pitää myös η:n päteä
  - silloin kun φ ei päde, ei η:sta väitetä mitään: se saa päteä tai olla pätemättä

$\Rightarrow$ ainoa kielletty tapaus on φ = **T** ja η = **F**

$\Rightarrow$ "$\rightarrow$":n totuustauluksi on päätetty

$\rightarrow$	**F**	**T**
F	**T**	**T**
T	**F**	**T**

- "siitä että sataa seuraa että kenkuttaa" ei lakkaa olemasta pätevä päättelysääntö, vaikka on päiviä, jolloin ei sada, mutta silti kenkuttaa
  - kenkuttaa esimerkiksi siksi että tuulee

  $\Rightarrow$ niinpä ei saa olla **F** $\rightarrow$ **T** = **F**

Päättelyoperaattorit

- loogisia väittämiä voi verrata toisiinsa seuraavilla *päättelyoperaattoreilla*:
  - "$\Rightarrow$" *seuraus*
  - "$\Leftrightarrow$" *yhtäpitävyys*
- päättelyoperaattoreilla esitetään päätelmiä
  $\varphi \Rightarrow \eta$ tai $\varphi \Leftrightarrow \eta$
  tai päättelyketjuja, esimerkiksi
  $\varphi_1 \Leftrightarrow \varphi_2 \Rightarrow \varphi_3 \Leftrightarrow \varphi_4 \Leftrightarrow \varphi_5 \Rightarrow \varphi_6$
- päättelyoperaattorit ovat *konjunktionaalisia*, ja sitovuudeltaan loogisia oper. matalammalla tasolla
  - esim. $P \Leftrightarrow Q \Leftrightarrow R$ tarkoittaa "$P \Leftrightarrow Q$ ja $Q \Leftrightarrow R$" eikä kuten esim. $P \leftrightarrow Q \leftrightarrow R \Leftrightarrow (P \leftrightarrow Q) \leftrightarrow R$
  - itse asiassa merkintä $(P \Leftrightarrow Q) \Leftrightarrow R$ ei tähän asti kerrotun pohjalta ole edes järkevä
- päättelyoperaattoreita käytettäessä usein oletetaan joukko tunnettuja sovellusaluekohtaisia asioita, joita ei toisteta joka kaavassa
  - esim. oletetaan, että kaikki muuttujat ovat kokonaislukuja, ja kokonaislukujen laskulait
  - esim. oletetaan annettu kiinteä joukko kaavoja (*aksioomat*)

"$\Rightarrow$" eli seuraus

- $\varphi \Rightarrow \eta$ tarkoittaa, että jos (yleisoletukset pätevät ja) φ pätee, niin η:kin pätee, mutta ei välttämättä päinvastoin

- käytetään käytännössä merkitsemään “oletan tai tiedän, että φ pätee, mistä päättelen, että η pätee”
  - tarkkaan ottaen merkintä ei edellytä, että η voidaan *päätellä* φ:stä; riittää, että aina kun φ pätee, myös η pätee
  - toisaalta propositiologiikassa ne ovat sama asia
- esimerkkejä:
  - viikonpäivien lait tunnettu:
    tänään on tiistai $\Rightarrow$ huomenna on arki
  - reaalilukujen lait tunnettu ja x reaaliluku:
    $x > 1 \Rightarrow x > 0$
  - ei yleisoletuksia: $P \wedge Q \Rightarrow P \vee Q$
- jos yleisoletuksia ei ole, niin $\varphi \Rightarrow \eta$ on oikea päätelmä jos ja vain jos $\varphi \rightarrow \eta$ on tautologia
- jos yleisoletukset ovat ψ, niin

$\varphi \Rightarrow \eta$	tarkoittaa
$\psi \Rightarrow \varphi \rightarrow \eta$	eli että
$\psi \rightarrow (\varphi \rightarrow \eta)$	on tautologia

- myös $\psi \wedge \varphi \rightarrow \eta$ on tämän kanssa yhtäpitävä kaava
  - tod. harjoitustehtävä
- siis $\varphi \Rightarrow \eta$ on oikein, jos ja vain jos kaikilla yleisoletusten sallimilla propositiosymbolien totuusarvoyhdistelmillä, joilla φ = **T**, myös η = **T**
- $\varphi \Rightarrow \eta$ on *looginen seuraus*, jos se pätee ilman mitään yleisoletuksia (ts. jos $\varphi \rightarrow \eta$ on tautologia)
- jos on päätelty $\varphi \Rightarrow \eta$ (eikä enempää), niin
  - ei saa jatkaa “jos φ ei päde, niin η ei päde” (esim. tänään ei ole tiistai $\Rightarrow$ huomenna ei arki)
  - saa päätellä “jos η ei päde, niin φ ei päde” (esim. huomenna ei ole arki $\Rightarrow$ tänään ei tiistai)

"$\Leftrightarrow$" eli yhtäpitävyys

- $\varphi \Leftrightarrow \eta$ tarkoittaa, että (jos yleisoletukset pätevät, niin) sekä φ että η pätevät, tai kumpikaan ei päde
  - siis tarkoittaa samaa kuin "$\varphi \Rightarrow \eta$ ja $\eta \Rightarrow \varphi$"
  - käytetään merkitsemään
    "korvaan φ:n samanveroisella kaavalla η"
- esimerkkejä:
  - viikonpäivien lait tunnettu:
    tänään on maanantai $\Leftrightarrow$ huomenna on tiistai
  - reaalilukujen lait tunnettu ja x reaaliluku:
    $x > 1 \Leftrightarrow x - 1 > 0$
  - kokonaislukujen lait tunnettu ja x kokonaisluku:
    $x > 1 \Leftrightarrow x \geq 2$
  - ei yleisoletuksia:
    $(P \rightarrow Q) \wedge \neg Q \Leftrightarrow \neg(P \vee Q)$
- $\varphi \Leftrightarrow \eta$ on oikea päätelmä
  - ilman yleisoletuksia, jos ja vain jos $\varphi \leftrightarrow \eta$ on tautologia
  - yleisoletuksilla ψ, jos ja vain jos $\psi \Rightarrow (\varphi \leftrightarrow \eta)$, eli $\psi \rightarrow (\varphi \leftrightarrow \eta)$ on tautologia
- siis $\varphi \Leftrightarrow \eta$ on oikein, jos ja vain jos kaikilla yleisoletusten sallimilla propositiosymbolien totuusarvoyhdistelmillä φ:llä ja η:lla on sama totuusarvo
- jos on päätelty $\varphi \Leftrightarrow \eta$, niin saa päätellä sekä
  "jos φ pätee, niin η pätee"
  että
  "jos φ ei päde, niin η ei päde"

"⇒":n ja "→":n sekä "⇔":n ja "↔":n suhteesta

- päättelyoperaattorit "⇒" ja "⇔" **eivät ole** propositiologiikan operaattoreita, vaan operaattoreita, joiden avulla voidaan
  - puhua propositiologiikan kaavojen keskinäisistä suhteista
  - esittää päättelyketjuja
- esimerkki

$Sataa \rightarrow Kenkuttaa$			2
$Sataa$		vrt.	+ 3
$\Rightarrow$ $Kenkuttaa$			= 5

- vertaa tilapredikaatin ja ohjelmointikielen totuusarvoisen lausekkeen välinen ero
- vertaa
  - tilojen ominaisuuksista puhutaan logiikalla
  - logiikan kaavoista puhutaan päättelyoperaattoreilla
  - päättelyoperaattoreista puhutaan … suomeksi!
- toisaalta, näimme, että jos saa kirjoittaa $\varphi \rightarrow \eta$, niin saa päätellä $\varphi \Rightarrow \eta$, ja päinvastoin
  - vastaavasti "↔":lle ja "⇔":lle

⇒ "→" ja "⇒" sekä "↔" ja "⇔" ilmaisevat olennaisesti saman asian (vaikkakin eri tasoilla)

⇒ silloin kun päättelyn tasoja "formaalisti propositiologiikan sisällä" — "formalismin ulkopuolella" ei tarvitse pitää erillään, voimme samaistaa operaattorit seuraavasti:
  - "→" = "⇒" = implikaatio
  - "↔" = "⇔" = ekvivalenssi
- tällä kurssilla samaistus ei ole sallittu

- muista kuitenkin
  - sitovuuksien erot
  - "$\Rightarrow$" sallii yleisoletuksiin vetoamisen, "$\rightarrow$" ei

Propositiologiikan lakeja

- nämä voi tarkastaa esimerkiksi totuustaulujen avulla
- totuusarvovakioita koskevia lakeja:

  $\mathbf{F} \Rightarrow \varphi$ (valheesta seuraa mitä tahansa)

  $\varphi \Rightarrow \mathbf{T}$ (mistä tahansa seuraa totuus)

  $\neg \mathbf{F} \Leftrightarrow \mathbf{T}$ $\quad$ $\neg \mathbf{T} \Leftrightarrow \mathbf{F}$

  $\varphi \wedge \mathbf{F} \Leftrightarrow \mathbf{F}$ $\quad$ $\varphi \wedge \mathbf{T} \Leftrightarrow \varphi$

  $\varphi \vee \mathbf{F} \Leftrightarrow \varphi$ $\quad$ $\varphi \vee \mathbf{T} \Leftrightarrow \mathbf{T}$

  $\varphi \rightarrow \mathbf{F} \Leftrightarrow \neg\varphi$ $\quad$ $\varphi \rightarrow \mathbf{T} \Leftrightarrow \mathbf{T}$

  $\mathbf{F} \rightarrow \varphi \Leftrightarrow \mathbf{T}$ $\quad$ $\mathbf{T} \rightarrow \varphi \Leftrightarrow \varphi$

  $\varphi \leftrightarrow \mathbf{F} \Leftrightarrow \neg\varphi$ $\quad$ $\varphi \leftrightarrow \mathbf{T} \Leftrightarrow \varphi$

- kaksoiskiellon poisto:

  $\neg\neg\varphi \Leftrightarrow \varphi$

- sekalaista:

  $\varphi \wedge \varphi \Leftrightarrow \varphi$ $\quad$ $\varphi \vee \varphi \Leftrightarrow \varphi$

  $\varphi \wedge \neg\varphi \Leftrightarrow \mathbf{F}$ $\quad$ $\varphi \vee \neg\varphi \Leftrightarrow \mathbf{T}$

  $\varphi \wedge \eta \Rightarrow \varphi$ $\quad$ $\varphi \wedge \eta \Rightarrow \eta$

  $\varphi \Rightarrow \varphi \vee \eta$ $\quad$ $\eta \Rightarrow \varphi \vee \eta$

- vaihdannaisuus:

  $\varphi \wedge \eta \Leftrightarrow \eta \wedge \varphi$ $\quad$ $\varphi \vee \eta \Leftrightarrow \eta \vee \varphi$

  $\varphi \leftrightarrow \eta \Leftrightarrow \eta \leftrightarrow \varphi$

- liitännäisyys:
  $$(\varphi \wedge \eta) \wedge \gamma \quad \Leftrightarrow \quad \varphi \wedge (\eta \wedge \gamma)$$
  $$(\varphi \vee \eta) \vee \gamma \quad \Leftrightarrow \quad \varphi \vee (\eta \vee \gamma)$$
  $$(\varphi \leftrightarrow \eta) \leftrightarrow \gamma \quad \Leftrightarrow \quad \varphi \leftrightarrow (\eta \leftrightarrow \gamma)$$
  - edellä määrittelimme laskujärjestyksen siten, että esim. $P \wedge Q \wedge R \;\Leftrightarrow\; (P \wedge Q) \wedge R$
  - näitten sääntöjen ansiosta myös
    $$P \wedge Q \wedge R \;\Leftrightarrow\; P \wedge (Q \wedge R)$$

  $\Rightarrow$ sulkujen paikoista ei tarvitse välittää “$\wedge$”:n, “$\vee$”:n ja “$\leftrightarrow$”:n tapauksessa
  - huom! kohta näytetään, että “$\rightarrow$”:lle tämä ei päde!
- osittelulait:
  $$\varphi \wedge (\eta \vee \gamma) \;\Leftrightarrow\; (\varphi \wedge \eta) \;\vee\; (\varphi \wedge \gamma)$$
  $$\varphi \vee (\eta \wedge \gamma) \;\Leftrightarrow\; (\varphi \vee \eta) \;\wedge\; (\varphi \vee \gamma)$$
  - näissä on yllättävää, että osittelu toimii molemmin päin
  - vrt. yhteen- ja kertolasku: $a(b+c) = ab+ac$, mutta esim. $1+2\cdot 3 = 7 \neq (1+2)\cdot(1+3) = 12$
- absorptiolakeja:
  $$\varphi \vee (\varphi \wedge \eta) \;\Leftrightarrow\; \varphi$$
  $$\varphi \wedge (\varphi \vee \eta) \;\Leftrightarrow\; \varphi$$
- DeMorganin lait:
  $$\neg(\varphi \wedge \eta) \;\Leftrightarrow\; \neg\varphi \vee \neg\eta$$
  $$\neg(\varphi \vee \eta) \;\Leftrightarrow\; \neg\varphi \wedge \neg\eta$$
- implikaatio- ja ekvivalenssioperaattorin poistolait:
  $$\varphi \rightarrow \eta \;\Leftrightarrow\; \neg\varphi \vee \eta$$
  $$\varphi \leftrightarrow \eta \;\Leftrightarrow\; (\varphi \rightarrow \eta) \wedge (\eta \rightarrow \varphi)$$
  $$\varphi \leftrightarrow \eta \;\Leftrightarrow\; \varphi \wedge \eta \vee \neg\varphi \wedge \neg\eta$$

- muita implikaatio- ja ekvivalenssilakeja:
  $\varphi \rightarrow \varphi \;\Leftrightarrow\; \mathbf{T}$
  $\varphi \leftrightarrow \varphi \;\Leftrightarrow\; \mathbf{T}$
  $(\varphi \rightarrow \eta) \wedge (\eta \rightarrow \gamma) \;\Rightarrow\; \varphi \rightarrow \gamma$
  $(\varphi \leftrightarrow \eta) \wedge (\eta \leftrightarrow \gamma) \;\Rightarrow\; \varphi \leftrightarrow \gamma$

Päättelyoperaattoreita koskevia lakeja

- osa näistä vastaa edellä olleita implikaatio- ja ekvivalenssilakeja
- lue
  - “$\varphi \Rightarrow \eta$”: “φ:stä saa päätellä η”
  - “$\varphi \Leftrightarrow \eta$”: “$\varphi$:stä saa päätellä η ja η:sta φ”
- (tarkkaan ottaen
  - “$\varphi \Rightarrow \eta$”: “η seuraa loogisesti φ:stä yleisoletuksilla”
  - “$\varphi \Leftrightarrow \eta$”: “$\varphi$ ja η ovat loogisesti yhtäpitävät yleisoletuksilla”)
- refleksisyys:
  $\varphi \Rightarrow \varphi$ ja $\varphi \Leftrightarrow \varphi$
- “$\Leftrightarrow$”:n symmetrisyys:
  jos $\varphi \Leftrightarrow \eta$ niin $\eta \Leftrightarrow \varphi$
  (“$\Rightarrow$”:lle tämä ei välttämättä päde)
- transitiivisuus:
  jos $\varphi \Rightarrow \eta$ ja $\eta \Rightarrow \gamma$ niin $\varphi \Rightarrow \gamma$
  jos $\varphi \Leftrightarrow \eta$ ja $\eta \Leftrightarrow \gamma$ niin $\varphi \Leftrightarrow \gamma$
- keskinäinen suhde:
  $\varphi \Leftrightarrow \eta$ jos ja vain jos $\varphi \Rightarrow \eta$ ja $\eta \Rightarrow \varphi$

- näiden ansiosta pitkän päättelyketjun saa esittää
  $\varphi_1 \Leftrightarrow \varphi_2 \Leftrightarrow \ldots \Leftrightarrow \varphi_n$
  ja lukea lopputulokseksi $\varphi_1 \Leftrightarrow \varphi_n$, tai
  $\varphi_1 \rightarrowtail_1 \varphi_2 \rightarrowtail_2 \cdots \rightarrowtail_{n-1} \varphi_n$,
  missä kukin "$\rightarrowtail_i$" on "$\Leftrightarrow$" tai "$\Rightarrow$"
  ja lukea lopputulokseksi $\varphi_1 \Rightarrow \varphi_n$
- "$\Leftrightarrow$":n kongruenssiominaisuus:

  Jos $\varphi \Leftrightarrow \eta$ *ja* $\gamma(\varphi)$ *on kaava, jossa* φ *on osana, niin* $\gamma(\varphi) \Leftrightarrow \gamma(\eta)$.
- kongruenssiominaisuus sallii ekvivalenssin soveltamisen kaavan osaan
  - esim. koska $\varphi \rightarrow \eta \;\Leftrightarrow\; \neg\varphi \vee \eta$, niin
    $(P \rightarrow Q) \wedge (Q \rightarrow R) \;\Leftrightarrow\; (\neg P \vee Q) \wedge (\neg Q \vee R)$
- **varoitus!** "$\Rightarrow$":lle kongruenssiominaisuus ei päde!
  - ongelmana ennen kaikkea "$\neg$": voi olla $\varphi \Rightarrow \eta$ ilman että $\neg\varphi \Rightarrow \neg\eta$
  - pohdittavaksi: onko myös "$\wedge$" ongelma? entä "$\rightarrow$"?

Laskeminen propositiologiikassa

- jokaisen kaavan pätevyys voidaan tarkastaa kokeilemalla kattavasti eri totuusarvoilla
- esimerkki: osoitettava $P \vee (P \wedge Q) \;\Leftrightarrow\; P$

P	Q	$P \wedge Q$	$P \vee (P \wedge Q)$
F	**F**	**F**	**F**
F	**T**	**F**	**F**
T	**F**	**F**	**T**
T	**T**	**T**	**T**

  - *totuustaulu*

- tämä on kuitenkin virhealtista ja työlästä
  - n propositiosymbolia $\Rightarrow$ 2^n riviä

  $\Rightarrow$ käytännössä on usein mukavampaa laskea soveltamalla edellä annettuja lakeja
- tärkeämpää on, että totuustaulumenetelmä ei yleisty seuraavaksi käsiteltävään predikaattilogiikkaan

  $\Rightarrow$ lakien avulla laskeminen on opittava viimeistään silloin
- esimerkki: osoitettava "$P \wedge (P \rightarrow Q) \Rightarrow Q$":

  $P \wedge (P \rightarrow Q)$ $\Leftrightarrow$ (implikaation poisto)
  $P \wedge (\neg P \vee Q)$ $\Leftrightarrow$ (osittelulaki)
  $(P \wedge \neg P) \vee (P \wedge Q)$ $\Leftrightarrow$ (2. sekalainen)
  $\mathbf{F} \vee (P \wedge Q)$ $\Leftrightarrow$ (vaihdannaisuus)
  $(P \wedge Q) \vee \mathbf{F}$ $\Leftrightarrow$ (vakiolaki)
  $P \wedge Q$ $\Rightarrow$ (7. sekalainen)
  Q

  siis $P \wedge (P \rightarrow Q) \Rightarrow Q$
- harjoittelun jälkeen laskelma menee rutiinilla:

  $P \wedge (P \rightarrow Q) \Leftrightarrow P \wedge (\neg P \vee Q) \Leftrightarrow$
  $(P \wedge \neg P) \vee (P \wedge Q) \Leftrightarrow P \wedge Q \Rightarrow Q$
- esimerkki: osoitettava $(P \wedge Q) \vee (P \wedge \neg Q) \vee \neg P$ tautologiaksi:

  $(P \wedge Q) \vee (P \wedge \neg Q) \vee \neg P \Leftrightarrow$
  $(P \wedge (Q \vee \neg Q)) \vee \neg P \Leftrightarrow (P \wedge \mathbf{T}) \vee \neg P \Leftrightarrow$
  $P \vee \neg P \Leftrightarrow \mathbf{T}$
- usein laskut sujuvat näppärästi sijoittamalla osalle propositiosymboleista vuorotellen eri arvot, ja soveltamalla tuloksiin lakeja

- esimerkki: osoitettava "$\wedge$":n liitännäisyys
  jos $\gamma \Leftrightarrow \mathbf{F}$, niin
  $(\varphi \wedge \eta) \wedge \gamma \Leftrightarrow (\varphi \wedge \eta) \wedge \mathbf{F} \Leftrightarrow \mathbf{F}$
  $\Leftrightarrow \varphi \wedge \mathbf{F} \Leftrightarrow \varphi \wedge (\eta \wedge \mathbf{F}) \Leftrightarrow \varphi \wedge (\eta \wedge \gamma)$
  (viimeinen rivi alunperin päätelty takaperin)
  jos $\gamma \Leftrightarrow \mathbf{T}$, niin
  $(\varphi \wedge \eta) \wedge \gamma \Leftrightarrow (\varphi \wedge \eta) \wedge \mathbf{T} \Leftrightarrow \varphi \wedge \eta$
  $\Leftrightarrow \varphi \wedge (\eta \wedge \mathbf{T}) \Leftrightarrow \varphi \wedge (\eta \wedge \gamma)$
- joskus kannattaa koettaa löytää propositiosymboleille arvot, joilla kaava *ei* päde
  – erityisesti, kun ei tiedetä, päteekö kaava
  – tai kun todistus ei millään meinaa mennä läpi
- esimerkiksi päteekö $(\varphi \rightarrow \eta) \rightarrow \gamma \Leftrightarrow \varphi \rightarrow (\eta \rightarrow \gamma)$?
  – koska aina $\chi \rightarrow \mathbf{T} \Leftrightarrow \mathbf{T}$, niin heti nähdään, että jos $\gamma \Leftrightarrow \mathbf{T}$, niin molemmat puolet $\Leftrightarrow \mathbf{T}$
  $\Rightarrow$ jäljellä tapaus $\gamma \Leftrightarrow \mathbf{F}$
  – sijoitus $\gamma \Leftrightarrow \mathbf{F}$ ja laki $\varphi \rightarrow \mathbf{F} \Leftrightarrow \neg\varphi$ antavat $\neg(\varphi \rightarrow \eta) \Leftrightarrow \varphi \rightarrow \neg\eta$
  – vasen puoli sievenee
  $\ldots \Leftrightarrow \neg(\neg\varphi \vee \eta) \Leftrightarrow \varphi \wedge \neg\eta$
  – oikea $\ldots \Leftrightarrow \neg\varphi \vee \neg\eta$
  – jos $\varphi \Leftrightarrow \mathbf{T}$, niin kumpikin puoli on $\neg\eta$
  – jos $\varphi \Leftrightarrow \mathbf{F}$, niin vasen on **F** ja oikea **T**
  $\Rightarrow$ ei päde, kun $\varphi \Leftrightarrow \gamma \Leftrightarrow \mathbf{F}$
- tuloksen tarkastus:
  – $(\mathbf{F} \rightarrow \eta) \rightarrow \mathbf{F} \Leftrightarrow \mathbf{T} \rightarrow \mathbf{F} \Leftrightarrow \mathbf{F}$
  – $\mathbf{F} \rightarrow (\eta \rightarrow \mathbf{F}) \Leftrightarrow \mathbf{T}$
  $\Rightarrow$ kyllä, eroa on!
- ei väliä miten lasket, kunhan
  – pääset tulokseen, ja
  – tulos on varmasti oikein!
  $\Rightarrow$ kannattaa tehdä tarkastuksia

Päättelyn lähtökohtien riittävyydestä

- joskus kaavan todistaminen ei millään onnistu, vaikka kaava on “selvästi” tosi
- vikana voi olla, että kaikkia tarpeellisia oletuksia ei ole formalisoitu yleisoletuksiin eikä premisseihin
- jos tarpeellisia oletuksia puuttuu, niin
  - kaavaa ei voi todistaa vääräksi, koska se on “todellisuudessa” tosi
  - kaavaa ei voi todistaa oikeaksi, koska oletusten puitteissa se voisi olla vääräkin

  $\Rightarrow$ kaavan totuusarvo jää vääjäämättä avoimeksi
- tämä on sukua sille, että jokainen kaava on joko tautologia, ristiriita tai tulkinnasta riippuva
  - kaava “lähtökohdat $\rightarrow$ todistettava kaava” on silloin tulkinnasta riippuva
  - sen pitäisi olla tautologia, jotta todistaminen voisi onnistua
  - puuttuvat välttämättömät oletukset sulkevat pois ne tulkinnat, joissa todistettava kaava on epätosi vaikka lähtökohdat ovat todet

  Formaali päättelyjärjestelmä voi käyttää hyväksi vain sitä tietoa, mikä sille on kerrottu!

Esimerkki: olkoon annettu premissit

$$Sataa \rightarrow Kenkuttaa$$
$$Tuulee \rightarrow Kenkuttaa$$
$$Kenkuttaa \leftrightarrow \neg Hymyilyttää$$
$$\neg(Sataa \wedge Paistaa)$$

- voidaan osoittaa $Sataa \rightarrow \neg Hymyilyttää$
- voidaan osoittaa $\neg(Kenkuttaa \wedge Hymyilyttää)$
- ei voida osoittaa $Paistaa \rightarrow Hymyilyttää$
  eikä $\neg(Paistaa \rightarrow Hymyilyttää)$,
  koska premissit sallivat totuusarvot

S	T	P	K	H	$P \rightarrow H$
F	**F**	**T**	**T**	**F**	**F**
F	**F**	**F**	**F**	**T**	**T**

  $\Rightarrow$ annetuilla premisseillä $Paistaa \rightarrow Hymyilyttää$ ei voi todistaa oikeaksi eikä vääräksi

$\Rightarrow$ jos laskut eivät mene läpi, kannattaa tarkistaa, onko niissä otettu huomioon kaikki lähtötiedot

- mitä vahvemmat premissit, sitä useamman kaavan voi osoittaa oikeaksi tai vääräksi
- joskus premisseistä voi todistaa ristiriidan, eli sekä jonkin kaavan φ että sen negaation $\neg\varphi$
  - tällöin premissit ovat sisäisesti ristiriitaiset
  - tällöin voidaan osoittaa
    $\varphi \wedge \neg\varphi \Leftrightarrow \mathbf{F} \Rightarrow \eta$
    eli mitä tahansa!

Siis:

- jos premissit ovat ristiriidattomat, jokaiselle kaavalle φ pätee jokin seuraavista:
  - voidaan osoittaa φ, mutta ei $\neg\varphi$
  - voidaan osoittaa $\neg\varphi$, mutta ei φ
  - ei voida osoittaa φ eikä $\neg\varphi$
- jos premissit ovat ristiriitaiset, jokaiselle kaavalle φ pätee:
  - voidaan osoittaa sekä φ että $\neg\varphi$
- jos premissit ovat ristiriitaiset, niiden kuvaama tilanne ei todellisuudessa voi esiintyä
  - ainakin niin me vakaasti uskomme
  - todellisuus on asia, josta emme koskaan voi olla ihan aivan täysin varmoja!
  - kukaan ei tiedä, onko matematiikka ristiriidaton

2.3 Predikaattilogiikan kertausta

Propositiologiikka kytkeytyy sovelluskohteeseensa propositioiden välityksellä

- se ei katso propositioiden sisärakennetta
- jokaisessa maailman tilassa kukin propositio tuottaa arvon **F** tai **T**

Predikaattilogiikka kytkeytyy sovelluskohteeseensa siellä olevien arvojen, niiden funktioiden ja niiden välisten relaatioiden välityksellä

- propositiosymbolit korvataan *predikaattisymboleilla*
  - ottavat sovellusalueen arvoja ja tuottavat totuusarvoja
  - edustavat sovellusalueen relaatioita
- kieleen lisätään käsitteitä sovellusalueen arvoista ja niiden funktioista puhumiseen
  - tärkein lisäys: *kvanttorit*
- predikaattilogiikka olettaa sovellusalueesta automaattisesti vain yhden asian:

  siellä on ainakin yksi arvo

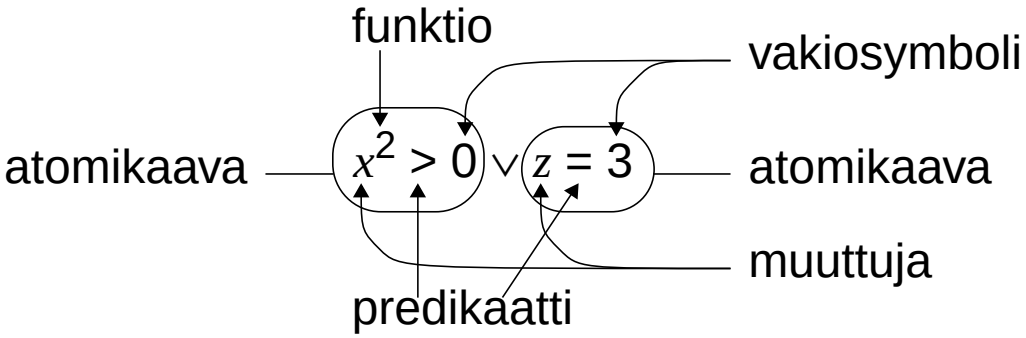
  - muut sovellusalueen ominaisuudet ilmaistaan ottamalla halutut predikaattilogiikan kaavat aksioomiksi
  - (usein ne oletetaan tunnetuiksi ja jätetään mainitsematta)
- sovellusalueen arvoja kutsutaan *ei-loogisiksi* arvoiksi
  - esim. kokonaisluku matematiikassa
  - esim. ohjelman muuttujan x arvo

Predikaattilogiikan kielen osat

- *vakiosymboli* = ei-loogiselle (vakio)arvolle annettu nimi
  - esim. “0”, “π”, “sunnuntai”
- *muuttuja* = symboli, joka edustaa tuntematonta tai tilanteen mukaan vaihtuvaa ei-loogista arvoa
  - esim. “x”, “y”
- *funktiosymboli* = ei-loogisten arvojen väliselle funktiolle annettu nimi
  - esim. “^{2}”: $f(x) = x^2$
    “+”: $f(x,y) = x+y$
    “seur”: seur(maanantai) = tiistai
- usein sanotaan vain “vakio” jne. kun tarkoitetaan “vakiosymboli” jne.
- *termi* = vakiosymboleista, muuttujista ja / tai funktiosymboleista koostuva lauseke, jonka arvo on ei-looginen
  - esim. “0”, “$2 \cdot x + 5$”, “$(x+y)^2$”
- *predikaattisymboli* (eli predikaatti) = väittämälle annettu nimi
  - ottaa parametrikseen 0 tai useampia ei-loogisia arvoja
  - tuottaa totuusarvon
  - esim. “$\leq$”: $P(x,y) \Leftrightarrow x \leq y$
  - predikaatti voidaan tulkita nimeksi funktiolle ei-loogisista arvoista totuusarvoihin
  - (propositiosymboli voidaan tulkita 0-paikkaiseksi predikaatiksi)
- *atomikaava* = lauseke, joka koostuu predikaattisymbolista, jonka argumenteiksi on asetettu termit
  - esim. $x > 0$, $P(0, x^2+2)$

- *kaava* = atomikaava, tai niistä operaattoreilla “$\neg$”, “$\vee$”, “$\wedge$”, “$\rightarrow$”, “$\leftrightarrow$” ja / tai ns. *kvanttoreilla* “$\forall$” ja “$\exists$” koottu lauseke

Esimerkki kaavasta ja sen osista



- termit: x, x^2, 0, z ja 3

Toinen esimerkki

$$\forall\, \varepsilon: \underline{\varepsilon > 0} \rightarrow \exists\, \delta: \underline{\delta > 0} \wedge \forall\, y: \underline{|y-x| < \delta} \rightarrow \underline{|f(y)-f(x)| < \varepsilon}$$

- vakiot: 0
- muuttujat: ε, δ, x, y
- funktiot: $x - y$, $|x|$, $f(x)$
- termit: ε, 0, δ, x, y, $y-x$, $|y-x|$, $f(x)$, $f(y)$, $f(y)-f(x)$ ja $|f(y)-f(x)|$
- predikaatit: > ja <
- atomikaavat alleviivattu

Kvanttorit

- olkoot
  - x ei-loogisia arvoja saava muuttuja
  - $\varphi(x)$ kaava, jossa x (ehkä) esiintyy
  - (kuten kohta täsmennetään, x:n kaikkia esiintymiä ei välttämättä oteta huomioon)

- “∀” kaikki
  - $\forall\, x$: $\varphi(x)$ tarkoittaa, että $\varphi(x)$ pätee, olipa x:n arvo mikä tahansa
  - esim. jos x on reaaliluku, niin $\forall\, x$: $x^2 \geq 0$
- “∃” on olemassa
  - $\exists\, x$: $\varphi(x)$ tarkoittaa, että on olemassa ainakin yksi arvo x, jolla $\varphi(x)$ pätee
  - esim. jos x on reaaliluku, niin $\exists\, x$: $x^2 = 0$ (nimittäin $x = 0$)
- joskus käytetään myös kvanttoria “∃!” (tai “$\exists_1$”) eli “on olemassa täsmälleen yksi”
  - esim. $\exists!\, x$: $2 \cdot x = 4$
  - esim. $\neg\exists!\, x$: $x^2 - 2 \cdot x = 0$
  - esim. $\neg\exists!\, x$: $x^2 = -2$
  - tätä ei tällä kurssilla käytetä syystä, jonka eräs harjoitustehtävä havainnollistaa

Kvanttorin vaikutusalue

- on tarpeen tietää, mille kaavan osalle kvanttori vaikuttaa
  - esim. onko $P \vee \exists\, x\colon Q(x) \rightarrow R$ sama kuin $(P \vee \exists\, x\colon Q(x)) \rightarrow R$ vai $P \vee (\exists\, x\colon Q(x) \rightarrow R)$?
- kirjallisuudessa ei aina anneta selviä sääntöjä
  $\Rightarrow$ sulkuja kannattaa käyttää reilusti!
- tällä kurssilla kvanttorin vaikutusalue ulottuu seuraavaan “⇒”, “⇔” tai kaavan loppuun
  - siis $P \vee \exists\, x\colon Q(x) \rightarrow R \;\Rightarrow\; S$ tulkitaan kuten $P \vee (\exists\, x\colon Q(x) \rightarrow R) \;\Rightarrow\; S$
  - kuten tavallista, suluilla voi ohittaa tämän säännön
    $(P \vee \exists\, x\colon Q(x)) \rightarrow R \;\Rightarrow\; S$

Avoimet ja suljetut kaavat

- muuttujan x esiintymä kaavassa on
  - *sidottu*, jos se on jossakin osakaavassa muotoa $\forall x: \varphi$ tai $\exists x: \varphi$
  - *vapaa* muulloin
  - esim. $\exists y: x+y = 0$ ➠ x on vapaa ja y sidottu
  - esim. $x > 0 \wedge (\exists x: x < 0) \wedge x \neq 1$
    ↑ vapaat esiintymät ↑
- vertaa

```
i = 7;
for( int i = … ){
   … x += A[i]; …
}
cout << i; …
```

- vertaa

```
void tyyt( int A[] ){
   for( int i = … ){ cout << A[i]; }
}
```

- kaava on
  - *suljettu*, jos siinä ei ole vapaita muuttujien esiintymiä
  - *avoin* muulloin
- logiikassa usein irrallinen avoin kaava tulkitaan suljetuksi olettamalla, että kaikki vapaat muuttujat on kvantifioitu “∀”:lla
  - tällöin esim. $x > 0 \vee \exists y: (x < y \wedge y \leq 0)$
    tarkoittaa $\forall x: x > 0 \vee \exists y: (x < y \wedge y \leq 0)$
    ja siis ei päde (jos sovellusalue on reaaliluvut ja “<” jne. tulkitaan sen mukaisesti)
  - tällä kurssilla näin tehdään vain osalle muuttujia

- avoin kaava voidaan myös tulkita totuusarvoiseksi funktioksi, joka ottaa parametrikseen arvot vapaille muuttujille
  - ts. vapaiden muuttujien arvojen oletetaan tulevan kaavan käyttötilanteesta

Predikaattilogiikan kaavojen tulkinta

- kuten propositiologiikassa, kaavojen totuusarvot riippuvat symbolien tulkinnasta
- nyt tulkittavana on
  - ei-loogisten arvojen alue (ei saa valita Ø)
  - vakio-, funktio- ja predikaattisymbolit
  - vapaat muuttujat
- esimerkkejä
  - $\forall x: x > 0 \rightarrow x \geq 1$ pätee kokonaisluvuille, mutta ei reaaliluvuille (symboleilla normaali tulkinta)
  - 1+1 = 1, jos tulkitaan “1” = “**T**” ja “+” = “$\vee$”
- toisinaan symboli “=” katsotaan osaksi logiikan kieltä ja tarkoittamaan “on sama arvo”
  - silloin “=”:n tulkintaa ei saa muuttaa
  - “=”:n tutut lait ovat vapaasti käytettävissä
  - tällä kurssilla tehdään niin
- kuten propositiologiikassa, kaava on *tautologia* eli *looginen totuus*, jos se pätee **kaikkien ei-loogisten symbolien** kaikilla tulkinnoilla
  - vastaavasti looginen seuraus
- esimerkiksi $\forall x: x > 1 \rightarrow x > 0$ ei ole tautologia, koska se ei päde, jos “>” tulkitaan “<”:ksi
  - vrt. $\forall x: x \succ 1 \rightarrow x \succ 0$
  - samoin $x > 1 \Rightarrow x > 0$ ei ole looginen seuraus

- esimerkkejä tautologioista:
  - $(P(x,y) \vee Q(y)) \wedge \neg P(x,y) \rightarrow Q(y)$
  - $1 > 0 \rightarrow \exists\, a\colon a > 0$
  - $x = y \rightarrow (P(x) \leftrightarrow P(y))$
- esimerkkejä loogisista seurauksista:
  - $(P(x,y) \vee Q(y)) \wedge \neg P(x,y) \Rightarrow Q(y)$
  - $1 > 0 \Rightarrow \exists\, a\colon a > 0$
  - $x = y \Rightarrow P(x) \leftrightarrow P(y)$
- avoinkin kaava voi olla tautologia
  - esim. $P(x) \vee \neg P(x)$
- yleensä vakio-, funktio- ja predikaattisymbolien tulkinta määritellään yleisoletuksissa tai oletetaan tunnetuksi, mutta vapaiden muuttujien ei
- jokaisella suljetulla kaavalla on tällöin kiinteä totuusarvo **T** tai **F**
- avoimen kaavan totuusarvo jää riippuvaksi vapaille muuttujille annettavista arvoista
  - esim. $x > 0 \vee \exists\, y\colon (\, x < y \wedge y \leq 0\,)$ on totta, kun $x \neq 0$, ja muulloin ei
- esimerkiksi ohjelmia todistettaessa
  - ohjelman muuttujat esiintyvät kaavoissa vapaina
  - niiden arvot katsotaan siitä ohjelman tilasta, jossa kaava tulkitaan

Kaavojen vahvuus

- jos $\varphi \Rightarrow \eta$ pätee kaikilla vapaiden muuttujien arvojen valinnoilla, niin φ on *ainakin yhtä vahva* väittämä kuin η, ja η on *ainakin yhtä heikko* kuin φ
- jos lisäksi vapaille muuttujille voidaan valita arvot siten, että $\neg\varphi \wedge \eta$, niin φ on *(aidosti) vahvempi* kuin kuin η ja η on *(aidosti) heikompi* kuin φ

- esimerkkejä
  - "$x > 1$" on aidosti vahvempi kuin "$x > 0$"
  - "$x \geq 1$" on reaaliluvuilla mutta ei ole kokonaisluvuilla aidosti vahvempi kuin "$x > 0$"

Päätteleminen predikaattilogiikassa

- käsitteet, päättelyoperaattorit "$\Rightarrow$" ja "$\Leftrightarrow$", päättelytekniikat jne. enimmäkseen samoin kuin propositiologiikassa
  - kertauksen vuoksi:

    Kaava φ *seuraa premisseistä* $\varphi_1, \ldots, \varphi_n$ *jos ja vain jos* $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \varphi$ *on tautologia.*
- yksi **hyvin olennainen** ero on: kaavan pätevyyttä ei enää voi aina testata kokeilemalla kaikilla arvoilla
  - jos ei-loogisten arvojen joukko on ääretön, on mahdotonta kokeilla kaikilla arvoilla
  - esim. kokonaisluvut:

    $\neg \exists x: \exists y: \exists z: \exists n:$
    $x \geq 1 \wedge y \geq 1 \wedge z \geq 1 \wedge n \geq 3 \wedge x^n + y^n = z^n$

$\Rightarrow$ päättelyn on pakko perustua lakeihin tms.

- (1. kertaluvun) predikaattilogiikassa pätee:
  - jos kaava φ seuraa loogisesti premisseistä, φ voidaan formaalisti todistaa
  - jos premissien vallitessa φ ei voi olla **T**, $\neg\varphi$ voidaan todistaa
  - jos premissit eivät määrää kaavan totuusarvoa, voi olla mahdoton havaita, että tilanne on tämä (algoritmia, jolla voi aina testata, onko kaava premissien seuraus vai ei, **ei voi olla** olemassa)
  - (jos premissit ovat ristiriitaiset, niin mikä tahansa kaava vodaan todistaa)

- (1. kertaluvun logiikasta joudutaan ulos jos esimerkiksi sallitaan predikaattisymbolien tai funktiosymbolien kvantifiointi:
  - $\exists P: \forall x: P(x)$
  - $\exists n: \exists a_1, a_2, \ldots, a_n: \ldots)$

Predikaattilogiikan (loogisia) lakeja

- kaikki edellä mainitut propositiologiikan lait pätevät myös predikaattilogiikassa
- lisäksi annamme joukon lakeja
  - samuusoperaattorille “=”
  - kvanttoreille

Samuusoperaattorin lait

- olkoot
  - $t, t_1, \ldots, t_n, t'_1, \ldots, t'_n$ termejä
  - f n-paikkainen funktiosymboli
  - P n-paikkainen predikaattisymboli
- refleksiivisyys:
  $t = t$
- symmetrisyys:
  $t_1 = t_2 \Rightarrow t_2 = t_1$
- transitiivisuus:
  $t_1 = t_2 \wedge t_2 = t_3 \Rightarrow t_1 = t_3$
- kongruenssiominaisuus funktioiden ja predikaattien suhteen:
  $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \Rightarrow f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$
  $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \Rightarrow$
  $P(t_1, \ldots, t_n) \leftrightarrow P(t'_1, \ldots, t'_n)$

Ennen kvanttorien lakeja määritellään muutamia käsitteitä, joilla estetään muuttujien nimien törmäykset

- olkoon t termi, x muuttuja ja φ kaava
- merkinnällä $\varphi(x)$ tarkoitetaan, että x:n vapaat esiintymät φ:ssä tulkitaan muuttujiksi, joiden paikalle voi sijoittaa termin
  - ts. φ tulkitaan x:n totuusarvoiseksi funktioksi
- t:n sijoitus x:n paikalle $\varphi(x)$:ään
  - merkitään $\varphi(t)$
  - tarkoittaa, että kaikkien x:n **vapaiden** esiintymien φ:ssä paikalle kirjoitetaan t
  - tarvittaessa t:n ympärille lisätään sulkuja takaamaan laskentajärjestyksen säilyminen
- esimerkkejä
  - jos $\varphi(x)$ on $x > 0 \wedge (\exists\, x\colon x < 0) \wedge x \neq 1$, niin $\varphi(y)$ on $y > 0 \wedge (\exists\, x\colon x < 0) \wedge y \neq 1$ ja $\varphi(x \cdot y)$ on $x \cdot y > 0 \wedge (\exists\, x\colon x < 0) \wedge x \cdot y \neq 1$
  - jos $\varphi(x)$ on $x \cdot 2 = 0$, niin $\varphi(y + 1)$ on $(y + 1) \cdot 2 = 0$
- sijoitus kohdistuu vain vapaisiin muuttujiin, koska samanniminen sidottu muuttuja on oikeastaan eri muuttuja
  - sidotun muuttujan arvo määräytyy kvanttorista
  - vapaa muuttuja saa arvonsa esim. ohjelman tilasta

  $\Rightarrow$ sijoitus sidottuun muuttujaan voisi johtaa virheelliseen lopputulokseen
- esimerkiksi $\exists\, x\colon (x > 0 \wedge \exists\, x\colon x < 0)$ sisältää itse asiassa kaksi sidottua muuttujaa seuraavasti: $\exists\, x_1\colon (x_1 > 0 \wedge \exists\, x_2\colon x_2 < 0)$

- *t on vapaa x:n paikalle $\varphi(x)$:ssä*, jos mikään t:ssä esiintyvä muuttuja ei joudu sidotuksi sijoituksen $\varphi(t)$ seurauksena
  - esim. $y + 1$ ei ole vapaa x:n paikalle kaavassa
    $\exists y: y > x$, koska sijoitus tuottaisi
    $\exists y: y > y + 1$, missä y on sidottu
  - esim. $y + 1$ on vapaa x:n paikalle kaavassa
    $x > 0 \wedge \forall x: \exists y: y > x$, koska sijoitus $\Rightarrow$
    $y + 1 > 0 \wedge \forall x: \exists y: y > x$
- yllä oleva siis vaatii, että
  jos y on t:ssä esiintyvä muuttuja,
  niin x ei esiinny vapaana φ:n osakaavoissa muotoa $\forall y: \varphi$ tai $\exists y: \varphi$
- tämä vaatimus takaa, että sijoituksen jälkeenkin t edustaa **yhtä**, vapaiden muuttujien arvojen määräämää, kvantifioiduista muuttujista riippumatonta arvoa
- yleensä t:n sijoitus x:n paikalle $\varphi(x)$:ään sallitaan vain kun t on vapaa x:n paikalle $\varphi(x)$:ssä
- erityisesti x on aina vapaa x:n paikalle $\varphi(x)$:ssä

Kvanttorien lakeja

- muuttujan vaihto: jos y ei esiinny vapaana $\varphi(x)$:ssä ja y on vapaa x:n paikalle $\varphi(x)$:ssä, niin
  $\forall x: \varphi(x) \Leftrightarrow \forall y: \varphi(y)$
  $\exists x: \varphi(x) \Leftrightarrow \exists y: \varphi(y)$
- esimerkkejä
  - $\exists y: y > x \Leftrightarrow \exists z: z > x$ oikein
  - $\exists y: y > x \Leftrightarrow \exists x: x > x$ virhe! x vapaa $\varphi(y)$:ssä
  - $\forall x: \exists y: y > x \Leftrightarrow \forall y: \exists y: y > y$ virhe!
    y ei vapaa x:n paikalle $\varphi(x)$:ssä

- kaikkikvanttorin poisto: olkoon t vapaa x:n paikalle $\varphi(x)$:ssä:
  $$\forall x: \varphi(x) \Rightarrow \varphi(t)$$
  - esim. $\forall x: \exists y: y > x \Rightarrow \exists y: y > 1917$
  - esim. sijoitus $x := y + 1$ on kielletty (se tuottaisi $\forall x: \exists y: y > x \Rightarrow \exists y: y > y + 1$)
  - esim. $\forall x: \varphi(x) \Rightarrow \varphi(x)$
- kaikkikvanttorin lisäys: jos x ei esiinny vapaana premisseissä eikä yleisoletuksissa, ja voidaan johtaa φ, niin saa kirjoittaa $\forall x: \varphi$
- olemassaolokvanttorin lisäys: olkoon t vapaa x:n paikalle $\varphi(x)$:ssä:
  $$\varphi(t) \Rightarrow \exists x: \varphi(x)$$
- DeMorganin lait
  $$\neg\exists x: \varphi \Leftrightarrow \forall x: \neg\varphi$$
  $$\neg\forall x: \varphi \Leftrightarrow \exists x: \neg\varphi$$
- kvanttoreiden vaihdot:
  $$\forall x: \forall y: \varphi \Leftrightarrow \forall y: \forall x: \varphi$$
  $$\exists x: \exists y: \varphi \Leftrightarrow \exists y: \exists x: \varphi$$
  $$\exists x: \forall y: \varphi \Rightarrow \forall y: \exists x: \varphi$$
  - **huom!** alin laki on yksisuuntainen
- osittelulakeja:
  $$\forall x: \varphi \wedge \eta \Leftrightarrow (\forall x: \varphi) \wedge \forall x: \eta$$
  $$\exists x: \varphi \vee \eta \Leftrightarrow (\exists x: \varphi) \vee \exists x: \eta$$
- lisää osittelulakeja, kun x ei esiinny vapaana φ:ssä:
  $$\forall x: \varphi \wedge \eta \Leftrightarrow \varphi \wedge \forall x: \eta$$
  $$\forall x: \varphi \vee \eta \Leftrightarrow \varphi \vee \forall x: \eta$$
  $$\exists x: \varphi \vee \eta \Leftrightarrow \varphi \vee \exists x: \eta$$
  $$\exists x: \varphi \wedge \eta \Leftrightarrow \varphi \wedge \exists x: \eta$$

- näitten ansiosta aiemmista osittelulaeista voi poistaa sulkuja:

$$\forall x: \varphi \wedge \eta \;\Leftrightarrow\; \forall x: \varphi \wedge \forall x: \eta$$
$$\exists x: \varphi \vee \eta \;\Leftrightarrow\; \exists x: \varphi \vee \exists x: \eta$$

- sekalaisia lakeja:

$$\exists x: \varphi \wedge \eta \;\Rightarrow\; \exists x: \varphi \wedge \exists x: \eta$$
$$\forall x: \varphi \vee \forall x: \eta \;\Rightarrow\; \forall x: \varphi \vee \eta$$
$$\forall x: \varphi \vee \eta \;\Rightarrow\; \exists x: \varphi \vee \forall x: \eta$$

- jos x ei esiinny vapaana termissä t:

$$\forall x: x = t \rightarrow \varphi(x) \;\Rightarrow\; \varphi(t)$$
$$\exists x: x = t \wedge \varphi(x) \;\Rightarrow\; \varphi(t)$$

Todistusesimerkkejä

- todistettava yksityiskohtaisesti

$$\forall x: \varphi \wedge \eta \;\Leftrightarrow\; \forall x: \varphi \wedge \forall x: \eta$$

sitä ennen annettujen lakien avulla:

$\forall x: \varphi \wedge \eta$	$\Leftrightarrow$	alkuper. osittelul.
$(\forall x: \varphi) \wedge \forall x: \eta$	$\Leftrightarrow$	lisää sulkuja
$(\forall x: \varphi) \wedge (\forall x: \eta)$	$\Leftrightarrow$	“$\wedge$” vaihdannainen
$(\forall x: \eta) \wedge (\forall x: \varphi)$	$\Leftrightarrow$	osittelu, x ei vapaa $(\forall x: \eta)$:ssa
$\forall x: ((\forall x: \eta) \wedge \varphi)$	$\Leftrightarrow$	“$\wedge$” vaihdann.
$\forall x: (\varphi \wedge (\forall x: \eta))$	$\Leftrightarrow$	turhat sulut pois
$\forall x: \varphi \wedge \forall x: \eta$		

- todistettava
  $\forall x: \eta \vee \gamma \rightarrow \varphi \;\Leftrightarrow\; (\forall x: \eta \rightarrow \varphi) \wedge (\forall x: \gamma \rightarrow \varphi)$
  - osittelulaki antaa:
    $(\forall x: \eta \rightarrow \varphi) \wedge (\forall x: \gamma \rightarrow \varphi) \;\Leftrightarrow$
    $\forall x: (\eta \rightarrow \varphi) \wedge (\gamma \rightarrow \varphi)$
  - käsitellään sisäosaa:
    $(\eta \rightarrow \varphi) \wedge (\gamma \rightarrow \varphi) \;\Leftrightarrow$
    $(\neg\eta \vee \varphi) \wedge (\neg\gamma \vee \varphi) \;\Leftrightarrow$
    $(\neg\eta \wedge \neg\gamma) \vee \varphi \;\Leftrightarrow$
    $\neg(\eta \vee \gamma) \vee \varphi \;\Leftrightarrow$
    $\eta \vee \gamma \rightarrow \varphi$
  - "⇔":n kongruenssiominaisuuden ansiosta
    $\forall x: (\eta \rightarrow \varphi) \wedge (\gamma \rightarrow \varphi) \;\Leftrightarrow\; \forall x: \eta \vee \gamma \rightarrow \varphi$

Kvanttorien suhde perusoperaattoreihin

- jos muuttujan x arvoalue on äärellinen joukko $\{x_1, x_2, \ldots, x_n\}$, niin
  $\forall x: \varphi(x) \;\Leftrightarrow\; \varphi(x_1) \wedge \varphi(x_2) \wedge \ldots \wedge \varphi(x_n)$
  $\exists x: \varphi(x) \;\Leftrightarrow\; \varphi(x_1) \vee \varphi(x_2) \vee \ldots \vee \varphi(x_n)$
- tämä suhde auttaa joskus muistamaan kaavoja
  - esim. $\neg\exists x: \varphi \;\Leftrightarrow\; \forall x: \neg\varphi$ vastaa propositiologiikan DeMorganin lakia

Lyhennysmerkintöjä kvanttoreille

- näillä saadaan kvanttoreita sisältävät kaavat helpommiksi lukea ja käsitellä
- muuttujan arvon rajaus johonkin joukkoon
  - määritelmä:
    $\forall x \in A: \varphi \;\Leftrightarrow\; \forall x: x \in A \rightarrow \varphi$
    $\exists x \in A: \varphi \;\Leftrightarrow\; \exists x: x \in A \wedge \varphi$
  - esimerkkejä ($\boldsymbol{R}$ on reaalilukujen joukko):
    $\forall x \in \boldsymbol{R}: x^2 \geq 0$
    $\exists x \in \boldsymbol{R}: x^2 = 2$

- rajaajana voi käyttää myös predikaattia
  - määritelmä:
    $\forall x ; \eta: \varphi \Leftrightarrow \forall x: \eta \rightarrow \varphi$
    $\exists x ; \eta: \varphi \Leftrightarrow \exists x: \eta \wedge \varphi$
  - esim. $\forall i ; 1 \leq i \leq n: A[i] = 0$
- muuttujia voi luetella monta samassa kvanttorissa
  - määritelmä:
    $\forall x_1, x_2, \ldots, x_n: \varphi \Leftrightarrow \forall x_1: \forall x_2: \ldots \forall x_n: \varphi$
    $\exists x_1, x_2, \ldots, x_n: \varphi \Leftrightarrow \exists x_1: \exists x_2: \ldots \exists x_n: \varphi$
  - esim. kaikki-kvanttorin vaihtolaki sievenee
    $\forall x, y: \varphi \Leftrightarrow \forall y, x: \varphi$
  - (**huom!** jos yllä n ei ole luku vaan esim. kaavan muuttuja, niin joudutaan ulos 1. kertaluvun logiikasta)
- lyhenteitä voi yhdistellä
  - esim. $\exists x, y, z \in \boldsymbol{Z}^+: x^2 + y^2 = z^2$
    ($\boldsymbol{Z}^+$ on positiivisten kokonaislukujen joukko)
- kvanttorin negaatio voidaan esittää yliviivauksella
  - esim. $\nexists x: \varphi \Leftrightarrow \neg\exists x: \varphi$
- pari lakia:
  $\forall x ; \eta \vee \gamma: \varphi \Leftrightarrow \forall x ; \eta: \varphi \wedge \forall x ; \gamma: \varphi$
  $\exists x ; \eta \vee \gamma: \varphi \Leftrightarrow \exists x ; \eta: \varphi \vee \exists x ; \gamma: \varphi$

Laskeminen predikaattilogiikassa

- yksityiskohtainen laskenta predikaattilogiikassa johtaa usein hyvin pitkiin laskelmiin
  $\Rightarrow$ ei käytännöllistä
- laskennassa joudutaan vetoamaan sovellusalueen “tuttuihin” ominaisuuksiin
  - esim. tietoon $n \geq 1 \Rightarrow n > 0$
  - nämä voitaisiin lisätä formalismiin sopivilla aksioomilla, mutta ei yleensä maksa vaivaa

- käytännössä käytetään epäformaalia järkeilyä
  - hankalissa tai epävarmoissa kohdissa kannattaa olla tavallista formaalimpi
  - yllä olevia kaavoja voi käyttää apuna
- propositiologiikan yhteydessä "$\Rightarrow$":lle ja "$\Leftrightarrow$":lle annetut lait ovat edelleen voimassa
  - mm. kaavan osan saa korvata ekvivalentilla
- usein on kätevää pilkkoa todistus pieniin tapauksiin, jotka käsitellään kukin erikseen
- usein on kätevää johtaa φ lisäämällä oletuksiin $\neg\varphi$ ja johtamalla ristiriita
- todistuksen rakentamisessa on kovasti apua kaavojen esittämien väittämien ymmärtämisestä ja tilannekohtaisesta näkemyksestä!
  - auttaa valitsemaan todistukselle suunnan ja välivaiheet
- kuten propositiologiikassa, sitkeiden todistusyritysten epäonnistuminen voi johtua siitä, että kaikkia tarpeellisia lähtökohtia ei ole otettu mukaan päättelyyn
  - erityisesti sovellusaluekohtainen tieto voi jäädä helposti puuttumaan
- ohjelmia todistettaessa voi olla, että ohjelma ei edes toimi, ellei lähtöoletuksia hieman tarkenneta!

Esimerkki: löysikö hakuohjelma etsimänsä?

- kurssilla OHJ-2500 Ohjelmien todistaminen osoitetaan, että erään hakuohjelman loputtua pätee

$$1 \le a \le n+1 \land (a = n+1 \lor A[a] \ge x) \land (a = 1 \lor A[a-1] < x)$$

- osoitettava, että jos lisäksi taulukko $A[1\ldots n]$ on järjestyksessä ja alkio x on siinä, niin se on a:n kohdalla
  - (ts. ainakin yksi x on a:n kohdalla)
- formalisoimme oletukset:
  $\forall\, i, j\ ;\ 1 \le i \le j \le n\colon A[i] \le A[j]$
  $\exists\, i\ ;\ 1 \le i \le n\colon A[i] = x$
  ja tavoitteen: $A[a] = x$
- osoitamme aluksi, ettei x voi olla ennen kohtaa a:
  - tiedämme, että $a = 1 \vee A[a-1] < x$
  - jos $a = 1$, niin $1 \le i < a \Leftrightarrow$ **F**, joten
    $\forall\, i\ ;\ 1 \le i < a\colon A[i] \ne x$
  - jos $A[a-1] < x$, niin
    $\forall\, i, j\ ;\ 1 \le i \le j \le n\colon A[i] \le A[j] \Rightarrow$
    $\forall\, i\ ;\ 1 \le i \le a-1 \le n\colon A[i] \le A[a-1] < x \Rightarrow$
    $\forall\, i\ ;\ 1 \le i < a\colon A[i] \ne x$
- tämä tulos ja tieto $\exists\, i\ ;\ 1 \le i \le n\colon A[i] = x$
  tuottavat $\exists\, i\ ;\ a \le i \le n\colon A[i] = x$
- jos $a = n+1$, niin
  $\exists\, i\ ;\ a \le i \le n\colon A[i] = x \Rightarrow n+1 \le n \Leftrightarrow$ **F**
  joten oltava $a \ne n+1$

$\Rightarrow A[a] \ge x$

- jos $A[a] \ne x$, niin
  $A[a] > x \Rightarrow \forall\, j\ ;\ a \le j \le n\colon x < A[a] \le A[j] \Rightarrow$
  $\forall\, i\ ;\ a \le i \le n\colon A[i] \ne x \Rightarrow$ ristiriita

$\Rightarrow$ jäljelle jää vain mahdollisuus $A[a] = x$, M. O. T.

Merkintöjä “$\not\Rightarrow$” ja “$\not\Leftrightarrow$” tulee käyttää varoen!

- esimerkiksi $\varphi \not\Rightarrow \eta$ yrittää tarkoittaa “φ ei implikoi η:ta”

- mutta mitä seuraavista se tarkkaan ottaen tarkoittaa?
  - $\varphi \Rightarrow \eta$ ei päde koskaan, esim. $x = x \not\Rightarrow x \neq x$
  - η ei päde koskaan silloin kun φ pätee, esim. $x \neq 0 \not\Rightarrow x^2 \leq 0$
  - η ei päde aina silloin kun φ pätee, esim. $x \neq 0 \not\Rightarrow x^2 > 4$
  - η ei ole φ:n looginen seuraus, esim. “jouluaatto on 24.12.” $\not\Rightarrow (-2)^2 = 4$
- sen sijaan “$\not\rightarrow$” ja “$\not\leftrightarrow$” ovat turvallisia
  - niitä ei niin helposti mielletä päättelysääntöinä
  - $\varphi \not\rightarrow \eta \Leftrightarrow \neg(\varphi \rightarrow \eta)$
  - $\varphi \not\leftrightarrow \eta \Leftrightarrow \neg(\varphi \leftrightarrow \eta)$

Määrittelemättömien tilapredikaattien ongelma

- tilapredikaatin arvo ei ole välttämättä aina määritelty
  - esim. “$A[i] = x$”, kun A:n indeksialue on 1, …, n ja $i = -3$
  - esim. “$p\text{^}.avain = x$”, kun osoittimen p arvo on Nil
  - esim. “$^1/_x = \ldots$”, kun $x = 0$
  - esim. “merkkijonon α ensimmäinen merkki on …”, kun α on tyhjä merkkijono
- määrittelemättömiä tilapredikaatteja tarvitaan usein osana isompaa lauseketta, jonka muut osat intuitiivisesti ajateltuna riittävät määräämään totuusarvon silloinkin, kun ko. osa on määrittelemätön
  - esim. $i = -3 \vee A[i] = x$ ➠ **T**, kun $i = -3$
  - esim. $p = \text{Nil} \vee p\text{^}.avain = x$ ➠ **T**, kun $p = \text{Nil}$
  - esim. $x > 0 \wedge {}^1/_x > 0$ ➠ **F**, kun $x = 0$
- usein määrittelemättömyys pyritään välttämään käyttämällä kvantifikaattorin rajausosaa
  - esim. $\forall x ; x \neq 0: {}^1/_x \neq 0$

- tämä ei kuitenkaan oikeasti ratkaise ongelmaa
  - esim. $\forall x ; x \neq 0: {}^{1}/_{x} \neq 0$ on "$\forall x ; \eta: \varphi$":n, "$\rightarrow$":n yms. määritelmien mukaan sama kuin
    $\forall x: {}^{1}/_{x} \neq 0 \vee x = 0$
- kirjallisuudesta on erittäin vaikea löytää käyttökelpoista selkeää kantaa sille, miten määrittelemättömät tilapredikaatit tulisi yleensä tulkita
  - joskin lausekkeet joissa "ympäristö määrää totuusarvon" tulkitaan "salaa" ym. tavalla
- ei se ole helppoa itsellekään
  - päteekö ${}^{1}/_{0} = 3$?
  - päteekö $\forall x: {}^{1}/_{0} \neq x$?
  - päteekö ${}^{1}/_{0} \neq {}^{1}/_{0}$?
  - x voi olla ${}^{1}/_{0}$, päteekö $\forall x: x < 0 \vee x = 0 \vee x > 0$?

Määrittelemättömyys ohjelmointikielissä

- ohjelmissa yritykset laskea määrittelemätön lauseke johtavat yleensä johonkin seuraavista:
  - ohjelma kaatuu
  - ohjelma jatkaa, mutta tulokseen ei voi luottaa
  - ohjelma jää ikuiseen silmukkaan (harvinaista)

$\Rightarrow$ huolellinen ohjelmoija kirjoittaa ohjelmat siten, että määrittelemättömiä lausekkeita ei yritetä laskea

- esimerkki: Pascal-kääntäjä saa aina laskettaa loogisen lausekkeen molemmat puolet
  $\Rightarrow$ **if** $p \neq$ Nil **and** p^.$avain = x$ **then** … voi johtaa ohjelman kaatumiseen, kun p = Nil
  $\Rightarrow$ Pascal-ohjelmoijan tulee kirjoittaa esim.

```
if p ≠ Nil then
    if p^.avain = x then
        …
```

- monet ohjelmointikielet tarjoavat keinoja ohjata totuusarvoisen lausekkeen osien laskentaa
- esimerkki: C:ssä ja C++:ssa loogisen operaattorin oikea puoli lasketaan vain, jos lopputulos ei ole "selvä" vasemman puolen laskennan jälkeen
  $\Rightarrow$ jos $i = -1$, niin lausekkeessa
  `( i == -1 || A[i] == x )`
  ei edes yritetä laskea, päteekö $A[i] = x$
- esimerkki: Adassa on erikseen operaattorit "**and**" ja "**and then**" sekä "**or**" ja "**or else**"
  – "**and**" ja "**or**": lasketaan aina molemmat puolet
  – "**and then**" ja "**or else**": kuten C:n "&&" ja " | | "
  – esim. **if** $i > 0$ **and then** $A[i] = x$ **then** …

Määrittelemättömien tilapredikaattien ongelma on sotkuinen, ja monenlaisia ratkaisuja on ehdotettu

- määrittelemättömiä lausekkeita ei saa kirjoittaa
  – ei onnistu: jatkuvasti tarvitaan kaavoja tyyliin
  $p = \text{Nil} \vee p\text{^}.avain = x$
  $i = 0 \vee A[i] \geq A[i - 1]$
- sovitaan, että jokainen määrittelemättömän lausekkeen sisältävä predikaatti tuottaa **F**
  – olkoon määritelty $P(x,y) :\Leftrightarrow x = y$ ja
  $Q(x,y) :\Leftrightarrow \neg P(x,y)$ (siis tarkoittaa $x \neq y$)
  – toisaalta $Q(^1/_0, ^1/_0) \Leftrightarrow$ **F** (eli ei $^1/_0 \neq ^1/_0$)
  – toisaalta $Q(^1/_0, ^1/_0) \Leftrightarrow \neg P(^1/_0, ^1/_0) \Leftrightarrow \neg$**F**!
- (":$\Leftrightarrow$" tarkoittaa "predikaatti määritellään tarkoittamaan"
  – merkintä tullaan perustelemaan luvussa 2.5)

- "**and then**" ja "**or else**" eivät ole vaihdannaisia
  - esim. kun $x = 0$, niin $x \leq 0$ **or else** $1/x \geq 0$ = **true** ja $1/x \geq 0$ **or else** $x \leq 0$ on määrittelemätön
  - $\Rightarrow$ "$\wedge$":n ja "$\vee$":n korvaaminen niillä romuttaisi monta tuttua kaavaa
  - $\Rightarrow$ ei suotavaa
- lisätään logiikkaan kolmas totuusarvo "**undefined**"
  - esim. "Vienna Development Method"
  - osa tutuista kaavoista menetetään
  - esim. $P \vee \neg P \Leftrightarrow$ **T** ei päde VDM:ssä
  - esim. $P \rightarrow P$ ei päde VDM:ssä
  - $\Rightarrow$ emme omaksu tätä!
- päätetään, että jokaisella lausekkeella on aina arvo, mutta aina ei tiedetä mikä arvo
  - siis esim. $1/0$ on jokin luku, muttemme tiedä mikä
  - tällöin $1/0 < 0 \vee 1/0 = 0 \vee 1/0 > 0$ pätee, mutta ei tiedetä mikä kolmesta vaihtoehdosta pätee
  - asiaa vakavasti tutkineista esim. Schneiderin kanta
  - omituinen ratkaisu, mutta toimii edes jotenkuten

- oletetaan yksi kokonaan uusi, perustyyppien ulkopuolinen arvo “⊥”, joka luetaan “määrittelemätön” tai “pohja”, ja tulkitaan määrittelemättömän lausekkeen arvoksi
  - vrt. osoittimen arvo “Nil”
  - erityisesti kaikki funktion kutsut, joissa jokin argumentti on ⊥, tuottavat ⊥
  - esim. $^1/_0 = \bot$, $3 \cdot \bot + 2 = \bot$, Nil^.*avain* = ⊥
  - $\forall\, x$; φ: η tulkitaan tarkoittamaan $\forall\, x$: ($\varphi \wedge x \neq \bot$) $\rightarrow$ η, ja vastaavasti $\exists\, x$; φ: η
  - haitta: voidaan osoittaa esim. min{ $x \in \boldsymbol{R}$ | $x > 0$ } = $^1/_0$
  - omituinen ratkaisu, mutta toimii edes jotenkuten

Tällä kurssilla

- osittain määriteltyjä predikaatteja saa kirjoittaa, jos koko kaavan totuusarvo määräytyy ympäristöstä seuraavilla säännöillä:
  F $\wedge$ (mitä tahansa) $\Leftrightarrow$ (mitä tahansa) $\wedge$ **F** $\Leftrightarrow$ **F**
  T $\vee$ (mitä tahansa) $\Leftrightarrow$ (mitä tahansa) $\vee$ **T** $\Leftrightarrow$ **T**
  F $\rightarrow$ (mitä tahansa) $\Leftrightarrow$ (mitä tahansa) $\rightarrow$ **T** $\Leftrightarrow$ **T**
- **jos** tulkinnalla on väliä, se on jompikumpi alimmista

Onko jokainen nainen

- äitinsä tytär?
- tyttärensä äiti?
- tyttäriensä äiti?
- entä jokainen lapseton mies?

2.4 Joukko-oppia

Miksi tarpeen?

- joukko on matematiikan kenties tärkein peruskäsite
  - tärkeämpi kuin luvut!
  - ⇒ muiden käsitteiden (myös ohjelmistotekniikan teoriassa esiintyvien) formalisoinnissa käytetään paljon joukkoja

⇒ monet formaalit määrittelymenetelmät perustuvat joukko-oppiin
  - esim. VDM, Z

- joukot tarjoavat lisää työkaluja tilapredikaattien muodostamiseen
  - esim. mahdollistavat tietyn ehdon täyttävien alkioiden määrän laskemisen
- algoritmeista voi hävittää epäolennaisia toteutusyksityiskohtia joukkoabstraktion avulla
- esimerkki: kurssin välittömien ja välillisten esitietojen läpikäynti

$kesken := \{k_0\}$; $löydetyt := \emptyset$
while $kesken \neq \emptyset$ **do**
  $valitse\text{-}jokin\ k \in kesken$
  forall $g \in välitt\text{-}esit(k)$ **do**
    if $g \notin löydetyt$ **then**
      $löydetyt := löydetyt \cup \{g\}$
      if $g \neq k_0$ **then** $kesken := kesken \cup \{g\}$ **endif**
    endif
  endfor
  $kesken := kesken - \{k\}$
endwhile

- esimerkkialgoritmi esittää läpikäynnin periaatteen ottamatta kantaa kurssien ja esitietorelaation toteutuksiin
  - *kesken* voidaan toteuttaa mm. jonona ja *löydetyt* jokaiseen kurssitietueeseen lisättävällä bitillä
- esimerkkialgoritmi on melko helppo osoittaa oikeaksi
- kun se on osoitettu oikeaksi, konkreettisen toteutuksen oikeaksi osoittamiseksi riittää esimerkiksi todistaa, että valitut jono- ja bittioperaatiot toteuttavat algoritmin joukko-operaatiot

$\Rightarrow$ ohjelman ymmärtäminen ja oikeaksi osoittaminen saadaan pilkottua kahteen pienempään, mahdollisesti uudelleen käytettävään osaan

Joukkojen merkintöjä

- $\{x_1, x_2, \ldots, x_n\}$ *joukko*, jossa on alkiot $x_1, x_2, \ldots, x_n$
  - esim. {1,5,8}, {**F**, **T**}, {'a', 'b', …, 'g'}
  - alkioiden luettelointijärjestyksellä ei ole väliä
  - alkion esiintymiskertojen määrällä ei ole väliä, kunhan ≥ 1
  - esim. {1,5,8,8,5,1} = {1,5,8} = {8,5,1}
- $\emptyset$ *tyhjä joukko*
  - pätee: $A = \emptyset \Leftrightarrow \forall x: x \notin A$
- $\{ x \mid P(x) \}$ niiden x joukko, joille kaava $P(x)$ pätee
  - esim. $\{ x \mid x^2 = 1 \} = \{ -1, 1 \}$
  - muunnelmia:

    $\{ x \in A \mid P(x) \} = \{ x \mid x \in A \wedge P(x) \}$

    $\{ f(x) \mid P(x) \} = \{ y \mid \exists x: y = f(x) \wedge P(x) \}$

    (alimmassa oltava varovainen, että näkyy, mikä on paikallinen muuttuja, vrt. $A_y = \{ x+y \mid x < y \}$)

Joukko-opin perusrelaatioita

- $a \in A$ alkio *a kuuluu* joukkoon A
  $a \notin A$ alkio *a ei kuulu* joukkoon A
  - $a \in \{x_1, x_2, \ldots, x_n\} \quad \Leftrightarrow \exists\, i\,;\, 1 \le i \le n\colon a = x_i$
  - $a \in \{\, x \mid P(x) \,\} \quad \Leftrightarrow P(a)$
  - $a \notin A \quad \Leftrightarrow \neg(a \in A)$
- $A = B$ A ja B ovat sama joukko
  $A \neq B$ A ja B eivät ole sama joukko
  - "=": jokainen A:n alkio on myös B:n alkio, ja toisin päin
  - $A = B \;\Leftrightarrow\; \forall\, x\colon\; x \in A \;\leftrightarrow\; x \in B$
  - $A \neq B \;\Leftrightarrow\; \neg(A = B) \;\Leftrightarrow$
    $(\,\exists\, x\colon x \in A \wedge x \notin B\,) \;\vee\; (\,\exists\, x\colon x \in B \wedge x \notin A\,)$
- $A \subseteq B$ A on B:n *osajoukko*
  - jokainen A:n alkio on myös B:n alkio
  - $A \subseteq B \;\Leftrightarrow\; \forall\, x\colon\; x \in A \;\rightarrow\; x \in B$
  - esim. $\{2,3\} \subseteq \{1,2,3\}$ ja $\{1,2,3\} \subseteq \{1,2,3\}$
- $A \subset B$ A on B:n *aito* osajoukko
  - $A \subset B \;\Leftrightarrow\; A \subseteq B \wedge A \neq B$
  - jokainen A:n alkio on myös B:n alkio, mutta B:ssä on ainakin yksi alkio, joka ei ole A:ssa
  - $\{2,3\} \subset \{1,2,3\}$, mutta $\{1,2,3\} \not\subset \{1,2,3\}$

Joukko-opin perusoperaatioita

- $A \cup B$ A:n ja B:n *unioni*
  - sisältää sekä A:n että B:n alkiot
  - $A \cup B = \{\, x \mid x \in A \vee x \in B \,\}$
  - esim. $\{1,2,3\} \cup \{2,4\} = \{1,2,3,4\}$
- $A \cap B$ A:n ja B:n *leikkaus*
  - sisältää A:n ja B:n yhteiset alkiot
  - $A \cap B = \{\, x \mid x \in A \wedge x \in B \,\}$
  - esim. $\{1,2,3\} \cap \{2,4\} = \{2\}$

- $A - B$ A:n ja B:n *erotus*
  - sisältää ne A:n alkiot, jotka eivät kuulu B:hen
  - $A - B = \{ x \mid x \in A \wedge x \notin B \}$
  - esim. $\{1,2,3\} - \{2,4\} = \{1,3\}$
- $\overline{A}$ A:n *komplementti*
  - määritelty vain, jos kaikki käsiteltävät alkiot kuuluvat johonkin ennalta sovittuun joukkoon (universumiin) U (muuten saadaan aikaan kuuluisia paradokseja)
  - sisältää ne universumin alkiot, jotka eivät kuulu A:han
  - $\overline{A} = U - A$
  - esim. jos $U = \{\ldots,-2,-1,0,1,2,\ldots\}$, niin $\overline{\{1,2,3\}} = \{\ldots,-2,-1,0\} \cup \{4,5,6,\ldots\}$
- $A \times B$ A:n ja B:n *tulojoukko* eli *karteesinen tulo* (*Cartesian product*)
  - sisältää kaikki parit, joiden ensimmäinen komponentti kuuluu A:han ja toinen B:hen
  - $A \times B = \{ (x,y) \mid x \in A \wedge y \in B \}$
  - esim. $\{1,2,3\} \times \{2,4\} =$ $\{ (1,2), (1,4), (2,2), (2,4), (3,2), (3,4) \}$
  - yleistys: $A_1 \times A_2 \times \ldots \times A_n =$ $\{ (x_1,x_2,\ldots,x_n) \mid \forall\, i\, ;\, 1 \le i \le n\colon x_i \in A_i \}$
  - joskus käytetään selvyyden vuoksi toisenlaisia sulkuja: $\langle x_1,x_2,\ldots,x_n \rangle$
- A^n A:n n:s *potenssi*
  - $A^n = A \times A \times \ldots \times A$ (n kpl)
  - esim. $\{1,2\}^2 = \{ (1,1), (1,2), (2,1), (2,2) \}$
- $A^* = A^0 \cup A^1 \cup A^2 \cup \ldots$
- $A^+ = A^1 \cup A^2 \cup A^3 \cup \ldots$

Joukko-opin lakeja

- $A \subseteq B \wedge B \subseteq A \Leftrightarrow A = B$
- $A \cap B \subseteq A \subseteq A \cup B$
  $A - B \subseteq A$
- $\emptyset \subseteq A$
- $A \cup A = A \cap A = A$ ja $A - A = \emptyset$
  $A \cup \emptyset = A - \emptyset = A$ ja $A \cap \emptyset = \emptyset$
- vaihdannaisuus ja liitännäisyys:
  $A \cup B = B \cup A$ $\quad (A \cup B) \cup C = A \cup (B \cup C)$
  $A \cap B = B \cap A$ $\quad (A \cap B) \cap C = A \cap (B \cap C)$
- osittelulakeja:
  $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
  $A \times (B \cup C) = (A \times B) \cup (A \times C)$
  $A \times (B \cap C) = (A \times B) \cap (A \times C)$
- absorptiolakeja:
  $A \cup (A \cap B) = A$
  $A \cap (A \cup B) = A$
- DeMorganin lait:
  $A - (B \cup C) = (A - B) \cap (A - C)$
  $A - (B \cap C) = (A - B) \cup (A - C)$
- A:n osittelu B:n avulla:
  $A = (A - B) \cup (A \cap B)$
  $(A - B) \cap (A \cap B) = (A - B) \cap B = \emptyset$
- komplementtilakeja:
  $\overline{\overline{A}} = A$, $A \cup \overline{A} = U$ ja $A \cap \overline{A} = \emptyset$
- DeMorganin lait, komplementtiversio:
  $\overline{A \cup B} = \overline{A} \cap \overline{B}$ $\quad \overline{A \cap B} = \overline{A} \cup \overline{B}$

Potenssijoukko

- A:n *potenssijoukko* on A:n kaikkien osajoukkojen muodostama joukko:
  $$2^A = \mathcal{P}(A) = \{ X \mid X \subseteq A \}$$
  – se on siis joukko, jonka alkiotkin ovat joukkoja
- $\emptyset \in 2^A$ ja $A \in 2^A$
- esimerkiksi $2^{\{1,2\}} = \{ \emptyset, \{1\}, \{2\}, \{1,2\} \}$

Äärellisten joukkojen koot

- $|A|$ = joukon A alkioiden määrä
- $|\emptyset| = 0$, $|A| \geq 0$, $|A \cup B| \leq |A|+|B|$ jne.
- huom! $|\{x_1, x_2, \ldots, x_n\}| \leq n$

Lukujoukot

- $\boldsymbol{N}$ *luonnollisten lukujen* joukko $\{0,1,2,\ldots\}$
- $\boldsymbol{Z}$ *kokonaislukujen* joukko $\{\ldots,-2,-1,0,1,2,\ldots\}$
- $\boldsymbol{Q}$ *rationaalilukujen* joukko $\{ {}^{x}/_{y} \mid x, y \in \boldsymbol{Z} \wedge y \neq 0 \}$
- $\boldsymbol{R}$ *reaalilukujen* joukko
- $\boldsymbol{Z}^+ = \{ x \in \boldsymbol{Z} \mid x > 0 \}$ ja $\boldsymbol{Z}^- = \{ x \in \boldsymbol{Z} \mid x < 0 \}$
  – vastaavasti $\boldsymbol{Q}^+$, $\boldsymbol{Q}^-$, $\boldsymbol{R}^+$ ja $\boldsymbol{R}^-$
- kirjainten $\boldsymbol{N}$ jne. pitäisi olla onttoja, mutta tällä tekstinkäsittelyohjelmalla niitä ei saa ontoiksi

Operaattorit

- *operaattori* = symboli, joka edustaa funktiota, joka ottaa nolla tai useamman arvon ja tuottaa yhden arvon
- *unaari*operaattorilla on yksi argumentti
  – esim. “$\neg$”, etumerkki-“–”
  – kirjoitetaan yleensä argumenttinsa eteen, esim. “$-x$”, “$\neg(P \wedge Q)$”

- *binääri*operaattorilla on kaksi argumenttia
  - esim. “$\wedge$”, “$\cap$”, “+”, “–”
  - pannaan yleensä argumenttiensa väliin, esim. “x+3”, “$P \wedge Q$”
- nolla-argumenttinen operaattori on vakiosymboli
  - esim. “0”, “1”, “π”, “**F**”

Binäärioperaattorien ominaisuuksia

- olkoon “$\clubsuit$” jokin binäärioperaattori
- “$\clubsuit$” on *liitännäinen* eli *assosiatiivinen*, joss
  $$\forall\, x, y, z\colon\ (x \clubsuit y) \clubsuit z \;=\; x \clubsuit (y \clubsuit z)$$
  - seuraavat operaattorit ovat liitännäisiä:
    “$\wedge$”, “$\vee$”, “$\cup$”, “$\cap$”, “$\leftrightarrow$”
  - seuraavat operaattorit eivät ole liitännäisiä:
    “$\rightarrow$”, “$\times$”, (joukko- tai aritmeettinen) “–”
- jos “$\clubsuit$” on liitännäinen, sulkuja voi jättää — ja yleensä jätetään — pois:
  $$(x \clubsuit y) \clubsuit z \;=\; x \clubsuit (y \clubsuit z) \;=\; x \clubsuit y \clubsuit z$$
- “$\clubsuit$” on *vaihdannainen* eli *kommutatiivinen*, joss
  $$\forall\, x, y\colon\ x \clubsuit y \;=\; y \clubsuit x$$
  - seuraavat operaattorit ovat vaihdannaisia:
    “$\leftrightarrow$”, “$\wedge$”, “$\vee$”, “$\cup$”, “$\cap$”
  - seuraavat operaattorit eivät ole vaihdannaisia:
    “$\rightarrow$”, “–”, “$\times$”
- miksi “$\times$” ei ole vaihdannainen?
  - parin osien järjestys on olennainen
    $$\{1\} \times \{2\} = \{(1,2)\} \neq \{(2,1)\} = \{2\} \times \{1\}$$
- miksi “$\times$” ei ole liitännäinen?
  - alkioiden ryhmittely pareiksi on erilainen
    $$\{((1,1),1)\} = (\{1\} \times \{1\}) \times \{1\} \neq$$
    $$\{(1,(1,1))\} = \{1\} \times (\{1\} \times \{1\}) \neq$$
    $$\{(1,1,1)\} = \{1\} \times \{1\} \times \{1\}$$

- joissakin sovelluksissa olisi kätevää, jos pätisi

$$A \times B \times C = (A \times B) \times C = A \times (B \times C)$$

  $\Rightarrow$ toisinaan erikseen sovitaan, että tämä pätee
  – tarkoittaa, että päätetään olla välittämättä tavasta, jolla lista rakentuu osalistoista; vain listaan kuuluvat alkiot ja niiden järjestys merkitsevät

2.5 Tilapredikaatin kirjoittaminen

Esimerkki: taulukko $A[1\ldots n]$ on järjestyksessä

- miksi seuraava ei kelpaa?

$$\textit{järjestyksessä1}(A[1\ldots n]) \;:\Leftrightarrow$$
$$\forall\, i\,;\, 1 \le i \le n\colon A[i] \le A[i+1]$$

- korjattu versio:

$$\textit{järjestyksessä}(A[1\ldots n]) \;:\Leftrightarrow$$
$$\forall\, i\,;\, 1 \le i \le n-1\colon A[i] \le A[i+1]$$

- minkä lisävaatimuksen seuraava asettaa?

$$\textit{järjestyksessä2}(A[1\ldots n]) \;:\Leftrightarrow$$
$$\forall\, i\,;\, 1 \le i \le n-1\colon A[i] < A[i+1]$$

Määrittelymerkinnät $:\Leftrightarrow$ ja $:=$

- tarkoittavat “määritellään tarkoittamaan”
  - $:\Leftrightarrow$ predikaateille
  - $:=$ muille
- määritelmän *symboli* := *lauseke* jälkeen pätee *symboli* = *lauseke*
  - vastaavasti $:\Leftrightarrow$
- määritelmä tekee muutakin kuin on yhtälö
  - ottaa käyttöön uuden symbolin

$\Rightarrow$ hyvä olla oma, epäsymmetrinen merkintä

- kirjallisuudessa esiintyy mm. $=_{\text{def}}$ ja $\triangleq$
- $:\Leftrightarrow$ ja $:=$ matkivat ohjelmointikielten sijoitusta

Esimerkki: paikan valinta järjestetystä taulukosta

- oletamme, että $\textit{järjestyksessä}(A[1\ldots n])$ ja $n \ge 1$

- haluamme predikaatin *paikka*(A[1…n], *avain*, x), joka sanoo, että
  - jos *avain* on taulukossa, x osoittaa missä kohti
  - muutoin x osoittaa sitä kohtaa, missä alkion *avain* “pitäisi olla”
- x osoittaa taulukon kohtaa
  ⇒ järkevät x:n arvot ovat välillä $1 \le x \le n$
  ⇒ ei haittaa, vaikka predikaatti olisi muulloin määrittelemätön
- mitä “pitäisi olla” tarkoittaa?
  - esim. *avain* = 7

3	5	6	9	11	12	18	18	22	23
1	2	3	4	5	6	7	8	9	10

↑ ↑

x? x?

  - valinta: “pitäisi olla” ~ “lähinnä suurempi”
- 1. yritys: *paikka1*(A[1…n], *avain*, x) :⇔
  $A[x] = avain \vee (A[x-1] < avain \wedge A[x] > avain)$
  - määrittelemätön, kun $x = 1$ ja $A[1] > avain$
- *paikka2*(A[1…n], *avain*, x) :⇔
  $A[x] = avain$
  $\vee \quad (A[x] > avain \wedge (x = 1 \vee A[x-1] < avain))$
  - entä jos *avain* = 24?
  ⇒ arvo $x = n+1$ näyttää tarpeelliselta
- johtuuko arvon $x = n+1$ tarve itse tehtävästä vai ratkaisuyrityksen huonoudesta?
- koe: jos $n = 1$, niin
  - tarvitaan 2 vastausvaihtoehtoa “tähän” ja “perään” sen mukaan onko $A[1] \ge avain$
  - arvoalue $1 \le x \le n = 1$ sallii vain yhden
  - havainto yleistyy jokaiselle taulukon koolle

⇒ itse tehtävä vaatii, että arvoalueella on n+1 arvoa
- valitsemme arvoalueen $1 \le x \le n+1$

- $paikka3(A[1\ldots n], avain, x) :\Leftrightarrow$
  $A[x] = avain$
  $\vee \quad (A[x] > avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee \quad (x = n+1 \wedge A[n] < avain)$
  - jos on useita i siten, että $A[i] = avain$, ei määrittele, mikä valitaan
  - pitäisi valita pienin
- $paikka4(A[1\ldots n], avain, x) :\Leftrightarrow$
  $(A[x] = avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee \quad (A[x] > avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee \quad (x = n+1 \wedge A[n] < avain)$
  $\Leftrightarrow$
  $(A[x] \ge avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee \quad (x = n+1 \wedge A[n] < avain)$
- koska $n \ge 1$, niin $paikka4 \Leftrightarrow$
  $(A[x] \ge avain \wedge (x = 1 \vee A[x-1] < avain))$
  $\vee \quad (x = n+1 \wedge A[x-1] < avain)$
  $\Leftrightarrow$
  $(x = n+1 \vee A[x] \ge avain) \wedge (x = 1 \vee A[x-1] < avain)$
  $\Leftrightarrow$: $paikka5(A[1\ldots n], avain, x)$
  - nyt on jo aika selkeä!
  - nimen vaihto oli tarpeen, koska sievennys perustui oletukseen $n \ge 1$
- osilla on selvä tulkinta:
  - $x = n+1 \vee A[x] \ge avain \Leftrightarrow x$ on tarpeeksi suuri
  - $x = 1 \vee A[x-1] < avain \Leftrightarrow x$ ei ole liian suuri
- valitsemme: $paikka(A[1\ldots n], avain, x) :\Leftrightarrow$
  $paikka5(A[1\ldots n], avain, x) \Leftrightarrow$
  $(x = n+1 \vee A[x] \ge avain) \wedge (x = 1 \vee A[x-1] < avain)$

Havaintoja

- predikaattia ei aina ole tarkoitettu käytettäväksi kaikilla argumenttien arvoyhdistelmillä
  - muiden arvoyhdistelmien huomioon otto tekee predikaatista usein turhan monimutkaisen

$\Rightarrow$ kannattaa tehdä päätös, mille alueelle se tarkoitetaan, ja unohtaa muut arvot
  - esimerkissä $n > 0$ ja $1 \leq x \leq n+1$

- jos käyttöalue on K, predikaatit P_1 ja P_2 ovat "samanveroiset", joss
$$K \Rightarrow P_1 \leftrightarrow P_2$$
- jos haluaa kaikkialla määritellyn tilapredikaatin, käyttöalueen ulkopuolisten arvojen tuottaman totuusarvon voi kiinnittää jälkeen päin
  - esim. "**F**" kirjoittamalla $1 \leq x \leq n+1 > 1 \wedge P(x)$
  - haluttaessa rajauksen voi lopuksi sieventää $P(x)$:n "sisälle"

$\Rightarrow$ $P(x)$:n suunnittelussa ei kannata ottaa taakkaa käyttöalueen ulkopuolisista arvoista
  - käyttöalue muistettava kertoa, jos $\neq$ **T** !

- yritys kirjoittaa tilapredikaatti nostaa usein esiin kysymyksiä, joita ei aikaisemmin ole huomattu
  - esim. mitä "pitäisi olla" tarkkaan ottaen tarkoittaa?
  - esim. mikä x valitaan, jos useita vaihtoehtoja?
  - täsmällisyyden etu ja haitta!
- predikaatin käyttöalueelle osuvien määrittelemättömien tilanteiden poisto saattaa vaatia vaivaa ja huolellisuutta

- formalisointi saattaa paljastaa katteettomia ennakko-oletuksia
  - esim. “arvoalue $1 \leq x \leq n$ riittää”
  - tällaiset oletukset ovat kiusallinen virhelähde!
- formalisointi saattaa paljastaa monikäsitteisyyttä
  - monikäsitteisyys ei aina ole haitaksi!
  - silti on hyvä tietää, onko monikäsitteisyyttä

$\Rightarrow$ ylläolevat selittävät sitä käytännön havaintoa, että formaali määrittely parantaa merkittävästi lopputuotteen laatua ja lyhentää kehitysaikaa
  - tosin hinta on korkea!

- tulosta kannattaa yrittää sieventää tai muuntaa erilaisiin muotoihin
  - yksinkertaisempi predikaatti on usein parempi predikaatti
  - sievennetyt muodot saattavat tukea intuitiota alkuperäistä paremmin
  - sievennetyt muodot saattavat olla tulkittavissa toisenlaisesta intuitiosta

  $\Rightarrow$ usko predikaatin “oikein” oloon vahvistuu
- sieventämisen luotettavuus ei edellytä intuitiota
  - lakeja voi (ja kannattaa) soveltaa täysin mekaanisesti
  - olennaista, jotta lopputuloksen intuitiivisuus todella lisäisi uskoa predikaatin “oikein” oloon
- hyvin suunniteltu predikaatti on joskus kelvollinen alkuperäisen alueen ulkopuolellakin
  - esim. *paikka* “toimii” myös kun $n = 0$
  - entä *paikka4*?

Predikaatin kirjoittaminen muistuttaa monessa suhteessa ohjelmointia

- intuitiivisen idean koodausta kankealle kielelle
  - joskus ajatus joudutaan esittämään hyvin epäsuorasti
- ei yleensä onnistu ensimmäisellä yrityksellä
- erikoistapauksia on mietittävä
- on otettava huomioon semanttisia rajoituksia tyyliin “nollalla ei saa jakaa”
- “syötteiden” sallittu alue sovittava
- kannattaa pyrkiä “kauniiseen” tulokseen

Erojakin on

- predikaateilla voi laskea, ohjelmilla normaalisti ei
  $\Rightarrow$ paljon enemmän sieventämismahdollisuuksia
- tehokkuusnäkökohdista ei tarvitse välittää

$\Rightarrow$ paremmat mahdollisuudet saavuttaa ymmärrettävä lopputulos

  - ymmärrettävyydestä ei tarvitse tinkiä muiden asioiden hyväksi
  - tietenkään väittämän sisältöä ei saa muuttaa ymmärrettävyyden vuoksi!
- keinokokoelma on erilainen
  - esim. laskujärjestystä ei voi ohjata
  - esim. ohjelmoinnissa ei ole niin voimakkaita operaattoreita kuin “$\exists$”
- predikaatteja ei voi koeajaa!

- (on olemassa eräs lukuteorian kaava $\varphi(x)$ siten, että **ei voi** olla olemassa algoritmia, joka ottaa syötteekseen x ja vastaa aina ja vieläpä oikein kysymykseen, päteekö $\varphi(x)$)

Iso kysymys:

Mistä voi olla varma, että tilapredikaatti on oikein?

- viime kädessä vastaus on: **ei mistään!**
- voidaan asettaa joitakin muodollisia, yleispäteviä oikeellisuuden kriteereitä
  - tilapredikaatin arvo ei saa riippua $\perp$:n (sivu 61) ominaisuuksista, jos argumentit ovat sallitulla alueella
  - on epäilyttävää, jos tilapredikaatti tuottaa aina **F** tai aina **T**, kun argumentit ovat laillisia
  - vrt. ohjelmat
- tällaiset eivät kuitenkaan kerro, onko tilapredikaatin *merkitys* se, mitä haluttiin
- tilapredikaatin merkityksen todistaminen oikeaksi vaatisi, että käytettävissä olisi formaalissa muodossa tieto siitä, mikä on “oikea merkitys”

$\Rightarrow$ tarvittaisiin toinen formaali kaava
  - mistä voi tietää, että se on oikein?

$\Rightarrow$ ketjun viimeistä formaalia kaavaa φ ei voi todistaa oikeaksi, koska sen oikeellisuuden mitta ei ole käytettävissä formaalissa muodossa
  - jos olisi, φ ei olisikaan ketjun viimeinen

$\Rightarrow$ usko ketjun viimeisen kaavan “oikeellisuuteen” voi perustua vain intuitioon

- ohjelman spesifikaatiossa olevan tilapredikaatin tehtävä on olla osa ohjelman oikeaksi todistamisessa käytettävää oikeellisuuden mittaa

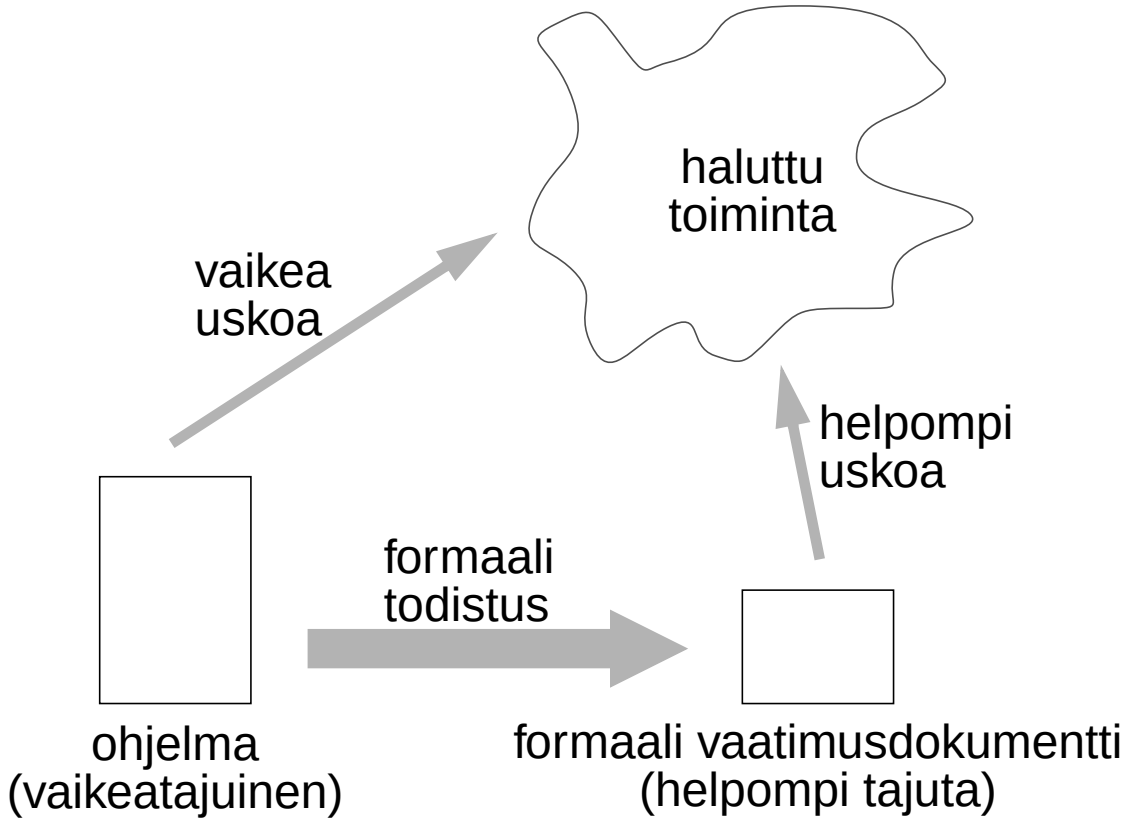
$\Rightarrow$ se on yleensä tarkoitettu ketjun viimeiseksi kaavaksi

$\Rightarrow$ sen oikeellisuutta ei voi todistaa
  – ts. ei voida todistaa sitä, että sen *merkitys* on haluttu
  – kokonaan eri asia on, että usein voidaan osoittaa, että se *on voimassa* jossain tilassa

Seurauksia

- kysymys: Jos ei voida olla varmoja, että ohjelman todistuksessa käytettävä tilapredikaatti on oikein, miten sitten voidaan olla varmoja, että ohjelma on oikein?
- vastaus: ei mitenkään!
  – ei voida todistaa, että ohjelma toimii niin kuin halutaan
  – voidaan todistaa, että ohjelma toimii esim. tilapredikaateista muodostetun formaalin vaatimusmäärittelyn mukaan
  – onko vaatimusmäärittely oikein, jää uskon varaan
- kysymys: Mitä hyötyä ohjelmien todistamisesta sitten on?

- vastaus: Sillä voidaan olennaisesti “pienentää” sitä askelta, joka on vain uskottava.



- seurauksia
  - tilapredikaatin tulee olla mahdollisimman helppotajuinen (tietenkin sillä rajoituksella, että sen tulee silti sanoa oikea asia)
  - tilapredikaatteja kannattaa “testata”

Keinoja lisätä uskoa tilapredikaatin oikeellisuuteen

- alla olevissa esimerkeissä käytetään predikaattia
  $paikka(A[1\ldots n], avain, x) \Leftrightarrow$
  $(x = n+1 \vee A[x] \geq avain) \wedge (x = 1 \vee A[x-1] < avain)$

- tarkasta, että predikaatin arvo ei voi riippua ⊥:n ominaisuuksista, jos argumentit ovat sallitulla alueella
  - esim. *paikka*:n ainoat vaaralliset operaatiot ovat $A[x]$ kun $x = n+1$ ja $A[x-1]$ kun $x = 1$
  - tällöin ympäristö määrää tuloksen
  - (mitä jos $n = 0$?)
- testaa sijoittamalla osalle argumenteista arvot, sieventämällä kaavaa ja varmistamalla, että lopputulos vastaa intuitiota
  - erityisesti kannattaa testata rajatapauksilla
  - esim. kun $n = 0$, niin $paikka \Leftrightarrow (x = 1) \vee$ **?** ·/·
  - esim. kun $n = 1$, niin $1 \le x \le 2$, joten $paikka$
    $\Leftrightarrow (x = 2 \vee A[x] \ge avain) \wedge (x = 1 \vee A[x-1] < avain)$
    $\Leftrightarrow (x = 2 \vee A[1] \ge avain) \wedge (x = 1 \vee A[1] < avain)$
    $\Leftrightarrow (A[1] < avain \rightarrow x = 2) \wedge (A[1] \ge avain \rightarrow x = 1)$
    täsmää, koska luettelee oikeat vastaukset testin "$A[1] < avain$" tuloksen funktiona
- voit myös testata sijoittamalla kaikille argumenteille arvot, ja tarkastamalla lopputulos
  - kannattaa testata sekä **F**- että **T**-tapauksilla
  - usein työläyteensä nähden tehoton testaustapa

  $\Rightarrow$ yleensä on parempi testata sieventämällä
- sievennä kaava toiseen muotoon, ja tarkasta, että tulos vastaa intuitiota
- esimerkki: $paikka(A[1\ldots n], avain, x) \Leftrightarrow$
  $x = 1 \wedge x = n+1$
  $\vee \quad (x = n+1 \wedge A[n] < avain)$
  $\vee \quad (x = 1 \wedge A[1] \ge avain)$
  $\vee \quad (A[x] \ge avain \wedge A[x-1] < avain)$
  - $x = 1 = n+1$ käsittelee oikein tapauksen $n = 0$
  - $x = n+1 \wedge A[n] < avain$ on oikea tapa käsitellä taulukon oikea reuna, jos $n > 0$

- $x = 1 \wedge A[1] \geq avain$ on oikea tapa käsitellä vasen reuna, jos $n > 0$
- $A[x] \geq avain \wedge A[x-1] < avain$ on oikein keskellä taulukkoa (määrittelemätön reunoissa, mutta silloin jokin edeltävistä tuottaa **T**)

Kannattaa käyttää hyväksi sitä, että tilapredikaateilla voi laskea.

Esimerkki: taulukon $A[1\ldots n]$ kaikki alkiot ovat eri suuria

- 1. yritys: $erisuuria1(A[1\ldots n]) :\Leftrightarrow$
  $\forall\, i, j \in \{1,\ldots,n\}: A[i] \neq A[j]$
- testataan rajatapauksella $n = 1$
  - silloin $erisuuria1(A[1\ldots n]) \Leftrightarrow A[1] \neq A[1]$
  hupsis!
- $erisuuria2(A[1\ldots n]) :\Leftrightarrow$
  $\forall\, i, j \in \{1,\ldots,n\}: (i \neq j \rightarrow A[i] \neq A[j])$
  - toimii rajatapauksissa $n = 0$ ja $n = 1$
  - ei indeksoi A:ta laittomasti
- intuition mukaan tulisi päteä
  $järjestyksessä2(A[1\ldots n]) \Leftrightarrow$
  $järjestyksessä(A[1\ldots n]) \wedge erisuuria2(A[1\ldots n])$
  tarkastetaanpa!
  - "$\Leftarrow$": helppo
  - "$järj.2(A[1\ldots n]) \Rightarrow järj.(A[1\ldots n])$": välitön
  - "$järj.2(A[1\ldots n]) \Rightarrow eris.2(A[1\ldots n])$": kun $i < j$, niin $1 \leq i < n$, joten $A[i] < A[i+1] < \ldots < A[j]$, joten $A[i] \neq A[j]$; vastaavasti, kun $j < i$; tapaus $i = j$ välitön

$\Rightarrow$ uskomme, että $erisuuria2$ on oikein
$erisuuria(A[1\ldots n]) :\Leftrightarrow erisuuria2(A[1\ldots n])$

Esimerkki tilapredikaatin kirjoittamisesta joukkojen avulla: samat alkiot

- *samat-alk*(A[1…n], B[1…n]) ~ taulukoissa A ja B on samat alkiot
- 1. yritys: *samat-alk1*(A[1…n], B[1…n]) :⇔
  $\forall\, i\,;\, 1 \le i \le n$: $\exists\, j\,;\, 1 \le j \le n$: $A[i] = B[j]$
  – sallii A = [1,1], B = [1,2]
- *samat-alk2*(A, B) :⇔
  samat-alk1(A, B) $\wedge$ *samat-alk1*(B, A)
  – sallii A = [1,1,2], B = [1,2,2]

⇒ näyttää tarpeelliselta laskea, kuinka monta kutakin alkiota on

⇒ otamme käyttöön apumerkinnän:
määrä(x, A[1…n]) := $|\{\, i \mid 1 \le i \le n \wedge A[i] = x \,\}|$

- *samat-alk3*(A, B) :⇔
  $\forall\, i\,;\, 1 \le i \le n$: *määrä*($A[i]$, A) = *määrä*($A[i]$, B)
- *samat-alk3* on epäsymmetrinen A:n ja B:n suhteen
  ⇒ voiko se olla oikein?
  – vastaoletus: symmetrinen kaava ei päde
  ⇒ $\exists\, k\,;\, 1 \le k \le n$: *määrä*($B[k]$, A) ≠ *määrä*($B[k]$, B)
  – jos *määrä*($B[k]$, A) > 0, niin $\exists\, i\,;\, 1 \le i \le n$: $B[k] = A[i]$, joten *samat-alk3* ⇒ *määrä*($B[k]$, A) = *määrä*($B[k]$, B)
  ⇒ *määrä*($B[k]$, A) = 0 ⇒ *määrä*($B[k]$, B) > 0
  ⇒ B:ssä on enemmän alkioita kuin A:ssa
  ⇒ ristiriita
  ⇒ vastaoletus on väärin
  ⇒ epäsymmetria on vain näennäistä

⇒ hyväksymme *samat-alk*(A, B) :⇔ *samat-alk3*(A, B)

Tilapredikaatin muotoileminen on joskus vaikeaa

- logiikan kieli on kankea
  - matemaatikkojen suunnittelema
  - ei suunniteltu isojen väittämien esittämiseen
  - ei suunniteltu tietokoneen luettavaksi
- toisaalta logiikka sallii omien merkintöjen lisäilyn
  ⇒ saa luvan riittää meille
  - omia merkintätapoja saa tällä kurssilla ottaa tarvittaessa käyttöön, mutta ne on määriteltävä (paitsi jos merkitys on ilmeinen)
  - esim. nyt kun *järjestyksessä* on määritelty, sitä voidaan käyttää muiden predikaattien osana
- uusia tietoabstraktioita käytettäessä on usein tarpeen ottaa käyttöön ko. abstraktion käsittelyyn sopivia merkintöjä
  - esim. jono, pino, graafi
- tilapredikaatin kirjoitusvaikeuksien syynä on usein se, että väittämän tarkkaa sisältöä kaikissa tilanteissa ei ole aikaisemmin mietitty
  - siis predikaatin vaikeuden syynä on usein se, että itse **asia** on vaikea
  - ilman väittämän formalisointia sen aukot jäävät yleensä huomaamatta
    ⇒ kiusallisia virheitä
- formalismin käytön ensimmäinen ja usein suurin hyöty tulee siitä, että formalismi pakottaa miettimään asian tarkasti

2.6 Ohjelman spesifikaatio

Ohjelman oikeellisuus

- on osoittautunut hyödylliseksi jakaa ohjelman oikeellisuusvaatimus kahteen osaan:
  - *osittainen oikeellisuus* (*partial correctness*)
  - *pysähtyvyys* eli *loppuun pääsy*
- osittainen oikeellisuus =
  ohjelma ei saa tehdä mitään väärää
  - ei saa antaa väärää vastausta
  - ei saa lopetettuaan jäädä väärään tilaan
- pysähtyvyys = ohjelman on lopulta pysähdyttävä hallitusti
  - ei saa jäädä ikuiseen silmukkaan
  - ei saa tehdä mitään kiellettyä (nollalla jako, taulukon indeksointi ohi rajojen tms.)
- seuraava ohjelma on osittain oikea, olipa spesifikaatio mikä tahansa:

```
while true do
    (* älä vain tee mitään! *)
endwhile
```

- *täysi oikeellisuus* (*total correctness*) =
  osittainen oikeellisuus + pysähtyvyys
- pysähtyvyyden vaatimus on sama kaikille ohjelmille
- osittaisen oikeellisuuden vaatimus on tapauskohtainen
  - esim. järjestämisohjelman lopussa taulukko sisältää alkuperäiset alkiot suuruusjärjestyksessä
  - esim. etsimisohjelman lopussa *i* osoittaa kysyttyä tietuetta

$\Rightarrow$ tarvitsemme esitystavan, jolla ohjelmalle asetettava osittaisen oikeellisuuden vaatimus voidaan ilmaista

Osittaisen oikeellisuuden spesifikaatio

- ohjelman osittaisen oikeellisuuden vaatimus voidaan ilmoittaa muodossa

  $\{alkuehto\}\ ohjelma\ \{loppuehto\}$

  missä *alkuehto* (*precondition*) ja *loppuehto* (*postcondition*) ovat tilapredikaatteja
- esimerkki: potenssiin korotus

  $\{\ n \geq 0\ \}$
  $x := 1$
  for $i := 1$ **to** n **do**
  $\quad x := x \cdot a$
  endfor
  $\{\ x = a^n\ \}$
- esimerkki: alustus

  $\{$ **T** $\}$
  for $i := 1$ **to** n **do**
  $\quad A[i] := 0$
  endfor
  $\{\ \forall\ i\ ;\ 1 \leq i \leq n : A[i] = 0\ \}$
- tarvittaessa otetaan käyttöön apusymboleita edustamaan arvoja, jotka eivät muutu ohjelman suorituksen aikana
  - eivät edusta ohjelman muuttujia, vaan esim. niiden alkuarvoja
  - esitetään usein alaindeksillä 0, esim. x_0, $A_0[i]$
  - ns. *haamumuuttujat* (*ghost variables*)

- esimerkiksi muuttujien arvojen vaihto

  $\{ x = x_0 \wedge y = y_0 \}$
  $t := x; x := y; y := t$
  $\{ x = y_0 \wedge y = x_0 \}$

- siis: tällä kurssilla merkintä

  $\{alkuehto\}\ ohjelma\ \{loppuehto\}$

  tarkoittaa seuraavaa:

  jos *ohjelma*:a käynnistettäessä *alkuehto* pätee,
  niin *ohjelma*:n pysähdyttyä *loppuehto* pätee

- **varoitus!** merkitys vaihtelee hieman eri kirjoittajilla
  - meidän valintamme on sama kuin Backhousen
  - esim. Gries esittää saman muodossa

    $alkuehto\ \{ohjelma\}\ loppuehto$

Minkä muuttujien arvoja saa muuttaa?

- seuraava ohjelma selvästikin täyttää muodollisen spesifikaationsa!

  $\{ n \geq 0 \}$
  $x := 1; a := 1; n := 1$
  $\{ x = a^n \}$

- muuttujien arvojen muuttaminen voidaan kieltää vaatimalla, että niiden arvot lopussa ovat samat kuin alussa:

  $\{ n = n_0 \geq 0 \wedge a = a_0 \}$
  …
  $\{ x = a^n \wedge n = n_0 \wedge a = a_0 \}$

- työlästä ongelman yleisyyteen nähden

⇒ ratkaisu: jaetaan muuttujat kahteen ryhmään sen mukaan, saako ohjelma muuttaa niiden arvoja
  – *kiinteät* muuttujat
  – tavalliset muuttujat

- miten suhtautua seuraavaan?

  { $n \geq 0$ }
  $apu := n$; $n := 0$
  … (* tässä ei kosketa *apu*:un *)
  $n := apu$
  { $x = a^n$ }

- ratkaisu
  – on vaikea (joskus mahdoton) tarkastaa, muuttaako ohjelma todella jonkin muuttujan arvoa
  – on helppo tarkastaa, onko ohjelmassa k.o. muuttujaan kohdistuvia sijoituslauseita
  ⇒ **päätös:** kiinteään muuttujaan ei saa edes yrittää sijoittaa
  – (useiden saantipolkujen ongelma)
- kiinteät muuttujat ovat eri asia kuin haamumuuttujat
  – haamumuuttujat eivät esiinny ohjelmassa, ainoastaan tilapredikaateissa
- syöte-, apu- ja tulosmuuttujat
  – syötteet annetaan syötemuuttujissa
  – tulokset palautetaan tulosmuuttujissa
  – apumuuttujat ovat välituloksia varten
- jako kiinteisiin ja tavallisiin muuttujiin on eri asia kuin jako syöte-, apu- ja tulosmuuttujiin
  – kiinteät muuttujat ovat yleensä syötemuuttujia
  – kaikki syötemuuttujat eivät ole kiinteitä: esim. taulukon järjestäminen paikallaan

- yleensä jako tavallisiin ja kiinteisiin muuttujiin oletetaan tunnetuksi ja jätetään mainitsematta
  $\Rightarrow$ ryhmille ei edes ole vakiintuneita nimiä

Osittainen oikeellisuus ja loppuun pääsy

- seuraava ohjelma on osittain oikea, olkoot P ja Q mitä tahansa:

  $\{P\}$
  while true do
  endwhile
  $\{Q\}$

$\Rightarrow$ spesifikaatio ei ole täydellinen, ellei vaadita, että ohjelma pääsee loppuun

- yleensä ohjelmien halutaan pääsevän loppuun
  – tärkeä poikkeus: reaktiiviset ohjelmat (ei käsitellä tällä kurssilla)
- ohjelman loppuun pääsemisen todistaminen vaatii usein huomattavaa lisävaivaa
  – esim.

  $x := 1; y := 1; z := 1; n := 3$
  while $x^n + y^n \neq z^n$ **do**
  if $y > 1$ **then** $x := x+1; y := y-1$
  elsif $z > 1$ **then** $z := z-1; y := x+1; x := 1$
  elsif $n > 3$ **then** $n := n-1; z := x+1; x := 1$
  else $n := x+3; x := 1$
  endif
  endwhile
  $\{ x \geq 1 \wedge y \geq 1 \wedge z \geq 1 \wedge n \geq 3 \wedge x^n + y^n = z^n \}$

$\Rightarrow$ osittainen oikeellisuus ja loppuun pääsy todistetaan yleensä erikseen

⇒ käsittelemme paljon välivaiheita, joissa loppuun pääsyä ei vaadita

⇒ tällä kurssilla on oltava tarkkana siitä, milloin loppuun pääsy vaaditaan ja milloin ei

Loppuun pääsyn spesifiointi

- ohjelman loppuun pääsy vaaditaan (tällä kurssilla) asettamalla tilapredikaatit kulmasulkeisiin:

$$\langle\, x = x_0 \wedge y = y_0 \,\rangle$$
$$t := x;\ x := y;\ y := t$$
$$\langle\, x = y_0 \wedge y = x_0 \,\rangle$$

- siis: tällä kurssilla merkintä

$$\langle \mathit{alkuehto} \rangle\ \mathit{ohjelma}\ \langle \mathit{loppuehto} \rangle$$

tarkoittaa seuraavaa:

jos *ohjelma*:a käynnistettäessä *alkuehto* pätee,
niin *ohjelma* pääsee loppuun, ja siellä *loppuehto* pätee

- **varoitus!** tämäkään merkintä ei ole vakiintunut
  - joillakin kirjoittajilla $\{P\}\ \mathit{ohjelma}\ \{Q\}$ sisältää loppuun pääsyn vaatimuksen (esim. Gries)
  - meidän tapamme: N. Francez: *Program Verification*

⇒ pelkkä loppuun pääsemisen vaatimus voidaan esittää seuraavasti:

$$\langle \mathit{alkuehto} \rangle\ \mathit{ohjelma}\ \langle \mathbf{T} \rangle$$

Yleisoletuksia

- jotta esimerkeissä, tenttivastauksissa yms. ei tarvitsisi luetella suurta joukkoa yleismääritelmiä, oletamme alla olevat, ellei toisin sanota
- muuttujat saavat arvonsa joukosta $\boldsymbol{Z}$

- merkintä $A[a \ldots y]$ tarkoittaa taulukkoa, jonka indeksialue on a, $a+1$, …, y
  - a ja y ovat kiinteitä
  - $y \geq a-1$
  - moniulotteiset taulukot esim. $A[1 \ldots n, 1 \ldots m]$
- kun taulukon rajat on kerran annettu, samaa taulukon nimeä saa käyttää ilman rajoja
- haamumuuttujan tyyppi, koko jne. määräytyvät vastaavan ohjelmamuuttujan vastaavista
  - esim. jos $A[1 \ldots n]$, niin $A = A_0$ tarkoittaa, että A_0:n indeksialue on 1, …, n, ja $\forall\ i\ ;\ 1 \leq i \leq n$: $A_0[i] = A[i]$

Spesifiointiesimerkki: taulukon $A[1 \ldots n]$ järjestäminen

- 1. yritys:

  $\langle$ **T** $\rangle$
  järjestä
  $\langle$ *järjestyksessä*(A) $\rangle$

- pulma: sallii

  $\langle$ **T** $\rangle$
  for i := 1 **to** n **do** $A[i]$:= 0 **endfor**
  $\langle$ *järjestyksessä*(A) $\rangle$

- ratkaisu:

  $\langle\ A = A_0\ \rangle$
  järjestä
  $\langle$ *järjestyksessä*$(A) \wedge$ *samat-alk*(A, A_0) $\rangle$

Kuinka vakava alkuperäisen spesifikaation puute on?

- usein järjestämisalgoritmin ainoa taulukkoa muuttava operaatio on kahden alkion vaihto

$\Rightarrow$ *samat-alk*(A, A_0) on helppo nähdä voimassa olevaksi

⇒ todistuksen pääpaino keskittyy väittämän *järjestyksessä*(A) todistamiseen

- joskus *samat-alk*(A, A_0) ei kuitenkaan ole ilmeinen
  - alla olevassa esimerkissä *samat-alk*(B, A_0)

```
COUNTING-SORT(A[1…n], B[1…n], M)
{ ∀ i ; 1 ≤ i ≤ n: 0 ≤ A[i] ≤ M }
for k := 0 to M do C[k] := 0 endfor
for i := 1 to n do
    C[A[i].avain] := C[A[i].avain] + 1
endfor
(* nyt C[k] tietää kuinka monen alkion avain = k *)
for k := 1 to M do C[k] := C[k] + C[k–1] endfor
(* nyt C[k] tietää kuinka monen alkion avain ≤ k *)
for i := n downto 1 do
    B[C[A[i].avain]] := A[i]
    C[A[i].avain] := C[A[i].avain] – 1
endfor
```

⇒ virheen vakavuus riippuu spesifikaation käyttötarkoituksesta
  - jos se annetaan ilkeämieliselle lukijalle (amerikkalainen lakimies), sen on oltava oikein
  - kun sitä käytetään tilanteessa, jossa puuttuva osa on kaikkien mielestä silmin nähden totta, virhe ei ole vakava

- vrt. spesifikaatio

  “saat jälkiruokaa sitten kun lautanen on tyhjä”

- on parempi esittää vaatimus *samat-alk*(A, A_0) vaikka epäformaalisti kuin jättää se kokonaan pois

Esimerkki: puolitushaku spesifikaatioineen

- $A[1\ldots n]$ ja *avain* kiinteitä

$\langle\ \forall\ i\ ;\ 1 \le i \le n-1 \colon A[i] \le A[i+1]\ \rangle$
$a := 1;\ y := n+1$
while $a < y$ **do**
    $v := \lfloor\ (a+y) / 2\ \rfloor$
    if $A[v] <$ *avain* **then** $a := v+1$
    else $y := v$
    endif
endwhile
$\langle\ (a = n+1 \lor A[a] \ge \mathit{avain}) \land$
  $(a = 1 \lor A[a-1] < \mathit{avain})\ \rangle$

Esimerkki: binääripotenssi spesifikaatioineen

- a ja n kiinteitä
- x, a ja b jotain lukutyyppiä, esimerkiksi $\boldsymbol{R}$

$\langle\ n \ge 0\ \rangle$
$i := n;\ b := a;\ x := 1$
while $i > 0$ **do**
    if $i \bmod 2 = 1$ **then** $x := b \cdot x$ **endif**
    $i := \lfloor\ i / 2\ \rfloor;\ b := b \cdot b$
endwhile
$\langle\ x = a^n\ \rangle$

Esimerkki: suurimman 0-neliön etsiminen matriisista, jonka alkiot ovat 0:ia ja 1:iä

- aputilapredikaatti:

  $nollaneliö(\ i, j, k, A[1\ldots n, 1\ldots m]\ ) :\Leftrightarrow$
  $i+k-1 \le n \wedge j+k-1 \le m \wedge$
  $\forall\ h\ ;\ i \le h \le i+k-1 \colon \forall\ l\ ;\ j \le l \le j+k-1 \colon A[h, l] = 0$

  – käyttöalue: $1 \le i \le n$, $1 \le j \le m$, $k \ge 0$
  – miksi tarvitaan ehto $i+k-1 \le n \wedge j+k-1 \le m$?
- A kiinteä
- m ja n ovat kiinteitä ilman eri mainintaa yleisoletuksen vuoksi
- vastauksen
  – se nurkka, jonka koordinaatit ovat pienimmät on kohdassa (i, j)
  – koko on k

  $\langle\ \forall\ i\ ;\ 1 \le i \le n \colon \forall\ j\ ;\ 1 \le j \le m \colon A[i, j] \in \{0,1\}\ \rangle$
  etsi_isoin_nollaneliö
  $\langle\ 1 \le i \le n \wedge 1 \le j \le m \wedge k \ge 0 \wedge$
  $nollaneliö(\ i, j, k, A[1\ldots n, 1\ldots m]\ ) \wedge$
  $\forall\ i\ ;\ 1 \le i \le n \colon \forall\ j\ ;\ 1 \le j \le m \colon$
  $\forall\ h\ ;\ h > k \colon \neg nollaneliö(\ i, j, h, A[1\ldots n, 1\ldots m]\ ))$
  $\rangle$

Esimerkki: kurssin välittömien ja välillisten esitietojen läpikäynti

- kaikkien kurssien joukko on $Kurssit$
- kurssin k välittömät esitiedot ovat joukko
  $välitt\text{-}esit(k)$
- otamme käyttöön merkintöjä ($k, g \in Kurssit$)
  - $k \rightarrow g \;:\Leftrightarrow\; g \in välitt\text{-}esit(k)$
  - $k \rightarrow^{+} g \;:\Leftrightarrow\; \exists\, n \in \mathbf{N}: n > 0 \wedge \exists\, k_0, k_1, \ldots, k_n:$
    $k = k_0 \wedge k_n = g \wedge \forall\, i\, ;\, 1 \le i \le n: k_{i-1} \rightarrow k_i$
- algoritmi spesifikaatioineen
  - $Kurssit$, $välitt\text{-}esit$ ja k_0 kiinteitä
  - muuttujien k, g ja k_0 tyyppi on $Kurssit$
  - $kesken$, $löydetyt$ ja $välitt\text{-}esit$ ovat tyyppiä $2^{Kurssit}$, siis joukko kursseja

```
kesken := {k0}; löydetyt := Ø
while kesken ≠ Ø do
    valitse-jokin k ∈ kesken
    forall g ∈ välitt-esit(k) do
        if g ∉ löydetyt then
            löydetyt := löydetyt ∪ {g}
            if g ≠ k0 then kesken := kesken∪{g} endif
        endif
    endfor
    kesken := kesken – {k}
endwhile
⟨ löydetyt = { k ∈ Kurssit | k0 →+ k } ⟩
```

- johtopäätös: monimutkaisten tai abstraktien rakenteiden parissa puuhattaessa on usein tarpeen ottaa käyttöön alakohtaisia merkintöjä
- (vastaavasti todistuksissa saatetaan joutua hyödyntämään alakohtaisia tuloksia)

3 KÄSITTEIDEN MÄÄRITTELEMINEN JOUKKOJEN AVULLA

Tämän luvun tavoitteena on esitellä keinoja, joilla asioita tai käsitteitä voi määritellä joukkojen avulla

Luvussa

- esitellään joukoilla tapahtuvan määrittelemisen idea
- annetaan hyvän määritelmän tarkastuslista
- esitetään muitakin määrittelemisen avuksi tarkoitettuja suosituksia ja havaintoja
- tunnistetaan neljä erilaista täsmällisyysluokkaa

Koko luvun ajan käytetään esimerkkinä graafeja ja niihin liittyviä käsitteitä

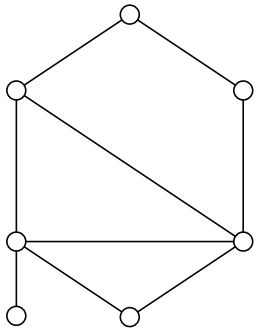
Myöhemmin luvun periaatteita käytetään uusien käsitteiden määritelemiseen

- esimerkiksi äärelliset automaatit

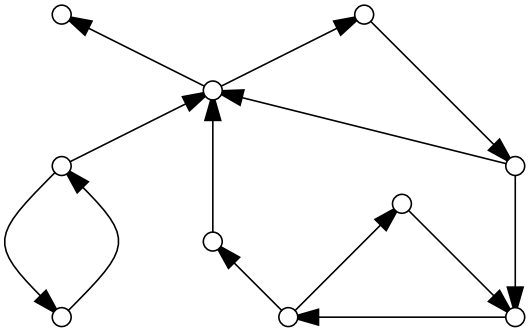
Esimerkki alkaa: *graafin* (*graph*) määrittely: graafin intuitio

- graafi on eräänlainen verkko
  - osat:
    solmut (*vertex*, monikko *vertices*) ja
    kaaret (*edge*)
- graafi voi olla
  - *suuntaamaton* (*undirected*): kaaret viivoja
  - *suunnattu* (*directed*): kaaret nuolia

- esimerkkejä:



G_1

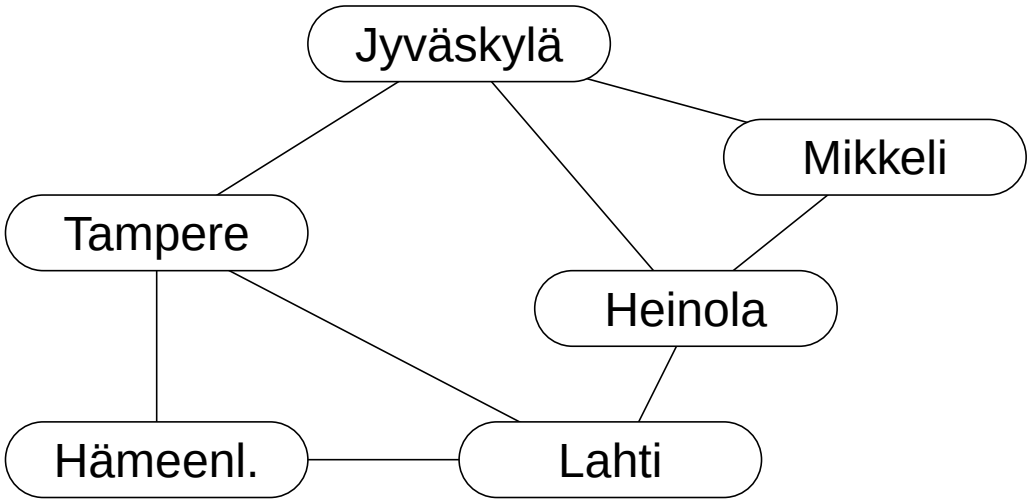


G_2

  - G_1 on suuntaamaton ja G_2 on suunnattu graafi

Graafeilla ja niiden kaltaisilla käsitteillä mallitetaan monenlaisia kohteita tietokonekäsittelyä varten

- maantieverkko
- kurssien väliset riippuvuudet
- tilakone (joka puolestaan on esimerkiksi hissin ohjausohjelmiston malli)
- äärellinen automaatti
  - luku 4
- …

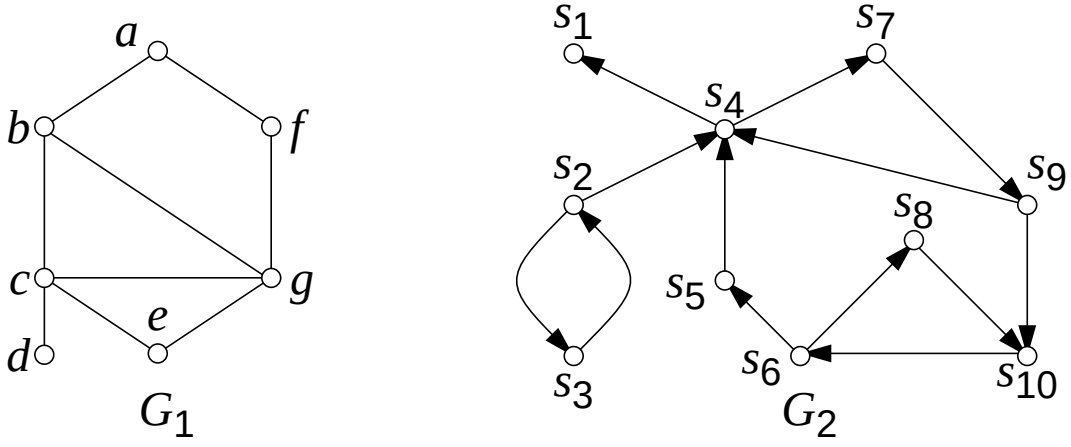


Graafin formaalin määritelmän muodostaminen

- graafin yksilöimiseksi on jollain tavalla ilmoitettava sen solmut ja kaaret
  - tai esittämiseksi tietokoneessa

⇒ koko graafi G on pari (V, E), missä V on solmujen ja E on kaarien joukko
  - (melkein yhtä hyvin olisimme voineet sopia, että kaarien joukko ilmoitetaan ensin)

- annamme alkajaisiksi solmuille nimet



  - nyt G_1:n ja G_2:n solmujen joukot voidaan ilmoittaa:
    $V_1 = \{ a, b, c, d, e, f, g \}$
    $V_2 = \{ s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10} \}$

- annetaanko kaarillekin nimet?
  - niinkin voi ja joskus kannattaa tehdä
  - nyt on kuitenkin helpompaa tehdä toisin

- suunnatun graafin kaaressa on olennaista vain kaksi asiaa: mistä se alkaa ja mihin se päättyy

⇒ kaari voidaan määritellä ilmoittamalla sen päät parina muodossa

(alkupää, loppupää)

- taas valinta “alkupää ensin” on mielivaltainen, mutta kun se on kerran tehty, se koskee kaikkien suunnattujen graafien kaikkia kaaria
- vastaavasti suuntaamattoman graafin kaari voidaan määritellä 2-alkioisena joukkona { pää, toinen_pää }
  - alkioiden ilmoitusjärjestyksellä ei ole väliä
- esimerkiksi
  - $\{a, b\}$ eli $\{b, a\}$ on G_1:n kaari
  - $\{a, c\}$ ei ole G_1:n kaari
  - (s_9, s_4) on G_2:n kaari
  - (s_1, s_7) ja (s_4, s_9) eivät ole G_2:n kaaria
- kun näin tehdään, niin
  - suunnatun graafin kaarten joukko on solmujen joukon neliön osajoukko

    $$E \subseteq V \times V$$

  - suuntaamattoman graafin kaarten joukko on solmujen joukon 2-alkioisten osajoukkojen joukon osajoukko

    $$E \subseteq \{ X \mid X \subseteq V \wedge |X| = 2 \}$$
- toisaalta, jos on annettu solmujen joukko V ja jokin joukko E siten, että $E \subseteq V \times V$ tai $E \subseteq \{ X \mid X \ldots = 2 \}$, voimme aina tulkita jokaisen E:n alkion V:n alkioita yhdistäväksi kaareksi

$\Rightarrow$ graafin formaaliksi määritelmäksi voidaan sopia seuraava:

> *Suunnattu graafi* (*directed graph*) on pari (V, E), missä V on jokin joukko ja $E \subseteq V \times V$.
>
> *Suuntaamaton graafi* (*undirected graph*) on pari (V, E), missä V on jokin joukko ja $E \subseteq \{ X \mid X \subseteq V \wedge |X| = 2 \}$.

- nyt näkyy, että järjestys (V, E) on luontevampi kuin (E, V)
  - E:n määritelmä viittaa V:hen muttei toisinpäin

Graafin formaalin määritelmän suhde intuitiiviseen graafin käsitteeseen

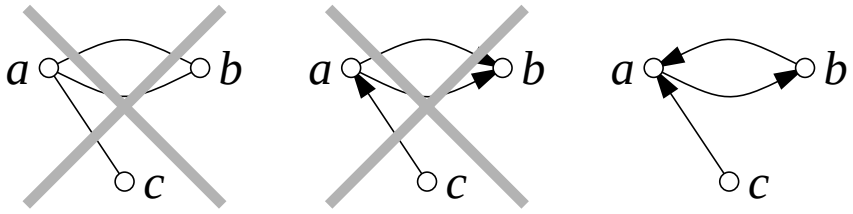
- graafin formaalia määritelmää **tulkitaan** seuraavasti:
  - V:n alkiot ovat solmuja
  - E:n alkiot ovat kaaria
  - solmusta v_1 on kaari solmuun v_2, jos ja vain jos $(v_1, v_2) \in E$ tai $\{v_1, v_2\} \in E$
- nyt esimerkiksi $G_1 = (V_1, E_1)$, missä
  $V_1 = \{ a, b, c, d, e, f, g \}$
  $E_1 = \{ \{a, b\}, \{a, f\}, \{b, c\}, \{b, g\}, \{c, d\}, \{c, e\}, \{c, g\}, \{e, g\}, \{f, g\} \}$

Graafin määritelmän seurauksia

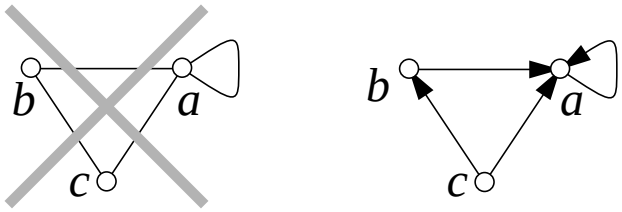
- formaali määritelmä kertoo tarkasti, mikä on ja mikä ei ole graafi
- esimerkiksi yksinäinen solmu on graafi:
  - esim. $V = \{a\}$, $E = \emptyset \subseteq V \times V$

  - myös joukko erillisiä solmuja on graafi:
    esim. $V = \{ a, b, c, d \}$, $E = \emptyset \subseteq V \times V$

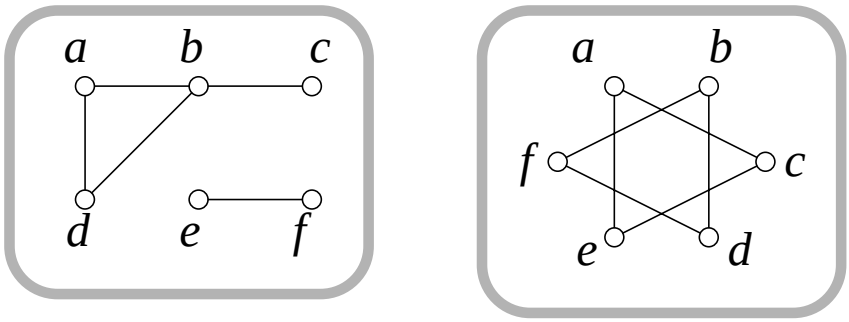
- graafissa ei voi olla kahta (saman suuntaista) kaarta samojen solmujen välillä
  - alkio (a, b) tai $\{a, b\}$ voi olla joukossa E enintään kerran
  - suunnatussa graafissa sallitaan eri suuntaiset kaaret samojen solmujen välillä: $(a, b) \neq (b, a)$



- suuntaamattomassa graafissa ei voi, mutta suunnatussa voi olla kaari solmusta itseensä
  - $|\{a, a\}| = 1 \neq 2$
  - jos $a \in V$, niin $(a, a) \in V \times V$



- graafin ei tarvitse olla *kytketty*: näennäisesti monta graafia voi olla yhden ja saman graafin osia



- formaali määritelmä on kyllä tarkka, mutta
  - sen sisältö ei ole aina ilmeinen
  - sen sisältö ei ole aina haluttu!

Mitä teimme?

- määrittelimme formaalisti graafin *käsitteen*
- vrt. “henkilötietue”
- vrt. olio-ohjelmoinnin luokka

```
class graafi{
   solmu_joukko V;
   joukon_neliö< solmu_joukko > E;
public:
   ...
};
```

  - esimerkki ei tarkoita, että graafit kannattaa toteuttaa näin
  - esimerkki tarkoittaa, että tapa, jolla kokonaisuus kootaan osistaan on molemmissa samankaltainen
  - tehokkaan toteutuksen voi määritellä joukko-opillisesti ja sitten analysoida sen ominaisuuksia ja osoittaa että se toteuttaa halutun käsitteen
- määritelmän myötä löimme tarkasti lukkoon graafin käsitteen rajat — järkevästi tai ei
- mikä tahansa yksittäinen graafi voidaan nyt määritellä tai ilmoittaa antamalla sen solmujen ja kaarien joukot
  - vrt. “Muttinen, Matti, opiskelija, 290281–123L”
  - vrt. olio-ohjelmoinnin instanssi
- graafin määritelmä ei kerro, mikä “tekee graafista graafin”
  - V:n ja E:n alkioiden tulkinta solmuiksi ja kaariksi ei ole osa formaalia määritelmää

- määritelmä kertoo vain
  - mitä tietoja on annettava *yksittäisen* graafin spesifioimiseksi
  - minkälaisia muodollisia rajoituksia graafin eri osilla on
- tiedot: vrt. instanssimuuttujat
- muodolliset rajoitukset:
  - luokan jäsenmuuttujien tyyppimääritelmät esittävät osan
  - loppu on vailla suoraa vastinetta, mutta on ainakin osittain esitettävissä pysyväisväittämillä — asiaan palataan
- jatkossa määritellään graafeihin liittyviä oheiskäsitteitä ja graafioperaatioita noudattaen johdonmukaisesti tulkintaa V = solmut, E = kaaret
  $\Rightarrow$ pikkuhiljaa muodostuu graafeja koskeva *teoria*, jossa voi mm. lausua ja todistaa graafeja koskevia tuloksia
  - V:n ja E:n tulkinta ei ole osa näitäkään määritelmiä
- vertaa olio-ohjelmointi: luokalla tekee jotakin vasta kun sille on määritelty operaatioita
  - myös muut kuin jäsenfunktiot kelpaavat tässä

Formaalin määritelmän ja todellisuuden suhteesta

- matematiikassa formaali määritelmä on tarkoitettu olemaan osa formaalia teoriaa
- teorian pitää koossa sen sisäinen johdonmukaisuus
  - teoria on rakennettava formaaliksi peliksi, vrt. sanan “formaali” esittely luvussa 2.2

- teoria puhuu graafeista vain jos haluamme sen niin tulkita
  - V:n ja E:n tulkinta
  - teoria tuottaa graafeja koskevia “oikeita” tuloksia, koska sen peruskäsitteet vastaavat “oikeita” graafeja
- teoria ei ole se todellisuus, jota se esittää, vaan sen *malli* (ja sittenkin vain jos se halutaan tulkita niin)
  - teorian tulokset ovat “absoluuttisen tosia” teorian sisällä, mutta vain siellä
- mallin vastaavuus todellisuuden kanssa on aina epäformaali uskon asia!
- **huom!** sanaa “malli” käytetään logiikassa päinvastaisessa merkityksessä
  - yllä “malli” on samankaltaisessa merkityksessä kuin esim. oliomallinnuksessa
  - logiikassa “teorian malli” on maailma, jossa teorian aksioomat pätevät

Formaalin teorian ja käytännön spesifioinnin suhteesta

- usein matemaatikot formalisoivat kaiken olennaisen teorioissaan
- käytännön spesifiointityössä kannattaa rakentaa “sekateoria”, missä (yleensä suuri) osa asioista on jätetty vain ihmisten ymmärrettävään muotoon
- formalisoida kannattaa siellä, missä on suurin monikäsitteisyyden tai väärin ymmärtämisen vaara

- myös sellaisia asioita kannattaa ainakin yrittää formalisoida, joiden kohdalla ajatukset eivät millään tahdo pysyä koossa
  - auttaa jäsentämään käsitteitä ja niiden välisiä suhteita
  - esim. paljastaa, että todellisuudessa onkin kaksi käsitettä, joita luultiin yhdeksi
- esimerkiksi kurssikuvauksessa tarvittaisiin erikseen opettaja ja vastuuhenkilö, vaikka he yleensä ovat sama ihminen
  - opettaja vastaa kyseisestä toteutuksesta: salien muutokset, tenttien arvostelu, …
  - vastuuhenkilö vastaa korvaavuuksista, ylimääräisten suoritusmahdollisuuksien järjestämisestä, uuden opettajan hankinnasta jos vanha lähtee, …

Suuntaamattomat graafit suunnattuina graafeina

- välttääksemme toistoa merkitsemme usein jatkossa myös suuntaamattoman graafin kaaria pareina (v_1, v_2) merkinnän $\{v_1, v_2\}$ sijaan
- täytyy vain muistaa, että suuntaamattomalle graafille (v_1, v_2) ja (v_2, v_1) ovat sama asia
  - erityisesti: $(v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$
- tämä on yleinen käytäntö (vaikka onkin tavallaan bluffia)
  - matematiikassa saa määritellä merkintöjä uusiksi
- matemaatikot harrastavat paljon tämänkaltaista
  - käytännöllistä
  - saattaa hämätä opiskelijoita

- matemaattisten merkintöjen tavoite on esittää asiat mahdollisimman selkeästi
  - siltä ei aina tunnu
  - merkinnät ovat usein selkeitä vain oikean taustan omaaville lukijoille
- ajattelutapa antaa keinon määritellä suuntaamaton graafi suunnatun erikoistapauksena

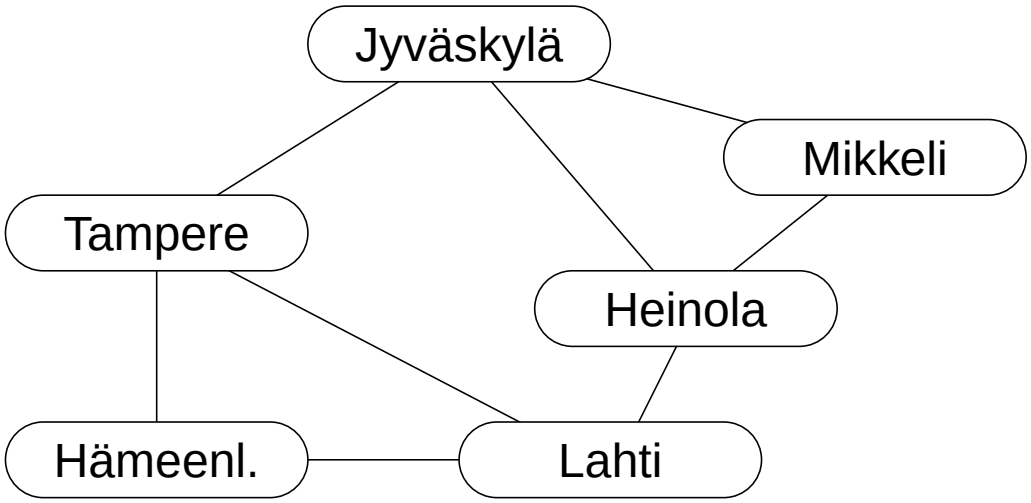
  Suuntaamaton graafi versio 2 on pari (V, E), missä V on jokin joukko, $E \subseteq V \times V$ ja $\forall\, v_1, v_2 \in V\colon (v_1, v_2) \in E \rightarrow (v_2, v_1) \in E$.
  - onko tämä yhtäpitävä aiemman suuntaamattoman graafin määritelmän kanssa?

Rajoitteet ja pysyväisväittämät

- versiossa 2 kaarten joukkoon liittyi kaksi rajoitetta
  - $E \subseteq V \times V$: tyyppi-informaatiota
  - $\forall\, v_1, v_2 \in V\colon (v_1, v_2) \in E \rightarrow (v_2, v_1) \in E$
- usein suuntaamaton graafi esitetään ohjelmassa juuri näin: suunnattu kaari kumpaankin suuntaan
- graafin muokkauksen virheiden löytämisessä on hyötyä pysyväisväittämästä, joka tarkastaa, että jokaiselle kaarelle on vastakkaissuuntainen kaari
  - voi vaatia ohjelmasilmukoita
  - usein on eduksi tarkastaa koko graafi kerralla ja vain silloin tällöin
- havaintoja
  - (lähes) samalla käsitteellä voi olla erilaisia määritelmiä
  - jokin määritelmä saattaa olla lähellä hyvää toteutusta
  - formaalin määritelmän rajoitteet tuottavat hyödyllisiä pysyväisväittämiä

Graafin polut

- eräs usein tarpeellinen käsite on solmusta toiseen kulkeva reitti tai polku
  - esim. G_2:ssa pääsee s_2:sta s_{10}:een kulkemalla seuraavat kaaret:
    (s_2, s_4), (s_4, s_7), (s_7, s_9), (s_9, s_{10})
  - esim. maantiekartan mallissa
    (Tampere, Jyväskylä) (Jyväskylä, Mikkeli)



- polun voisi määritellä jonona peräkkäisiä kaaria:

  Olkoon $G = (V, E)$ graafi. Jos $e = (v_1, v_2) \in E$, niin
  $alku(e) := v_1$ ja $loppu(e) := v_2$.
  G:n *polku* on jono $e_1 e_2 \ldots e_n$, missä $n \in \mathbf{Z}^+$,
  $\forall\, i \in \{ 1, 2, \ldots, n \}$: $e_i \in E$ ja
  $\forall\, i \in \{ 2, 3, \ldots, n \}$: $alku(e_i) = loppu(e_{i-1})$.
- tämä määritelmä on kuitenkin hieman kömpelö
  - joudutaan erikseen sanomaan, että seuraava kaari alkaa siitä, missä edellinen loppuu

⇒ yleisesti käytetään seuraavaa, helpompaa määritelmää:

> Graafin (V, E) *polku* (*path*) on jono $v_0 v_1 \ldots v_n$, missä $n \in \mathbf{N}$, $\forall\, i \in \{ 0, 1, \ldots, n \}$: $v_i \in V$ ja $\forall\, i \in \{ 1, 2, \ldots, n \}$: $(v_{i-1}, v_i) \in E$.

- nyt polku on jono solmuja, joiden välillä on kaaret
  - esim. Tampere Jyväskylä Mikkeli
- kysymys: onko tämä polun määritelmä yhtäpitävä edellisen kanssa?

Polkuihin liittyviä termejä

- suuntaamattoman graafin polku $v_0 v_1 \ldots v_n$ on solmujen v_0 ja v_n *välillä*
- suunnatun graafin polku $v_0 v_1 \ldots v_n$ *vie solmusta* v_0 *solmuun* v_n
- polun $v_0 v_1 \ldots v_n$ *pituus* on n
- esimerkissä Tampereen ja Mikkelin välisen polun pituus on 2
  ⇒ teorian käsite “pituus” ei selvästikään sovellu tulkittavaksi ajomatkaksi kilometreinä!

Vastaako tämä polun määritelmä intuitiota?

- periaate

  > jokaisella intuitiviisella polulla tulee olla tasan yksi vastaava formaali polku, ja päinvastoin

- saadaan neljä olennaista asiaa

Hyvän määritelmän tarkastuslista

1. Onko jokaiselle intuitiiviselle polulle vastine formaalin määritelmän puolella?

- ts. pystyykö formaali määritelmä esittämään kaikki intuitiiviiset polut?
- kyllä: jokainen intuitiivinen polku tuottaa jonon solmuja (ne solmut joiden kautta se kulkee)

2. Esittääkö intuitiivisen polun formaali vastine vain k.o. polun?

- ts. onko eri intuitiivisilla poluilla eri formaalit vastineet?
- kyse on siitä, tallettaako formaali määritelmä riittävästi informaatiota intuitiivisesta polusta
- kyllä: kautta kuljettujen solmujen jono riittää identifioimaan polun yksikäsitteisesti
  – jotain etua siitä, että samojen solmujen välillä samaan suuntaan voi olla vain yksi kaari!

3. Onko jokainen formaali polku myös intuitiivinen polku?

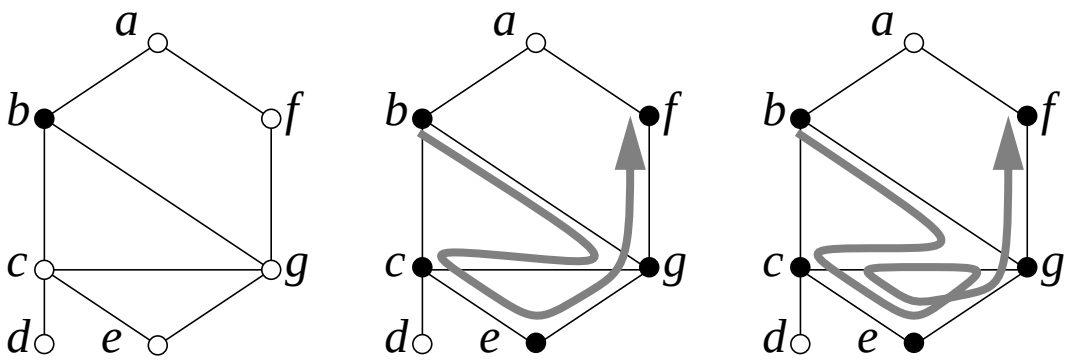
- ts. eihän formaali määritelmä kelpuuta poluksi sellaista, mitä ei saisi?
- määritelmän koko ulottuvuus voi olla vaikea hahmottaa

$\Rightarrow$ katsotaan ensin, mitä olioita määritelmä sisältää
  – yksi luonnollinen luku: n
  – n + 1 kappaletta solmuja: $v_0, v_1, \ldots, v_n$

- mitä arvoyhdistelmiä nämä voivat saada?

- koska solmujen määrä riippuu n:stä, tilanne kannattaa analysoida aloittamalla pienimmästä mahdollisesta n:n arvosta ja sitten kasvattamalla n:n arvoa kunnes riittävä ymmärrys on saavutettu
- $n \in \boldsymbol{N}$
  $\Rightarrow$ lyhimmillään polku voi koostua yhdestä solmusta, esim. b
  – haluammeko kelpuuttaa yhden solmun poluksi?
- kasvatetaan pituutta $n = 1, 2, 3, \ldots$ ja kokeillaan kakkia mahdollisia tapoja jatkaa polkua
- ehto $(v_{i-1}, v_i) \in E$ takaa, että formaalin polun voi kulkea ·/·

$\Rightarrow$ huomataan: polku voi kulkea useasti samojen solmujen kautta ja jopa useasti samat kaaret
– esim.
bgcegf ja *bgcegcegf*



$\Rightarrow$ vastaus listan kysymykseen 3 riippuu siitä, mitä hyväksymme intuitiiviseksi poluksi

4. Vastaavatko eri formaaleja polkuja eri intuitiiviset polut?

- ts. eihän formaali määritelmä ole liian yksityiskohtainen?
- kyllä vastaavat: jos yksikin solmu on eri, kyseessä on selvästi eri reitti

- liikainformaation poisto vaatii usein järeää matemaattista kalustoa
  - esim. isomorfismia tai ekvivalenssiluokkia

  $\Rightarrow$ liikainformaatiota joudutaan joskus sietämään

Listaa voi käyttää myös kahden formaalin käsitteen vertaamiseen

Yksinkertaiset polut

- haluamme uuden käsitteen: polku, joka ei kulje samojen solmujen kautta
- yritys:

  Graafin polku $v_0 v_1 \ldots v_n$ on *yksinkertainen*, jos ja vain jos $\forall\, i, j \in \{ 0, \ldots, n \}$: $v_i \neq v_j$.
- ei toimi: koska jokaisessa polussa on ainakin v_0 ja $v_0 \neq v_0$ ei päde, niin mikään polku ei täytä tätä ehtoa
- määritelmän tarkoitus on kyllä ihmislukijalle selvä: piti sanoa $\forall\, i, j \in \{ 0, \ldots, n \}$: ($i \neq j \;\rightarrow\; v_i \neq v_j$)

$\Rightarrow$ virhe ei aiheuta pahoja ongelmia esimerkiksi matemaattisissa teksteissä
  - ihmiset korjaavat sen, jopa tiedostamattaan
- virhe on vakava, jos määritelmä menee
  - sellaisenaan tietokoneelle
  - ihmiselle, joka ei tiedä asiasta tarpeeksi
- korjattu määritelmä:

  Graafin polku $v_0 v_1 \ldots v_n$ on *yksinkertainen* (*simple*), jos ja vain jos $v_i \neq v_j$ kun $0 \leq i < j \leq n$.
- jotkut kirjoittajat sallivat yksinkertaisen polun päättyvän lähtöpaikkaansa

Yksinkertaisen polun määritelmän analyysia

- nyt jokainen formaali polku on myös intuitiivinen polku, vaikka intuitiivinen polku tulkittaisiin ahtaasti
  – tarkastuslistan kohta 3
- ovatko listan kohdat 1, 2 ja 4 edelleen voimassa?
  – analysoimme lisäehdon "$v_i \neq v_j$ kun $0 \leq i < j \leq n$" vaikutukset

1. lisäehto estää tulemasta jo käytyyn solmuun uudelleen, mutta se oli tarkoituskin ·/·
2. formaaleja vastineita rajattiin pois, mutta vastaavuuksia ei lisätty eikä jäljelle jääneitä vastaavuuksia muutettu
   – ei voi synnyttää tilannetta, jossa kaksi eri vastaavuutta vie samaan kohteeseen
   ⇒ ei voi muuttaa kohtaa 2 epätodeksi ·/·
4. kuten kohta 2

Silmukat

- silmukka on vähintään yhden mittainen polku, joka päättyy lähtöpaikkaansa:

  Graafin *silmukka* (*cycle*) on polku $v_0v_1\ldots v_n$, missä $n \in \boldsymbol{Z}^+$ ja $v_n = v_0$.

- vaatimus "vähintään yhden mittainen" kätkeytyy kohtaan "$n \in \boldsymbol{Z}^+$"
- kuten polkujenkin tapauksessa, voimme kieltää silmukan leikkaamasta itseään

  Graafin silmukka $v_0v_1\ldots v_n$ on *yksinkertainen* (*simple*), jos ja vain jos $v_i \neq v_j$ kun $0 \leq i < j < n$.

- tässä pitää todellakin olla … $j < n$ (tai vaikkapa $1 \leq i < j \leq n$)
  - miksi?
  - miksei tässä tarvitse sanoa mitään v_n:stä?
- määritelmä rakentuu edellisten varaan

$\Rightarrow$ esimerkin vuoksi puramme sen auki:

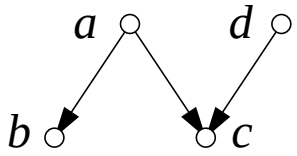
> Graafin (V, E) *yksinkertainen silmukka* (*simple cycle*) on jono $v_0 v_1 \ldots v_n$, missä
> - $n \in \boldsymbol{Z}^+$,
> - $v_i \in V$ kun $0 \leq i \leq n$,
> - $(v_{i-1}, v_i) \in E$ kun $1 \leq i \leq n$,
> - $v_n = v_0$ ja
> - $v_i \neq v_j$ kun $0 \leq i < j < n$.

Nyt voimme tarvittaessa kieltää useasta osasta koostuvat graafit

- suuntaamattomille graafeille se on helppoa:

  > Suuntaamaton graafi (V, E) on *kytketty* (*connected*), jos ja vain jos kaikille $u, v \in V$ on olemassa polku u:sta v:hen.

- suunnatuille graafeille sama määritelmä ei toimi
  - alla ei ole polkua b:stä c:hen, eikä edes b:stä a:han



- idea: lisätään jokaiselle kaarelle vastakkaiseen suuntaan kulkeva kaari, ja vaaditaan sitten, että kaikista solmuista päästään kaikkiin

Suunnattu graafi (V, E) on *vahvasti kytketty* (*strongly connected*), jos ja vain jos kaikille $u, v \in V$ on olemassa polku u:sta v:hen.

Suunnattu graafi (V, E) on *kytketty* (*connected*), jos ja vain jos (V, E') on vahvasti kytketty, missä $E' = E \cup \{ (v_1, v_2) \mid (v_2, v_1) \in E \}$.

Äskeisen analyysia

1. kytketyn graafin määritteleminen suoraan olisi ollut vaikeaa

- sen sijaan ei ollut vaikeaa edetä seuraavasti:
  graafi $\rightarrow$ polku $\rightarrow$ kytketty graafi

$\Rightarrow$ **periaate:**

> *Jos et voi määritellä suoraan sopivasti rajattua käsitettä, määrittele* ***laajempi*** *käsite, ja lisää rajaukset jälkeen päin.*

- vrt. hyvän määritelmän tarkastuslistan kohta 3

2. "yksinkertainen polku" määriteltiin lisäämällä rajoite, koska "polun" määritelmä sallii sellaista, mikä ei ole intuitiivisesti polku

$\Rightarrow$ kysymys: miksei kytkettyä graafia määritelty näin: "… on olemassa yksinkertainen polku …"?

- tämä määritelmä on loogisesti yhtäpitävä valitun kanssa
  - jokainen yksinkertainen polku on polku
    $\Rightarrow$ uudesta määritelmästä seuraa vanha
  - jokaisesta polusta saadaan yksinkertainen polku, jolla on samat päätepisteet, korvaamalla jokainen pätkä muotoa $vv_1v_2\ldots v_nv$ pelkällä v:llä
    $\Rightarrow$ vanhasta määritelmästä seuraa uusi

- vastaus: valittu määritelmä on paitsi hitusen lyhyempi, myös käyttäjälle helpompi:
  osoitettaessa graafi kytketyksi riittää löytää polku jokaisen solmuparin välille
  – ei tarvitse osoittaa, että se on yksinkertainen
- yleensä määritelmä on sitä kätevämpi käyttäjälle, mitä vähemmän se vaatii
  – laajempi sovellusalue
  – helpompi osoittaa, että oma tilanne sopii määritelmään
- vastaavasti määritelmä on sitä hankalampi määritelmään perustuvien väitteiden todistajalle, mitä vähemmän se vaatii
  – vähemmän, mihin nojata todistuksessa
- havainto: kytketyn graafin määrittelemiseen (ja moneen muuhun tehtävään) riittää pelkkä “polku”
  – se on jopa kätevämpi kuin yksinkertainen polku

⇒ **periaate:**

> *Kannattaa tarkistaa, voiko määritelmää yksinkertaistaa.*

- käsite saattaa siitä muuttua
- muuttunutkin käsite saattaa sopia alkuperäiseen tarkoitukseen
- ehkä kannattaa määritellä molemmat käsitteet tyyliin “polku” ~ “yksinkertainen polku”

3. suunnatulle graafille löytyi kaksi erilaista, luontevaa “kytketty”-käsitettä

⇒ **havainto:**

> *Käsite saattaa hajota useaksi käsitteeksi, kun sitä sovelletaan uuteen ympäristöön.*

4. kytketyn suunnatun graafin määritelmä sisältää esimerkin graafioperaation määritelmästä

- jokaiselle kaarelle lisättiin käänteinen kaari
- alkuperäistä graafia (V, E) ei muutettu, vaan operaation tulos annettiin uutena graafina
- kysymys: miksi määritelmässä ei tarvinnut varoa kaksoiskaarien muodostumista?

Määritelmissä toistui ilmaus “on olemassa polku u:sta v:hen”

$\Rightarrow$ säästämme vaivaa antamalla sille oman merkinnän

> Olkoon (V, E) suunnattu graafi, ja olkoot u, $v \in V$. Merkintä “$u \rightarrow^* v$” tarkoittaa “on olemassa polku u:sta v:hen”.

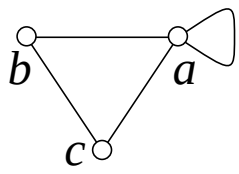
- hyvät merkinnät auttavat kaavojen hahmottamista
  - olennaiset asiat eivät kätkeydy pitkien sanallisten selitysten sekaan
- toisaalta uuden merkinnän mieleen painaminen on lukijalle rasite

$\Rightarrow$ uusia merkintätapoja kannattaa ottaa käyttöön vain, jos niistä on riittävästi hyötyä

Graafin käsitteen rajat eivät välttämättä mene kuten olisi sovelluksen kannalta järkevää

$\Rightarrow$ voi olla tarvetta muuntaa graafin käsitettä

Miten suuntaamattomassakin graafissa voisi sallia paikalliset silmukat?



- yksinkertaista: sallitaan myös yhden alkion joukot kaariksi

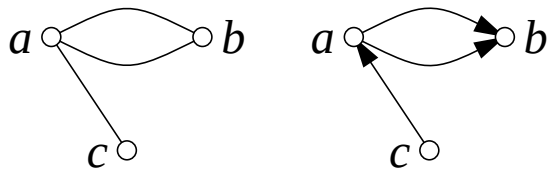
  Suuntaamaton muunnettu graafi on pari (V, E), missä V on joukko ja $E \subseteq \{ X \mid X \subseteq V \wedge 1 \leq |X| \leq 2 \}$.

- nyt $(a, a) = \{a, a\} = \{a\}$ on kaari, joss $\{a\} \in E$
- esimerkissä

  $V = \{a, b, c\}$
  $E = \{ \{a, b\}, \{a, c\}, \{b, c\}, \{a\} \}$

- lukija saattaa tarvita ylimääräisen vihjeen oivaltaakseen määritelmän intuition
  - mitä yhden solmun joukko edustaa?
  - "jos $|X| = 1$, niin X edustaa kaarta sisältämästään solmusta takaisin itseensä"
- **huom!** nimi "suuntaamaton muunnettu graafi" ei ole yleisessä käytössä

Miten sallia monta kaarta samojen solmujen välillä?



- vaihtoehto 1: lisätään funktio, joka liittää jokaiseen kaareen tiedon siitä, kuinka monena kappaleena kaari on:

  Suunnattu monigraafi (*directed multigraph*) on kolmikko (V, E, W), missä V on joukko, $E \subseteq V \times V$ ja W on funktio $E \to \boldsymbol{Z}^+$.

  Suuntaamaton monigraafi (*undirected multigraph*) on kolmikko (V, E, W), missä V on joukko, $E \subseteq \{ X \mid X \subseteq V \wedge |X| = 2 \}$, ja W on funktio $W: E \to \boldsymbol{Z}^+$.

- nyt esimerkiksi yllä on suunnattu monigraafi (V, E, W), missä
  $V = \{a, b, c\}$
  $E = \{ (a, b), (c, a) \}$
  $W((a, b)) = 2$ ja $W((c, a)) = 1$
- tämä vastaa ajattelua
  – kahden solmun välissä joko on kaaria tai sitten ei ole
  – jos niitä on, niitä on niin-ja-niin monta kpl
- usein on helpompi ajatella
  – kahden solmun välissä on nolla tai useampia kaaria

$\Rightarrow$ vaihtoehto 2:

Suunnattu monigraafi (*directed multigraph*) on pari (V, W), missä V on joukko ja
W on funktio W: $V \times V \rightarrow \boldsymbol{N}$.

Suuntaamaton monigraafi (*undirected multigraph*) on pari (V, W), missä V on joukko ja
W on funktio W: $\{ X \mid X \subseteq V \wedge |X| = 2 \} \rightarrow \boldsymbol{N}$.

- edellä oleva suunnattu monigraafi on nyt (V, W), missä $V = \{a, b, c\}$, ja W:n ilmoittaa oheinen taulukko:

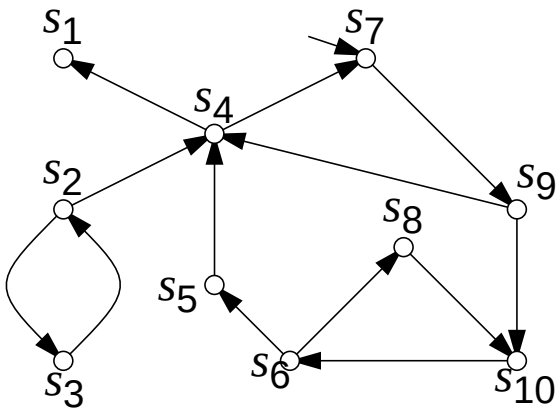
	$W(x, y)$	y = a	b	c
x	a	0	2	0
	b	0	0	0
	c	1	0	0

- menetimmekö joukon E?
  – voimmeko enää käyttää sanaa “kaari” teoriassa?

- ratkaisu: määritellään se erikseen

  Suunnatun monigraafin (V, W) *kaarten* joukko on $E = \{ (u, v) \mid W(u, v) > 0 \}$.

Miten liittää graafiin alkusolmu?



- liitetään määritelmään komponentti, joka tallettaa alkusolmun
- ko. komponentilta on välttämätöntä ja riittävää vaatia, että se on solmu

  Alkusolmullinen suunnattu graafi on kolmikko $(V, E, \hat{v})$, missä V on joukko, $E \subseteq V \times V$ ja $\hat{v} \in V$.

- yllä $\hat{v} = s_7$
- usein alkusolmun merkintänä on v_0 tms., mutta silloin seuraavan kaltaisten määritelmien teko vaikeutuu:

  Graafin *polku* on jono $v_0 v_1 \ldots v_n$, missä …

Graafioperaatioita

- tähän mennessä olemme määritelleet
  - graafin käsitteen erilaisia muunnelmia
  - joitakin graafeihin liittyviä käsitteitä (polku, silmukka)
- nyt määrittelemme esimerkin vuoksi muutaman operaation, jotka muuntavat graafeja toisiksi

Suunnatun graafin kaarien kääntö takaperin

> Suunnatun graafin (V, E) *käänteisgraafi* on pari (V, E'), missä $E' = \{ (u, v) \mid (v, u) \in E \}$.

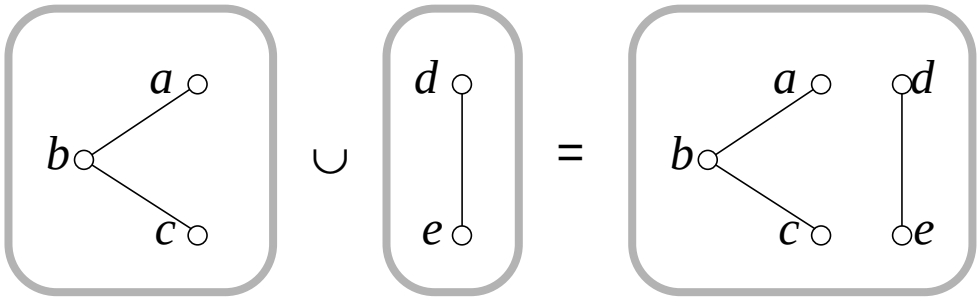
- ottaa yhden ja tuottaa yhden suunnatun graafin
- $E \subseteq V \times V$ takaa, että $E' \subseteq V \times V$
  $\Rightarrow$ tulos on varmasti suunnattu graafi

Suuntien poisto kaarista

> Suunnattua graafia (V, E) vastaava suuntaamaton graafi on (V, E'), missä
> $E' = \{ \{u, v\} \mid (u, v) \in E \wedge u \neq v \}$.

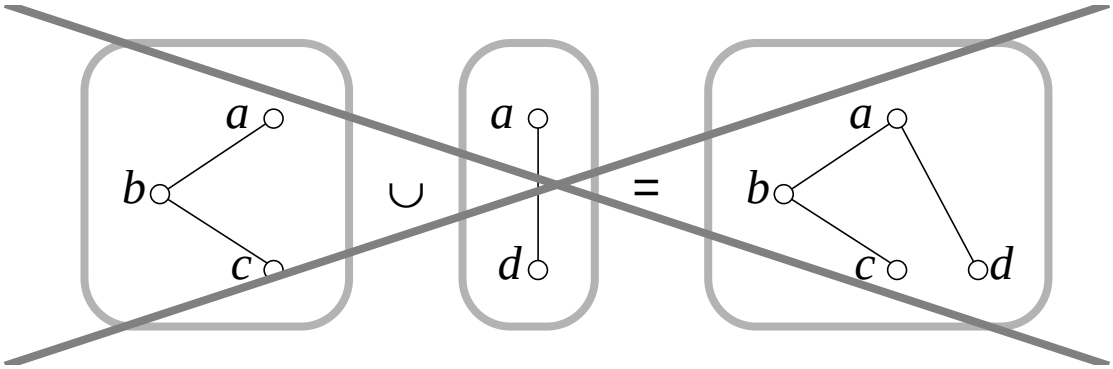
- ottaa yhden graafin ja tuottaa yhden graafin
- kysymyksiä
  - miksi mukana on vaatimus $u \neq v$?
  - mitä se aiheuttaa paikallisille silmukoille?
  - onko tämä hyvä asia?

Graafien liittäminen toisiinsa kytkemättä niitä yhteen



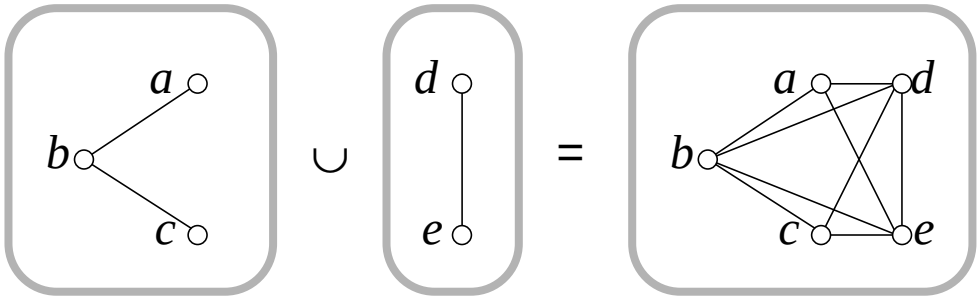
> Olkoot (V_1, E_1) ja (V_2, E_2) graafeja siten, että $V_1 \cap V_2 = \emptyset$. Niiden *erillinen unioni* on pari (V, E), missä
> $V = V_1 \cup V_2$ ja
> $E = E_1 \cup E_2$.

- tämä määritelmä vaatii, että $V_1 \cap V_2 = \emptyset$ epäselvyyksien välttämiseksi (erillisyys)



- useimmiten ei haluta, että saman nimiset solmut samaistetaan
- jos sellaista halutaan, sitä varten voidaan määritellä oma operaattori
- tuottaako tämä määritelmä varmasti graafin?
  - suunnatut: $E_1 \subseteq V_1 \times V_1 \subseteq V \times V$ ja $E_2 \subseteq V_2 \times V_2 \subseteq V \times V \Rightarrow E = E_1 \cup E_2 \subseteq V \times V$ ·/·
  - suuntaamattomat: $E_1 \subseteq \{ X \mid X \subseteq V_1 \wedge |X| = 2 \} \subseteq \{ X \mid X \subseteq V \wedge |X| = 2 \}$ ja samoin E_2, joten $E = E_1 \cup E_2 \subseteq \{ X \mid X \subseteq V \wedge |X| = 2 \}$ ·/·

Suuntaamattomien graafien liittäminen toisiinsa vetämällä kaikki yhdistävät kaaret



Olkoot (V_1, E_1) ja (V_2, E_2) suuntaamattomia graafeja siten, että $V_1 \cap V_2 = \emptyset$. Niiden *kytketty unioni* on pari (V, E), missä

$V = V_1 \cup V_2$ ja

$E = E_1 \cup E_2 \cup \{ \{v_1, v_2\} \mid v_1 \in V_1 \wedge v_2 \in V_2 \}$.

- tulos on suuntaamaton graafi sen ansiosta, että
  $V_1 \cap V_2 = \emptyset$ takaa, että $|\{v_1, v_2\}| = 2$ kun
  $v_1 \in V_1 \wedge v_2 \in V_2$
- suunnatuille graafeille voidaan kirjoittaa vastaava määritelmä ainakin kahdella tavalla:
  – yhdistävät kaaret vedetään yhteen suuntaan
  – yhdistävät kaaret vedetään molempiin suuntiin

Havaintoja

1. operaatiot eivät ole ohjelma-askelia!

- operaatiot eivät *muunna* alkuperäisiä graafeja samassa mielessä kuin esimerkiksi sijoituslause muuntaa kohteensa arvoa
- operaatiot tuottavat *kokonaan uuden graafin* alkuperäisten funktiona
- tämä on vakiintunut käytäntö matemaattisissa määritelmissä

2. huolellinen työ edellyttää, että osoitetaan, että määritelmän tuottama olio todella on (suuntaamaton, suunnattu) graafi

- usein tämä on niin selvää, että matemaatikot jättävät sen pois ilman mainintaa
- kuitenkin tätäkin kautta määritelmään tai teoriaan saattaa tulla kaiken romuttava virhe
- osoittamisesta suurin osa on yleensä helppoa rutiinia
- kannattaa nähdä osoittamisen vaiva, sillä siten vältetään monia mahdollisesti kalliita virheitä

Esimerkki

poista_solmu(V, E, v) = (V', E)

missä (V, E) on suunnattu graafi, ja $V' = V - \{v\}$

- onko tulos suunnattu graafi?
  - vaaditaan vain $E \subseteq V' \times V'$
  - $\Rightarrow$ osoitettava, että jos $(u, w) \in E$, niin $u \in V'$ ja $w \in V'$
  - ei onnistu, jos u tai w voi olla v

$\Rightarrow$ tarkastus kiinnitti huomion siihen, että jos poistetaan solmu, voivat siihen kytkeytyneet kaaret jäädä “vaille toista päätä”
  - vrt. ohjelmoinnissa hävitettyyn olioon osoittava osoitin

Määritelmän täsmällisyystasot

- määrittelyyn liittyy kaksi osaa
  - *syntaksi*: mitkä tekstit, kaaviot yms. tarkoittavat ylipäänsä mitään
  - *semantiikka*: mitä syntaktisesti oikeat tekstit ym. tarkoittavat
- kumpikin erikseen voi olla formaali tai epäformaali
  - esim. ohjelmointikielten syntaksi on yleensä määritelty formaalisti ja semantiikka epäformaalisti
  - esim. matemaattisten merkintöjen kohdalla tilanne on yleensä päinvastoin
- tietokoneella käsittely edellyttää, että ainakin syntaksi on määritelty formaalisti (ainakin melkein …)
- ihmiselle tarkoitetussa määritelmässä saa käytännössä olla avoimia kohtia ja jopa pikkuvirheitä
  - $\Rightarrow$ matemaattisista teksteistä löytyvät määritelmät ovat monesti muodollisesti puutteellisia
  - kirjoittaja on kertonut vain lukijoille oleellisen

- tietokoneen (ja lakimiehen?) luettavaksi tarkoitetussa määritelmässä on kaikki pikkuasiatkin oltava oikein
  ⇒ määritelmän kirjoittaminen muistuttaa ohjelmointia
  – ns. *formaaleissa menetelmissä* on tarkoitus noudattaa tätä täsmällisyystasoa
- syntaksin ja semantiikan täsmällisyystasojen mukaan saadaan seuraava kaavio:

		syntaksi	
		epätäsmällinen	täsmällinen
semantiikka	epätäsmällinen	luonnollinen kieli	ohjelmointikielet
	täsmällinen	matematiikka	formaalit menetelmät

⇒ voimme tunnistaa neljä täsmällisyysluokkaa:

4. luonnollinen kieli
   – käsite on olemassa vain intuitiivisesti
   – käsitteen tarkat rajat epäselviä
   – käsite voi olla sisäisesti ristiriitainen
2. matematiikka
   – käsite on täsmennetty
   – täsmennetty määritelmä on esitetty piittaamatta esitystavan muotoseikoista: lukijan oletetaan tulkitsevan tekstiä “hyväntahtoisesti”
3. tietokoneohjelmat
   – käsitteen sisältö ei ole tärkeä
   – esitystavan muotoseikat ovat tärkeitä
1. formaalit menetelmät
   – sekä käsitteen sisältö että esitystapa ovat tärkeitä

- mitä pienempi numero, sitä suurempi täsmällisyys
- asetin matematiikan tietokoneohjelmien edelle siksi, että ihmisten toiminnassa ideoiden täsmällisyys on olennaisempaa kuin esitystavan täsmällisyys
- muotoseikoiltaan virheellinen, ideoiltaan oikea matematiikka on hyödyllisempi kuin muotoseikoiltaan oikea, ideoiltaan virheellinen tietokoneohjelma

4 LAUSEKKEET

Lausekkeita (*expression*) löytyy joka puolelta matematiikkaa ja ohjelmointia

- aritmeettiset lausekkeet, esimerkiksi $-1 + 2 \cdot (x+3)$
- Boolen lausekkeet, esimerkiksi $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$
- joukko-opin lausekkeet, esimerkiksi $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$
- lausekkeet ohjelmointikielissä C ja C++, esimerkiksi

```
cout << ( luku[i1]*1000 + opisk[i1]/2 )
        / opisk[i1]
```

- ns. säännölliset lausekkeet tietojenkäsittelyteoriassa (luku 5.1)

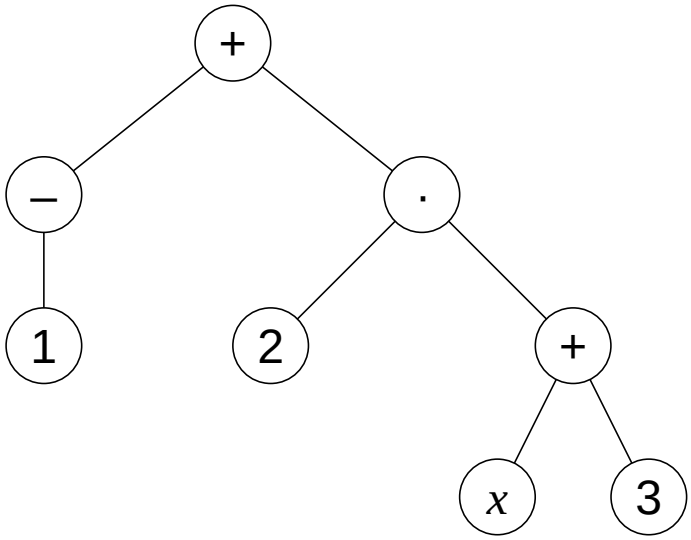
Tässä luvussa

- esitellään lausekkeisiin liittyvää käsitteistöä
- annetaan esimerkki lausekkeiden käsittelystä tietokoneohjelmassa

4.1 Lausekkeen rakenne ja tehtävä

Lausekkeen tehtävä

- lauseke ilmaisee puumaisen rakenteen tekstimuodossa
- esimerkki: $-1 + 2 \cdot (x+3)$



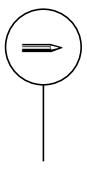
- kutsumme ko. puuta *lausekepuuksi*

Lausekkeen osat lausekepuussa

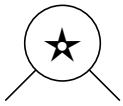
- puun lehtien paikalla on tilanteesta riippuen vakiosymboleita, muuttujasymboleita jne.
- puun muita solmuja vastaavat *operaattorit*
- puun rakennetta voi ohjata suluilla “(” ja “)”
- operaattorisolmusta roikkuvat alipuut (tai niiden tekstuaalinen esitys osana lauseketta) ovat ko. operaattorin *operandit*

Operaattorien lajeja, osa 1

- *unaari*operaattorit: yksi operandi
  - esim. lukujen etumerkit “+” ja “–”
  - esim. looginen negaatio “¬”
  - esim. C ja C++ “++” sekä osoitteen otto “&”



- *binääri*operaattorit: kaksi operandia
  - esim. lukujen yhteen- ja jakolasku “+” ja “/”
  - esim. looginen konjunktio eli “ja” “∧”
  - esim. C ja C++ “<<”, “+=”, “=” ja “`->`”



- *ternääri*operaattorit: kolme operandia
  - esim. C ja C++ “`? :`” kuten `x < 0 ? -x : x`
- vieläkin useampioperandisia operaattoreita voi määritellä, jos tarpeen

Operaattorien lajeja, osa 2

operaattori on	jos se kirjoitetaan operandiensa
prefix (*etuliite*)	eteen
infix (*sisäliite*)	väliin
postfix (*jälkiliite*?)	perään

- useimmat unaarioperaattorit ovat prefix:
  - esim. –1, `!onnistui`, `++i`
  - poikkeus: C ja C++ “++” ja “`--`” voivat olla myös postfix: `i++`
- binäärioperaattorit ovat tavallisesti infix
  - esim. 3 – 2, `n % taulukon_koko`
  - poikkeus: Scheme `( - 3 2 )`

Sitovuustaso- eli *presedenssi*säännöt

- sulkujen tarpeen vähentämiseksi on sovittu, että jotkin operaattorit *sitovat voimakkaammin* kuin toiset
  - *korkeampi sitovuustaso* = sitoo voimakkaammin
  - esim. 1 + 2·3 lasketaan
    1 + (2·3) = 1 + 6 = 7
    eikä vasemmalta oikealle
    (1 + 2)·3 = 3·3 = 9
  - (ainakin Smalltalk on poikkeus tästä esimerkistä!)

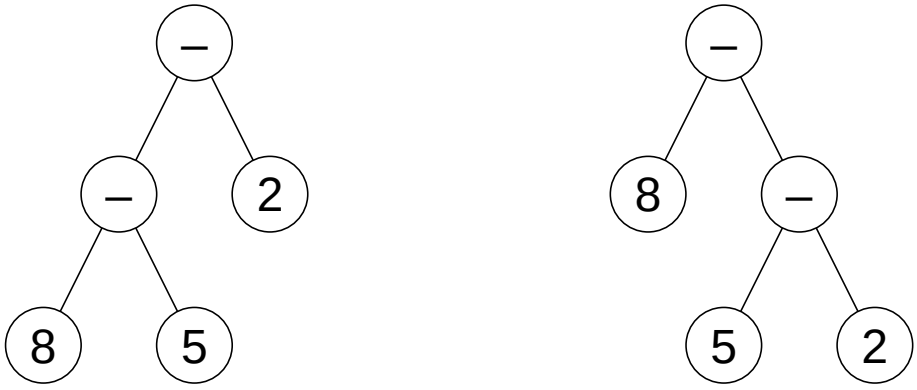
- saman osalausekkeen eri puolilla olevista operaattoreista lasketaan ensin se, joka sitoo voimakkaammin
  ⇒ “vaikutusalueena” pienempi osa lauseketta
  – esim. $-1 + \underbrace{2 \cdot \underbrace{(x+3)}}} - 4$
- esimerkiksi koulumatematiikassa käytetään seuraavia sitovuustasoja:

operaattori	sitovuustaso
etumerkki-“+” ja -“–”	korkein
“·” ja “/”	
binääri-“+” ja -“–”	matalin

- esimerkiksi C++:n operaattoreilla on kaikkiaan 18 eri sitovuustasoa
  – ylinnä binääri- ja unaari “`::`”
  – alinna “`,`”
  – etumerkki-“+” korkeammalla kuin “%” korkeammalla kuin binääri “+”
  – postfix-“++” korkeammalla kuin prefix-“++”
- sitovuustasojen kanssa kannattaa olla tarkkana!
  – esim. “`status_bit & 0xF == 5`”

Sitovuuden suunnat eli *liitännäisyyden suunnat*

- saman sitovuustason tapauksessa pitää olla määriteltynä, eteneekö laskenta vasemmalta oikealle vai toisinpäin
  - esim. 8 – 5 – 2 = (8 – 5) – 2 = 3 – 2 = 1
    eikä 8 – (5 – 2) = 8 – 3 = 5



- useimmat matematiikan ja ohjelmointikielten binäärioperaattorit *sitovat vasemmalle* eli ovat *vasemmalle liitännäisiä* (*left associative*)
  - vasemmalla puolella oleva saman sitovuustason operaatio lasketaan ennen ko. operaation laskemista
  - esim. koulumatematiikan “+”, “–”, “·” ja “/”
  - esim. C ja C++ “<<” ja “->”
- myös *oikealle sitovia* (*right-associative*) binäärioperaattoreita on
  - edellisen peilikuva
  - esim. matematiikan potenssiinkorotus:
    $4^{3^2} = 4^9 = 262144$ eikä $(4^3)^2 = 64^2 = 4096$
  - esim. Fortranin (muttei usein Basicin!) potenssioperaattori “**”
  - esim. C ja C++ “+=” ja “=”

- koetulos C++-kääntäjällä:

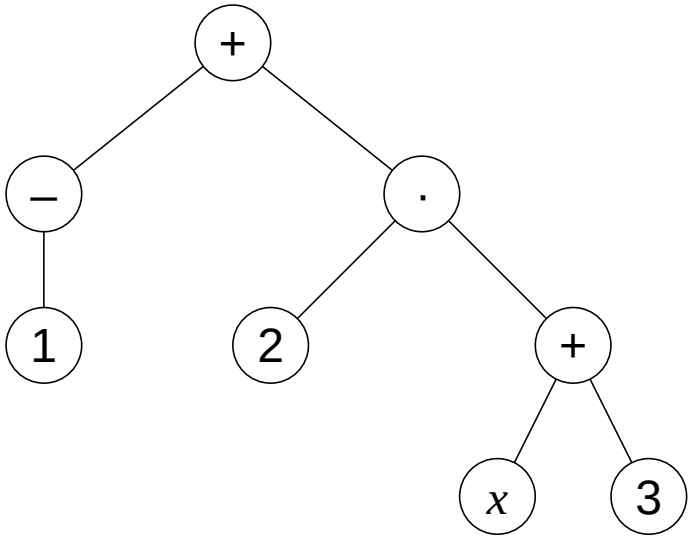
alussa			lopussa	
i	j	koodi	i	j
0	3	`i += j += 1;`	4	4
0	3	`i += (j += 1);`	4	4
0	3	`(i += j) += 1;`	4	3

⇒ myös sitovuuden suuntien kanssa kannattaa olla tarkkana!

Varoitus!

*Sitovuuden suunnat ja sitovuustasot eivät määrittele, missä järjestyksessä operaattorin **operandit** lasketaan.*

- ne määrittelevät vain lauseketta vastaavan puun rakenteen
  - alempana oleva solmu on laskettava ennen samassa haarassa ylempänä olevaa
  - eri haarojen välinen laskujärjestys ei määräydy

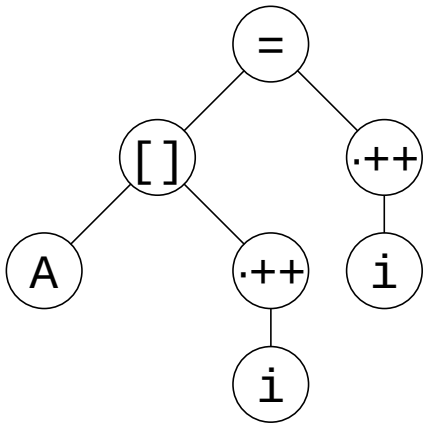


⇒ ne määrittelevät *ko. operaattorien keskinäisen* laskujärjestyksen

- esimerkiksi tapauksessa $f(x) + g(y)$ voidaan $f(x)$ laskea ennen $g(y)$:tä, sen jälkeen ja yhtäaikaa sen kanssa
- ohjelmointikielet sallivat usein monta vaihtoehtoista järjestystä
- esimerkiksi C++:n `i++`
  - lupaa, että `i`:n arvoa kasvatetaan vasta kun `i`:n arvo on otettu käyttöön *kyseisessä kohtaa* lauseketta
  - **ei lupaa**, että kasvatus tapahtuu vasta sen jälkeen kun `i`:n arvoa on käytetty muualla samassa lausekkeessa

⇒ esimerkiksi `A[i++] = i++;` voi tuottaa 3 eri tulosta

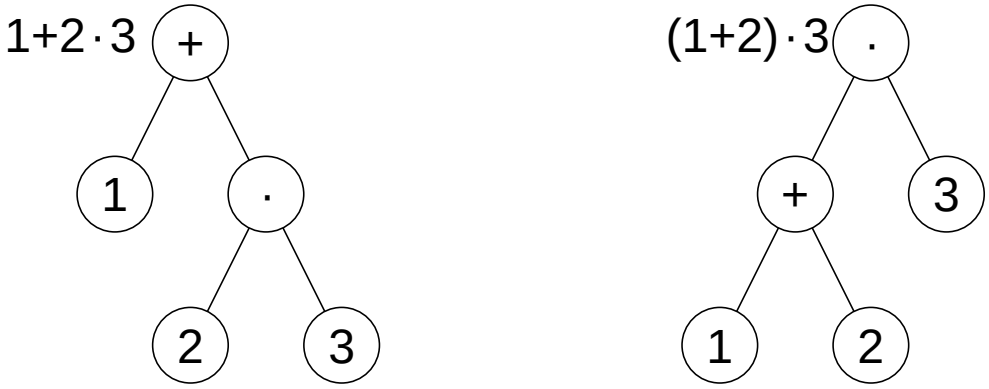
- vaiheet
  - A: A:n indeksointi
  - B: vasen `i++`
  - C: oikean p. arvotus
  - D: oikea `i++`
- luvataan vain, että A on ennen B:tä ja C ennen D:tä



järjestys	vaikutus, kun aluksi `i` = 0
ABCD	`A[0] = 1; i = 2;`
ACBD	`A[0] = 0; i = 2;`
ACDB	`A[0] = 0; i = 2;`
CABD	`A[0] = 0; i = 2;`
CADB	`A[0] = 0; i = 2;`
CDAB	`A[1] = 0; i = 2;`

Sulkujen käytöstä

- suluilla voi määrätä lasku- tai tulkintajärjestyksen haluamakseen sitovuustasoista ja sitovuuden suunnista huolimatta



- osa sitovuuden suunnista ja sitovuustasoista vaihtelee eri kirjoittajilla

$\Rightarrow$ joko

pitää määritellä mitä sääntöjä itse käyttää

tai

pitää käyttää sulkuja kaikissa epäselvissä tapauksissa

- edellinen parempi pitkissä, jälkimmäinen lyhyissä teksteissä
- kouluaritmetiikan säännöt voi olettaa tunnetuiksi

Jos kaikki operaattorit osoittaisivat vaikutusalueensa alku- ja loppukohdat suluilla tai muuten, sitovuustasoja ja sitovuuden suuntia ei tarvittaisi

- esimerkki: Lisp:

  $(\; x_1 \; x_2 \; \ldots \; x_k \;)$
- esimerkki: usein teoreettisissa logiikan kirjoissa aluksi:

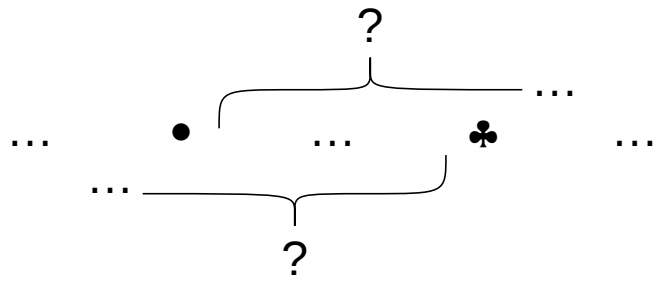
  $(\varphi \wedge \phi), \; (\varphi \vee \phi), \; (\neg \phi), \; (\forall x \, \phi), \ldots$

- esimerkki: Adan *kampamainen* rakenne
  if … **then** … **elsif** … **else** … **end if**;
  – hyvä, mutta aika vähän noudatettu käytäntö
- myöskään ns. *käänteisessä puolalaisessa notaatiossa* ei tarvita sulkuja
  – ensin operandit peräkkäin jotenkin toisistaan erotettuina, lopuksi operaattori
  ⇒ kaikki operaattorit postfix
  – mm. eräät Hewlett-Packardin valmistamat laskimet

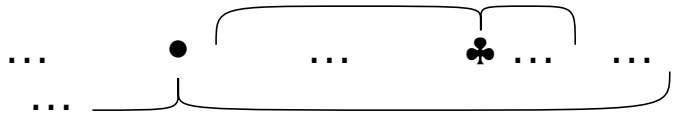
Sitovuuksien suuntien ja sitovuustasojen tarpeen analyysiä

- niitä tarvitaan osoittamaan operaattorin vaikutusalueen rajat, jos ne eivät muuten ole selvät
- sulut tekevät rajoista selvät
- prefix-operaattorin vaikutusalueen alku on selvä
- postfix-operaattorin vaikutusalueen loppu on selvä

⇒ tarve syntyy vain, jos saman osalausekkeen vasemmalla puolella on prefix- tai infix-operaattori, ja oikealla puolella postfix- tai infix-operaattori



- jos ko. tilanteessa operaattoreilla on eri sitovuustasot, tulkinta on selvä
  - korkeammalla tasolla oleva kaappaa sisällä olevan lausekkeen, matalammalla oleva laajemman alueen



- infix-operaattorin tapauksessa voi olla sama operaattori molemmin puolin
  ⇒ sitovuustaso ei riitä määräämään laskujärjestystä
  ⇒ on tiedettävä sitovuuden suunta

Käytännön neuvoja

- samalle sitovuustasolle ei kannata määritellä sekä oikealle että vasemmalle sitovia operaattoreita
  - harjoitustehtävä
- unaarioperaattorin sitovuuden suunnan ei tarvitse määräytyä siitä, onko se prefix vai postfix
  - harjoitustehtävä
- selkeintä on määritellä unaarioperaattorit eri sitovuustasolle kuin binäärioperaattorit

Vaihtoehto sitovuuden suunnille: vasen ja oikea sitovuustaso

- jokaiselle operaattorille määritellään sitovuustaso siihen suuntaan, mihin vaikutusalueen rajaa ei ole muuten osoitettu
  - prefix-operaattoreille oikea sitovuustaso
  - postfix-operaattoreille vasen sitovuustaso
  - infix-operaattoreille molemmat

- vasemmat ja oikeat sitovuustasot poimitaan erillisistä joukoista
  - esim. vasemmat sitovuustasot parillisia lukuja, oikeat parittomia

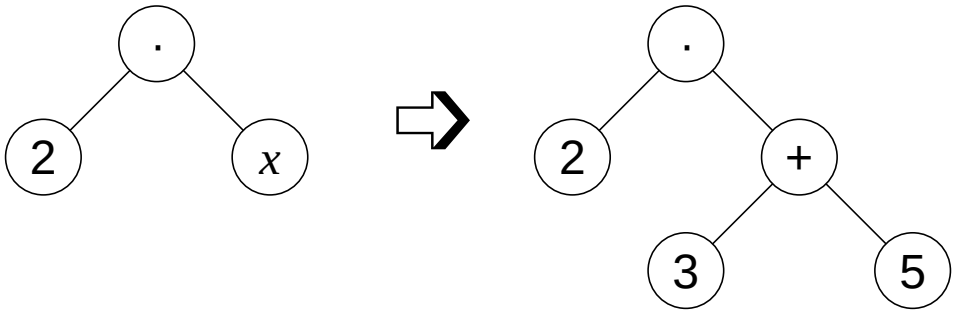
  ⇒ aina selvää, kummalla reunalla oleva operaattori kaappaa osalausekkeen

Parametrinvälitys, tekstikorvaus ja makrot

- jos lausekkeen osana on symboli, jonka paikalle voi panna alilausekkeen, korvauksen oletetaan yleensä tapahtuvan puurakenteen mukaan eikä suoraan tekstikorvauksena

  ⇒ loogisen merkityksen mukaan

  ⇒ saattaa olla tarpeen lisätä lausekkeeseen sulkuja
- esimerkiksi korvaus $x \leftarrow$ “3 + 5” lausekkeessa “$2 \cdot x$” tuottaa “$2 \cdot (3 + 5)$” eikä kuten tekstikorvaus “$2 \cdot 3 + 5$”



- aliohjelmien parametrit toimivat yleensä tähän tapaan
  - todellisuudessa ei välitetä alilauseketta, vaan sen tuottama arvo tai muistipaikan osoite
- makroissa kuitenkin käytetään tekstikorvausta
- esimerkki: C ja C++

```
#define tupla(x) 2*x
```

…

```
tupla(3+5)
```

tuottaa `2*3+5`

- tekstikorvauksella saa aikaan omituisempiakin:
  $(1 + x)\cdot(2 + 3)$
  – x:n tilalle teksti “9) + (1”
  ⇒ syntyy $(1 + 9) + (1)\cdot(2 + 3)$
  ⇒ kertolasku muuttui yhteenlaskuksi!
- toisaalta makroilla on mahdotonta toteuttaa luotettavasti niinkin yksinkertainen temppu kuin

```
void vaihda( int & x, int & y )
{ int tmp = x; x = y; y = tmp; }
```

⇒ makroja pitää välttää

Liitännäisyys ja vaihdannaisuus

- matemaatikkojen käsitteitä, joilla on seuraava merkitys

binäärioperaattori “$\star$” on	jos ja vain jos aina pätee
liitännäinen eli *assosiatiivinen*	$(x \star y) \star z = x \star (y \star z)$
vaihdannainen eli *kommutatiivinen*	$x \star y = y \star x$

- “liitännäisyys” on siis aivan eri asia kuin “liitännäisyyden suunta” eli sitovuuden suunta
- yhteys: jos operaattori on liitännäinen, sitovuuden suuntaaei tarvitse määritellä
- **varoitus!** operaattori, joka on matematiikassa vaihdannainen tai liitännäinen, ei välttämättä ole sitä ohjelmoinnissa
  – esim. liukulukulasku `0.1 + 1E40 + (-1E40)`
  – esim. `if( os && os-> ... )`
- abstraktilla alueella matemaatikoilla on tapana sallia kirjoittaa $x \star y \star z$ vain silloin kun “$\star$” on liitännäinen

Identiteettialkiot

- alkio i on binäärioperaattorin “$\star$”
  - *vasen identiteetti*, jos ja vain jos $i \star x$ tarkoittaa aina samaa kuin x
  - *oikea identiteetti*, jos ja vain jos $x \star i$ tarkoittaa aina samaa kuin x
  - *identiteetti*, jos ja vain jos se on “$\star$”:n vasen ja oikea identiteetti
- esimerkkejä
  - 0 on “+”:n identiteetti
  - 1 on “·”:n identiteetti
  - **T** on “$\wedge$”:n identiteetti
  - `0` on (tietyin varauksin) C:n “<<”-operaattorin oikea identiteetti ja “^”-operaattorin identiteetti
- kun $n = 0$, lausekkeella muotoa

  $$x_1 \star x_2 \star \ldots \star x_n$$

  tarkoitetaan yleensä “$\star$” :n (vasenta) identiteettiä
  - siksi esim. $\Sigma_{i=1}^{n} f(i) = 0$, kun $n = 0$
  - esim. $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ = **T**, kun $n = 0$
  - kätevää samantapaisista syistä kuin **for**-silmukan kierto nolla kertaa on kätevää ohjelmoinnissa

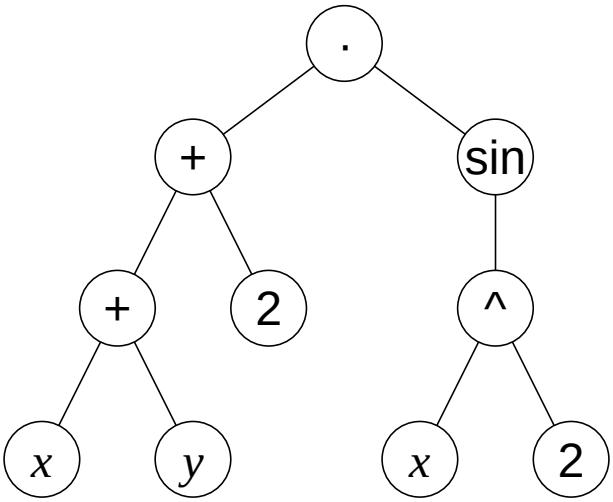
4.2 Lausekkeiden käsittelyesimerkki

Esittely

- aihe: reaalilukujen funktio tietueista koottuna puuna
- esimerkki kattaa funktioiden esittämisen, tulostamisen lausekkeena, yksinkertaisen sieventämisen ja derivoinnin
  - luvussa 6.3 lisätään lausekkeen lukeminen
- vaikka esimerkki liittyy matematiikan käsittelyyn tietokoneella, periaatteet ovat päteviä muuallakin
  - valitsemalla aihe matematiikasta saaiin varmasti kaikille tuttu esimerkki
- esimerkissä voisi hyödyntää olio-ohjelmoinnin keinoja
  - periminen, saantifunktiot, virtuaalifunktiot, …
- tietueet ovat varmemmin tuttuja kuin oliot

$\Rightarrow$ esimerkki esitetään tietueita käyttäen

- esimerkin tarkistamiseksi on tehty sitä vastaava C++-ohjelma
- esimerkin pitämiseksi yksinkertaisena muistin hallintaa ei esitetä
- esimerkkifunktio: $(x + y + 2) \cdot \sin x^2$



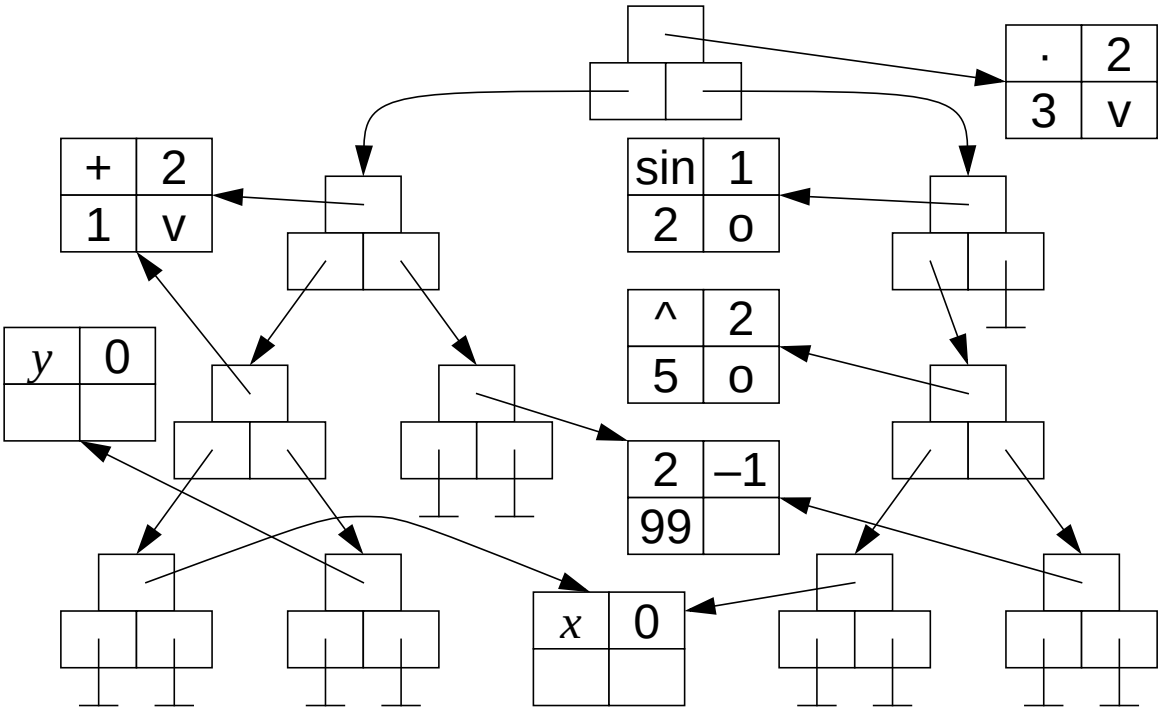
Toteutus tietueiden avulla

- solmutietueen (tai solmuolion) kentät
  - osoitin operaattorin lajiin *laji*
  - osoittimet enintään kahteen lapseen *vasen*, *oikea*
- operaattorin lajitietueen kentät
  - tulostusasu *nimi*, paitsi vakioilla *arvo*
  - parametrien määrä *par* $\in$ { –1, 0, 1, 2 }
    (–1 ja 0 erikoismerkitys, ks. jäljempää)
  - sitovuustaso *taso* $\in \boldsymbol{N}$
  - sitovuuden suunta *suunta* $\in$ { *vasen*, *oikea* }
- operaattorit
  - saman sitovuustason operaattorit sitovat samaan suuntaan
  - unaariop. pre- / postfix täsmää sitovuussuuntaan
  - esimerkkejä

nimi	^	+ ja –	· ja /	sin, cos	+ ja –
par	2	1	2	1	2
taso	5	4	3	2	1
suunta	o	o	v	o	v

- vakiot
  - *arvo* = vakion arvo lukuna
  - *par* = –1, jotta vakiot erottuisivat muuttujista
  - *taso*, *suunta*: ei väliä
- muuttujat
  - *nimi* = muuttujan nimi merkkijonona
  - *par* = 0
  - *taso*, *suunta*: ei väliä

- esimerkkilauseke tietueista koottuna puuna



Toteutusyksityiskohtia

- kutakin operaattoria, muuttujasymbolia ja (yleistä) vakion arvoa kohti on vain yksi lajitietue
  - säästää muistia
  - helpottaa jatkossa tapahtuvia vertailuja
- identtisistä osalausekkeista voi tehdä yhteisiä alipuita
  - monimutkaistaa muistin kierrätystä
- kattavasti tehtynä vaatii sopivan tietorakenteen, mutta helpottaa samuusvertailuja
  - esim. kun $f - f \Rightarrow 0$

Lausekkeen tulostus

- koodi

```
procedure tulosta( st : ^solmu, pv : N := 0, po : N := 0 )
    sulut : Boolean := false; pr : N
    if st^.laji.^par > 0 then
        pr := st^.laji.^taso
        sulut := st^.laji.^suunta = vasen ∧ pv ≥ pr
              ∨ st^.laji.^suunta = oikea ∧ po ≥ pr
              ∨ st^.laji.^par = 2 ∧ ( pv > pr ∨ po > pr )
    endif
    if sulut then print "("; pv := 0; po := 0 endif
    if st^.laji.^par = –1 then        ⇨ vakio
        print st^.laji.^arvo
    elsif st^.laji.^par = 0 then      ⇨ muuttuja
        print st^.laji.^nimi
    elsif st^.laji.^par = 1 then      ⇨ unaarioperaattori
        if st^.laji.^suunta = vasen then
            tulosta( st^.vasen, pv, pr ); print st^.laji.^nimi
        else
            print st^.laji.^nimi; tulosta( st^.vasen, pr, po )
        endif
    elsif st^.laji.^par = 2 then      ⇨ binäärioperaattori
        tulosta( st^.vasen, pv, pr )
        print st^.laji.^nimi
        tulosta( st^.oikea, pr, po )
    else error( "Laiton parametrien määrä" )
    endif
    if sulut then print ")" endif
endproc
```

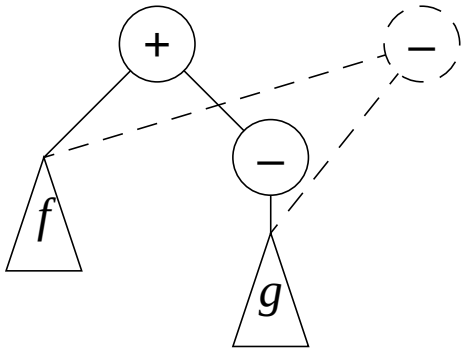
- kutsu

```
tulosta(st)
```

- tulostusesimerkkejä
  - $(x + y + 2)\cdot\sin x^2$ ⇨ `(x+y+2)*sinx^2`
  - $(x\cdot y\cdot 2) + \sin x^2$ ⇨ `x*y*2+sinx^2`
  - $(x + (2 + x))$ ⇨ `x+(2+x)`
  - $(\sin(((x^2)^{-3}\cdot y) + z)/2)$ ⇨
    `sin((x^2)^-3*y+z)/2`
  - $((\sin(x^2))^{(-3\cdot y)} + ((z)/2))$ ⇨
    `(sinx^2)^(-3*y)+z/2`

Lausekkeen sieventäminen

- tehdään vain ilmeisiä sievennyksiä kuten $f + 0$ ⇨ f
  - perusteellinen sieventäminen on todella kova työ!
  - ilmeisille sievennyksille tulee kohta tarvetta
- oletetaan, että *uplus_os* osoittaa siihen operaattorin lajitietueeseen, joka edustaa etumerkki-plussaa, jne.
  - riittää, että jokaista operaattoria kohti on yksi tietue
  - vakioita on yksi tietue jokaista vakion arvoa kohti
- oletetaan, että “uusi(*laji*, *vasen*, *oikea*)” varaa tai löytää solmutietueen ja palauttaa sen osoitteen
  - *oikea*n voi jättää unaarioperaattoreilla pois
  - muistinhallintaa ei esitetä
- sievennetty lauseke rakennetaan omaksi puukseen alkuperäistä lausekepuuta muuttamatta, mutta sen kanssa yhteisiä osia hyödyntäen
  - esim. $f + -g$ ⇨ $f - g$



- yleisrakenne

```
function siev( st : ^solmu, rek : Boolean := True )
: ^solmu
    if st^.laji^.par = 1 then
        sv := st^.vasen
        if rek then sv := siev(sv) endif
        if … then       ⇨ näille riveille unaarioper. …
            …           ⇨ … sievennyssääntöjä
        endif
        if rek then return uusi( st^.laji, sv ) endif
        return st
    elsif st^.laji^.par = 2 then
        sv := st^.vasen; so := st^.oikea
        if rek then sv := siev(sv); so := siev(so) endif
        if …            ⇨ tähän bin. op. siev.sääntöjä
        endif
        if rek then return uusi( st^.laji, sv, so ) endif
        return st
    else return st      ⇨ vakio ja muuttuja
    endif
endfunc
```

  - bitin *rek* avulla estetään alipuiden sieventäminen turhaan moneen kertaan
  - jollei mikään sievennyssääntö täsmää mutta alipuita on sievennetty, luodaan uusi solmu

- etumerkkiplussan säännöt ovat helpot: ne poistetaan

  if *st*^.*laji* = *uplus_os* **then return** *sv* **endif**

- etumerkki-miinuksen sääntöjä
  - $-0 \Rightarrow 0$
  - $--f \Rightarrow f$

```
if st^.laji = umiinus_os then
    if sv^.laji^.par = –1 and sv^.laji^.arvo = 0 then
        return sv
    elsif sv^.laji := umiinus_os then
        return sv^.vasen
    …
```

- esimerkki binääriplussan säännöistä:

  $f + (g \pm h) \Rightarrow (f + g) \pm h$

```
…
elsif so^.laji = bplus_os ∨ so^.laji = bmiinus_os then
    sv := siev( uusi( bplus_os, sv, so^.vasen ), false )
    return siev( uusi( so^.laji, sv, so^.oikea ), false )
elsif …
```

- pari esimerkkiä binäärimiinuksen säännöistä
  - $f - -g \Rightarrow f + g$
  - $f - (g - h) \Rightarrow (f - g) + h$

```
…
elsif so^.laji = umiinus_os then
    return siev(uusi(bplus_os, sv, so^.vasen), false)
elsif …
elsif so^.laji = bmiinus_os then
    sv := siev(uusi(bmiinus_os, sv, so^.vasen), false)
    return siev(uusi(bplus_os, sv, so^.oikea), false)
elsif …
```

- kaikkiaan yhteen-, vähennys-, kerto-, jako- ja potenssilaskun sääntöjä toteutettiin yli 30

Sievennystuloksia

- `(x+y+2)*sinx^2 ⇨ (2+x+y)*sinx^2`
- `x*y*2+sinx^2 ⇨ 2*x*y+sinx^2`
- `x+(2+x) ⇨ 2+x+x`
- `-3*(y*2)*(-z) ⇨ 6*y*z`

Lausekkeen derivointi

- derivoidaan parametrin *mt* suhteen
  - osoittaa muuttujatietuetta
  - oletus: kullakin muuttujalla on vain yksi tietue

  ⇒ "sama muuttuja" voidaan tarkastaa osoittimista
- pseudokoodi

```
function deriv( st : ^solmu, mt : ^op_laji ) : ^solmu
    if st^.laji^.par = –1 then        ⇨ vakion deriv. = 0
        return vakio_solmu( 0 )
    elsif st^.laji^.par = 0 then      ⇨ dx/dx = 1, dx/dy = 0
        if st^.laji = mt then return vakio_solmu( 1 )
        else return vakio_solmu( 0 )
        endif
    elsif st^.laji = umiinus_os then
        return uusi( umiinus_os, deriv( st^.vasen, mt ) )
    elsif …
    elsif st^.laji = kerto_os then    ⇨ d(f·g)/dx =
        return uusi( bplus_os,        ⇨ df/dx · g + f · dg/dx
                uusi(kerto_os, deriv(st^.vasen, mt), st^.oikea)
                uusi(kerto_os, st^.vasen, deriv(st^.oikea, mt))
            )
    elsif …
    else error( "Tätä en osaa derivoida" )
endfunc
```

- esimerkki: `x+(2+x)`
  - deriv. ennen siev.: `1+(0+1)`
  - deriv. siev. jälk.: `2`
- esimerkki: `-3*(y*2)*-z`
  - deriv. ennen siev.:
    `(-0*(y*2)+-3*(0*2+y*0))*-z+-3*(y*2)*-0`
  - deriv. siev. jälk.: `0`
- esimerkki: `x*y*2+sinx^2`
  - deriv. ennen siev.:
    `(1*y+x*0)*2+x*y*0+(cosx^2)*(2*x^1*1)`
  - deriv. siev. jälk.: `2*y+2*(cosx^2)*x`
- esimerkki: `(x+y+2)*sinx^2`
  - deriv. ennen siev.: `(1+0+0)*sinx^2+`
    `(x+y+2)*((cosx^2)*(2*x^1*1))`
  - deriv. siev. jälk.:
    `sinx^2+2*(2+x+y)*(cosx^2)*x`

$\Rightarrow$ sievennösten tarve ja hyöty ovat ilmeisiä!

Johtopäätöksiä

- puina (tai yhteisiä osia hyödyntävänä puiden kokoelmana) esitetyt lausekkeet ovat tehokas tietorakenne
  - helppoja käsitellä tietokoneohjelmilla (tosin muistin hallinta ohitettiin)
  - saadaan aikaan “älykkäitä” toimintoja
- derivointi on hyvin helppoa
- sieventäminen on yllättävän vaikeaa!

Lausekkeiden koneelliseen käsittelyyn ja sen taustalla olevaan teoriaan palataan luvussa 5

5 ÄÄRELLISET AUTOMAATIT

Automaattien ja formaalien kielten teoriassa *kieli* on joukko merkkijonoja (eikä mitään muuta)

- jokaista kieltä vastaa *päätöstehtävä* eli “kyllä” / “ei” -kysymys: kuuluuko syötemerkkijono ko. kieleen
- jokaista päätöstehtävää vastaa kieli: ne merkkijonot, joille vastaus on “kyllä”
  - esim. ne numerojonot, joita vastaava luku on alkuluku

$\Rightarrow$ päätöstehtävien teoria voidaan tulkita kielten teoriaksi

Automaattien ja formaalien kielten teoria tutkii muun muassa

- kielten ryhmittelyä kieliopillisin keinoin
  - kielioppi on keino määritellä kieli
- kielten ryhmittelyä sen mukaan, millaisella koneella merkkijonon kuulumisen kieleen voi tarkastaa
- tiettyyn kieleen kuuluvien merkkijonojen automatisoitua käsittelyä

Noam Chomskyn neliportainen kielioppien hierarkia (1956, 1959) tarjoaa erään yleiskuvan kielten teoriasta

- jos tyhjä merkkijono ε unohdetaan, niin

  säännölliset kielet (*regular sets*)
  $\subset$ *yhteysriippumattomat kielet* (*context-free languages*)
  $\subset$ *yhteysherkät kielet* (*context-sensitive languages*)
  $\subset$ *yleiset kielet*

  (varoitus! suomennokset vakiintumattomia)

- yleiset kielet = kaikki tietokoneella “hyväksyttävissä” olevat merkkijonojen joukot
  ⇒ tutkimme niiden teoriaa perin pohjin kurssilla OHJ-2300 Johdatus tietojenkäsittelyteoriaan
- säännölliset kielet vastaavat äärellisiä automaatteja
  – vrt. tilakone
  ⇒ teoria tärkeä monella ohjelmoinnin alueella
  – paljastavat (kiinteän kokoisen) äärellisen tietokoneen kykyjen rajat
- yhteysriippumattomat kielet tärkeitä ohjelmointi- yms. koneelle tarkoitettujen kielten määrittelyssä ja koneellisessa käsittelyssä

⇒ käsittelemme
  – tässä luvussa säännöllisiä kieliä ja äärellisiä automaatteja
  – luvussa 6 yhteysriippumattomia kieliä, BNF:ää ja jäsentämistä

Äärellinen automaatti on luonteva matemaattinen malli äärellistilaiselle tietokoneelle, joka saa syötteen merkki kerrallaan

- ts. ei voi selata syötettä edestakaisin

Äärellinen automaatti on monissa suhteissa hyödyllinen ja laajalti käytetty

- tekstialkioiden tunnistus kääntäjän etupäässä perustuu tavallisesti äärellisiin automaatteihin
- monet merkkijonojen käsittelyalgoritmit perustuvat äärellisiin automaatteihin
  – esim. Unixin `grep`

- ohjelmistojen suunnittelumenetelmistä ja tietoliikenteestä tuttu “tilakone” on samankaltainen kuin äärellinen automaatti
- äärellisen automaatin matematiikka on helppoa
  ⇒ tarjoaa hyvän ohjelmoinnin matemaattisten menetelmien harjoittelukohteen
  – hyödyllistä esitietoa jatkoa varten

5.1 Säännölliset lausekkeet

Merkkijonot

- olkoon Σ jokin äärellinen joukko merkkejä
  - *aakkosto* (*alphabet*)
  - esim. ASCII-merkit, ISO-Latin-1-merkit, 8-bittiset tavut, numeromerkit, { ‘0’, ‘1’, …, ‘9’ }, …
- tyhjää merkkijonoa merkitään symbolilla “ε” (epsilon)
- kaikkien Σ:sta muodostettavien äärellisten merkkijonojen joukkoa merkitään Σ^*
- esimerkkejä
  - $\{a\}^* = \{ \varepsilon, a, aa, aaa, \ldots \}$
  - $\{0, 1\}^* = \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots \}$
- merkkijonon α *pituus* $|\alpha|$ on α:n merkkien määrä
  - |kukkuu| = 6
  - $|\varepsilon| = 0$
- jokainen merkki on samalla merkkijono, jonka pituus on yksi
- selvyyden vuoksi sovimme
  - kreikkalaiset kirjaimet α, β, … edustavat merkkijonoja
  - kursivoidut kirjaimet a, b, … edustavat yksittäisiä merkkejä
  - jos on tarpeen korostaa, että kyse on konkreettisesta merkistä tai merkkijonosta (eikä sitä edustavasta symbolista), se voidaan panna lainausmerkkeihin kuten “abc”
- merkkijonon β liittäminen merkkijonon α perään esitetään panemalla ne peräkkäin
  - esim. jos α = “tietojen” ja β = “käsittely”, niin $\alpha\beta$ = “tietojenkäsittely”

- $\alpha\varepsilon = \varepsilon\alpha = \alpha$, olipa α mikä tahansa merkkijono
- jos $\alpha \in \Sigma^*$ ja $n \in \boldsymbol{N}$, niin
  - $\alpha^0 = \varepsilon$
  - $\alpha^{n+1} = \alpha^n\alpha$
- ts. α^n on α toistettuna n kertaa
  - esim. “10”^{3} = “101010”

Kieli (*language*) on mikä tahansa Σ^*:n osajoukko

- esimerkiksi jos $\Sigma = \{0, 1\}$, niin seuraavat ovat kieliä:
  - $\emptyset$
  - { 10011 }
  - { 10011, 0000000 }
  - { ε, 0, 00, 000 }
  - { ε, 0, 00, 000, … }
  - { 0, 1, 00, 11, 000, 111, … }
  - { ε, 0, 1, 00, 01, 10, 11, 000, 001, … }
- esimerkiksi “luonnolliset luvut 10-järjestelmässä” voidaan määritellä ISO-Latin-1*:n osajoukoksi, jossa on kaikki pelkästään numeromerkeistä koostuvat epätyhjät merkkijonot
- kieli on *äärellinen*, jos ja vain jos siihen kuuluu vain äärellinen määrä merkkijonoja

Kielten määrittelemisestä

- äärellisen kielen voi määritellä luettelemalla kaikki siihen kuuluvat merkkijonot
- äärettömälle kielelle tämä keino ei toimi
  $\Rightarrow$ tarvitaan muita keinoja
- kielen (äärellisen tai äärettömän) voi määritellä matemaattisella kaavalla
  - esim.
    {“0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}* – {ε}

- ihmisille tarkoitetuissa äärettömien kielten määritelmissä käytetään usein merkkiä "…"
  - vrt. "ja niin edelleen"
  - tulkinta ei aina selvä, esim. {0, 1, 00, 01, …}
- tässä luvussa pohditaan tietokoneelle sopivia keinoja määritellä (mahdollisesti äärettömiä) kieliä

Säännölliset lausekkeet

- *säännölliset lausekkeet* (eli *säännölliset ilmaukset*) ovat eräs yksinkertaisimmista keinoista määritellä (mahdollisesti ääretön) kieli
- määritelmä

  Olkoon Σ *aakkosto, joka ei sisällä merkkejä* "Ø", "ε", "·", "|", "*", "(" *ja* ")". ***Säännöllinen lauseke (regular expression)*** *on mikä tahansa lauseke, joka on koottu* Σ*:n alkioista, symboleista* "Ø" *ja* "ε", *ja / tai operaattoreista* "·", "|" *ja* "*" *seuraavien sääntöjen mukaisesti:*
  - *Jos* $a \in \Sigma$*, niin* Ø, ε *ja* a *ovat säännöllisiä lausekkeita.*
  - *Jos* r *ja* s *ovat säännöllisiä lausekkeita, niin* $(r \mid s)$, $(r \cdot s)$ *ja* (r^*) *ovat säännöllisiä lausekkeita.*
- tässä "lauseke" on ymmärrettävä sanan tavallisessa matemaattisessa mielessä
  - vrt. $x^2 + 2x + 1$
  - kuten luvussa 4.1
- sulkujen määrän vähentämiseksi sovimme, että "*" sitoo voimakkaimmin, sitten "·" ja lopuksi "|"
  - esim. $a \cdot b \mid c \cdot d^*$ tarkoittaa $((a \cdot b) \mid (c \cdot (d^*)))$
- tapana on jättää "kertolasku"operaattori "·" kirjoittamatta
  - esim. $a \cdot b \mid c \cdot d^*$ kirjoitetaan $ab \mid cd^*$

- säännöllisen lausekkeen r *pituus* $|r|$ on sen merkkien määrä niillä suluilla ja "$\cdot$"-merkeillä, jotka siihen on kirjoitettu
  - esim. $|a \cdot b \mid c \cdot d^*| = 8$ ja $|ab \mid cd^*| = 6$

Säännöllisten lausekkeiden merkitys

- määritelmä

  Jokainen säännöllinen lauseke r määrittelee kielen $\mathcal{L}(r) \subseteq \Sigma^*$ seuraavasti (taulukossa $a \in \Sigma$ ja s ja t ovat säännöllisiä lausekkeita):

r	$\mathcal{L}(r)$
$\emptyset$	$\emptyset$
ε	$\{\varepsilon\}$
a	$\{a\}$
$s \mid t$	$\mathcal{L}(s) \cup \mathcal{L}(t)$
st	$\{\alpha\beta \mid \alpha \in \mathcal{L}(s) \wedge \beta \in \mathcal{L}(t)\}$
s^*	$\{\varepsilon\} \cup \mathcal{L}(s) \cup \mathcal{L}(ss) \cup \mathcal{L}(sss) \cup \ldots$

  ***Säännölliset kielet** (**regular languages**) ovat ne kielet, jotka voi määritellä säännöllisillä lausekkeilla.*

- siis
  - $\mathcal{L}(\emptyset)$ ei sisällä yhtään merkkijonoa
  - $\mathcal{L}(\varepsilon)$ sisältää yhden jonon, nimittäin tyhjän jonon
  - $\mathcal{L}(a)$ sisältää yhden jonon, nimittäin a:n
  - $\mathcal{L}(s \mid t)$ sisältää kaikki s:n ja kaikki t:n jonot
  - $\mathcal{L}(st)$ sisältää kaikki ne jonot, jotka saadaan pistämällä jonkin s:n jonon perään jokin t:n jono
  - $\mathcal{L}(s^*)$ sisältää ne jonot, jotka saadaan pistämällä äärellinen määrä s:n jonoja peräkkäin

- esimerkkejä
  - $\mathcal{L}(\ ab \mid cd^*\ ) = \{ab, c, cd, cdd, cddd, cdddd, \ldots\}$
  - $\mathcal{L}(\ 1(0 \mid 1)^*\ ) = \{1, 10, 11, 100, 101, 110, 111, \ldots\}$
  - $\mathcal{L}(\ 0 \mid 1(0 \mid 1)^*\ )$ = binääriluvut ilman turhia etunollia
  - $\mathcal{L}(\ (a_1 \mid a_2 \mid \ldots \mid a_n)^*\ ) = \{a_1, a_2, \ldots, a_n\}^*$

Helpottaaksemme säännöllisten lausekkeiden kirjoittamista otamme käyttöön joukon lyhenteitä

- $r^+ := rr^*$
- r^n on r toistettuna n kertaa, ts.
  - $r^0 := \varepsilon$
  - $r^{n+1} := r^n r$
- sovimme, että $^+$ ja ^{n} sitovat yhtä voimakkaasti kuin *
  - esim. $ab^n = a(b^n)$
- muista sivulta 134, että kun r:n paikalle pannaan jotain, tarvittaessa lisätään sulkuja
  - esim. jos $r = ab$, niin $r^+ = (ab)^+ = ab(ab)^*$ eikä $abab^*$
- 0 | … | 9 = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9, ja vastaavasti muille yleisesti tunnetuille kolmen pisteen lyhenteille
- **huom!** nämä merkinnät eivät itse ole säännöllisiä lausekkeita, vaan lyhenteitä, joiden tarkoitus on helpottaa pitkien säännöllisten lausekkkeiden kirjoittamista
- erolla on merkitystä, kun tutkitaan säännöllisten lausekkeiden ilmaisuvoimaa ja pituutta
- esimerkiksi r^4 ei itse ole säännöllinen lauseke, vaan vain lyhenne lausekkeelle $rrrr$
  - esim. $|\ (abc)^4\ | = |\ abcabcabcabc\ | = 12$

- esimerkiksi lyhenteessä $a^n b^n$ n ei ole parametri, vaan sillä on oltava kiinteä arvo
  $\Rightarrow$ kieltä $\{ a^n b^n \mid n \in \boldsymbol{N} \}$ ei voi millään ilmeisellä tavalla esittää säännöllisenä lausekkeena
- meillä on nyt neljä tasoa käsitteitä:
  - yksittäiset merkit $a \in \Sigma$
  - merkkijonot $\alpha \in \Sigma^*$
  - säännölliset lausekkeet
    $r \in (\Sigma \cup \{\text{“Ø”}, \text{“ε”}, \text{“·”}, \text{“|”}, \text{“*”}, \text{“(”}, \text{“)”}\})^+$, $\mathcal{L}(r) \subseteq \Sigma^*$
  - lyhennemerkityt säännölliset lausekkeet
    $lyh \in$ matemaattiset merkinnät, lyh esittää jotain säännöllistä lauseketta r

5.2 Deterministiset äärelliset automaatit

Määritelmä

***Deterministinen äärellinen automaatti** (**deterministic finite automaton**) (**DFA**) on mikä tahansa rakenne* $(Q, \Sigma, \delta, \hat{q}, F)$*, jolle pätee:*

- Q *on äärellinen joukko,*
- Σ *on äärellinen joukko siten, että* $\varepsilon \notin \Sigma$
- δ *on osittainen funktio* $Q \times \Sigma \rightarrow Q$*,*
- $\hat{q} \in Q$ *ja*
- $F \subseteq Q$*.*

Osilla on seuraavat nimet:

- Q *on **tilojen** (**state**) joukko,*
- Σ ***syöteaakkosto** (**input alphabet**),*
- δ *on **tilasiirtymäfunktio** (**transition function**),*
- $\hat{q}$ *on **alkutila** (**initial state**) ja*
- F *on **lopputilojen** (**final states**) joukko.*

- yllä oleva määritelmä on tarkoituksellisesti muotoiltu siten, että DFA:lle asetettavat muodolliset vaatimukset (yläosa) ja sen osille annetut nimet (alaosa) erottuvat selvästi toisistaan
- tapana on kuitenkin esittää määritelmät hieman tiiviimmin:

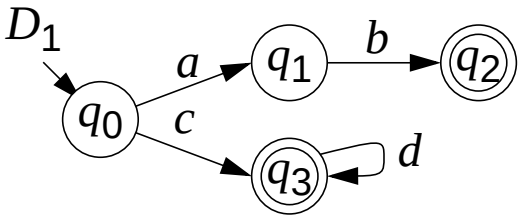
***Deterministinen äärellinen automaatti** (**deterministic finite automaton**) (**DFA**) on viisikko* $(Q, \Sigma, \delta, \hat{q}, F)$*, missä:*

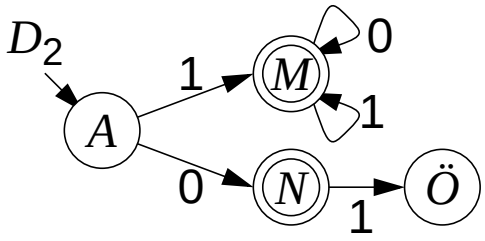
- Q: *äärellinen joukko **tiloja** (**state**),*
- Σ: *äärellinen **syöteaakkosto** (**input alphabet**),*
- δ: $Q \times \Sigma \rightarrow Q$: *osittainen **tilasiirtymäfunktio** (**transition function**),*
- $\hat{q} \in Q$: ***alkutila** (**initial state**), ja*
- $F \subseteq Q$: ***lopputilat** (**final states**).*

- että δ on “osittainen” tarkoittaa, että $\delta(q, a)$ ei välttämättä ole määritelty kaikille $q \in Q$ ja $a \in \Sigma$
  - vrt. $^1/_x$ ei ole määritelty kun $x = 0$

DFA esitetään usein kuvana

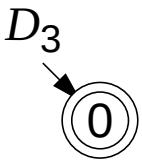
- aakkosto oletetaan tunnetuksi tai ilmoitetaan erikseen
  - jollei mitään ole sanottu, niin aakkosto = kuvassa esiintyvien kaarten nimien joukko
- tilat ja tilasiirtymät esitetään graafina, jonka solmut esittävät tiloja ja kaaret tilasiirtymiä
- tilasta q on a-niminen kaari tilaan q' jos ja vain jos $\delta(q, a) = q'$
  - samasta tilasta voi kullakin nimellä olla korkeintaan yksi lähtökaari
- alkutila osoitetaan tyhjästä alkavalla nuolella
- lopputilat esitetään kaksinkertaisella ympyrällä
- esimerkkejä





- automaatissa D_1
  - $Q_1 = \{q_0, q_1, q_2, q_3\}$
  - $\Sigma_1 = \{a, b, c, d\}$
  - $\delta_1 = \{ \ ((q_0, a), q_1), ((q_0, c), q_3), ((q_1, b), q_2), ((q_3, d), q_3) \ \}$
  - $\hat{q}_1 = q_0$
  - $F_1 = \{q_2, q_3\}$

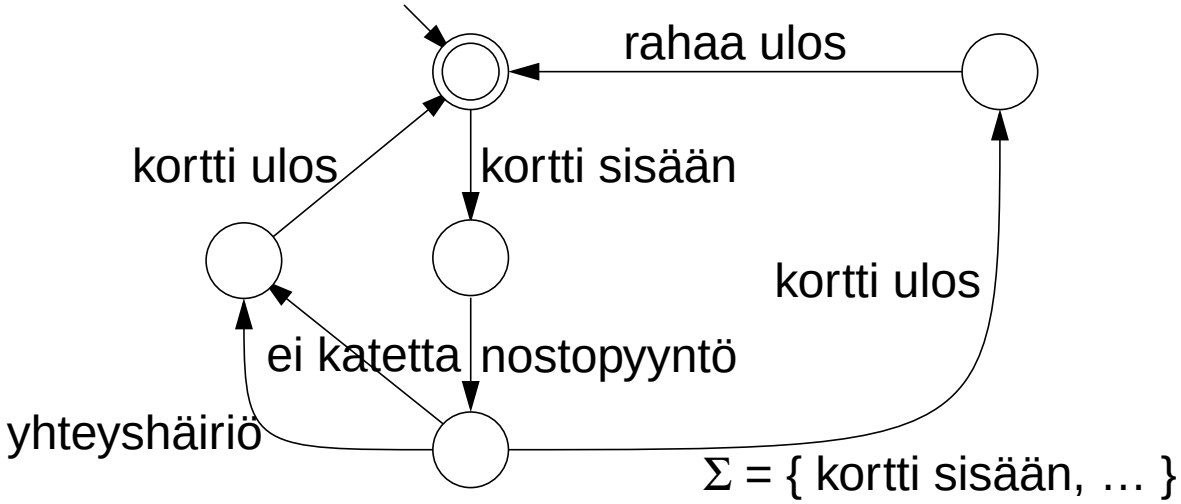
- automaatissa D_2
  - $Q_2 = \{A, M, N, Ö\}$
  - $\Sigma_2 = \{0, 1\}$
  - $\delta_2 = \{ (M,0,M), (A,1,M), (M,1,M), (N,1,Ö), (A,0,N) \}$
  - $\hat{q}_2 = A$ ja $F_2 = \{N, M\}$
- tilojen nimet eivät tärkeitä
  $\Rightarrow$ usein jätetään merkitsemättä kuvissa
- kaikkien aakkoston symbolien ei tarvitse esiintyä tilasiirtymissä!
- esimerkki



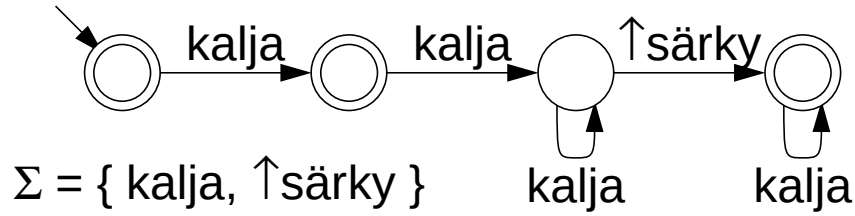
  - $Q_3 = \{0\}$
  - $\Sigma_3 = \{a, \ldots, z, 0, \ldots, 9, !, ?, @, \#, \$\}$
  - $\delta_3 = \emptyset$
  - $\hat{q}_3 = 0$
  - $F_3 = \{0\}$
-

Sovellusesimerkkejä

- pankkiautomaatti käyttäjän näkökulmasta



- opiskelijaelämää: jos (ja vain jos) juo ≥ 2 kaljaa, niin alkaa särkeä päätä



Deterministisen äärellisen automaatin toiminta

- olkoon $D = (Q, \Sigma, \delta, \hat{q}, F)$ deterministinen äärellinen automaatti
- D aloittaa toimintansa alkutilasta $\hat{q}$
- D lukee syöteaakkostoon Σ kuuluvia merkkejä
- kun D on tilassa q, merkin $a \in \Sigma$ lukemisen seurauksena voi tapahtua kaksi asiaa:
  - jos $\delta(q, a)$ on määritelty, niin D siirtyy tilaan $\delta(q, a)$
  - muutoin D lakkaa olemasta missään tilassa

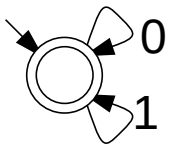
- δ voidaan laajentaa osittaiseksi funktioksi
  $\delta^*: Q \times \Sigma^* \to Q$
  seuraavasti:
  - $\delta^*(q, \varepsilon) := q$
  - $\delta^*(q, a_1a_2\ldots a_na_{n+1})$ on määritelty, jos ja vain jos $\delta^*(q, a_1a_2\ldots a_n)$ ja $\delta(\delta^*(q, a_1a_2\ldots a_n), a_{n+1})$ ovat määritellyt, ja silloin
    $\delta^*(q, a_1a_2\ldots a_na_{n+1}) := \delta(\delta^*(q, a_1a_2\ldots a_n), a_{n+1})$
- $\delta^*(q, a_1a_2\ldots a_n)$ palauttaa sen tilan, johon D siirtyy, kun se lukee merkit $a_1, a_2, \ldots, a_n$ alkaen tilasta q
- määritelmästä seuraa
  $\delta^*(q, a) = \delta^*(q, \varepsilon a) = \delta(\delta^*(q, \varepsilon), a) = \delta(q, a)$
  $\Rightarrow$ $\delta^*(q, a) = \delta(q, a)$
  $\Rightarrow$ funktion δ^* merkitseminen symbolilla “δ” ei aiheuta väärinkäsityksen vaaraa
  $\Rightarrow$ teemme jatkossa niin
- myös pätee $\delta(\delta(q, \alpha), \beta) = \delta(q, \alpha\beta)$ (harjoitustehtävä)
- esimerkiksi D_1 vaihtaa tiloja seuraavasti:
  - $\delta_1(q_0, a) = q_1$
  - $\delta_1(q_1, b) = q_2$
  - $\delta_1(q_0, ab) = q_2$
  - $\delta_1(q_0, c) = \delta_1(q_0, cddddd) = q_3$
  - $\delta_1(q_0, abc)$ ei ole määritelty

Deterministisen äärellisen automaatin hyväksymä kieli

- määritelmä

  Deterministinen äärellinen automaatti $D = (Q, \Sigma, \delta, \hat{q}, F)$ ***hyväksyy*** *(****accepts****) merkkijonon* $\alpha \in \Sigma^*$, *jos ja vain jos* $\delta(\hat{q}, \alpha) \in F$. *D:n* ***hyväksymä kieli*** *on* $\mathcal{L}(D) := \{ \alpha \in \Sigma^* \mid \delta(\hat{q}, \alpha) \in F \}$.
- määritelmässä “$\delta(\hat{q}, \alpha) \in F$” sisältää myös vaatimuksen, että $\delta(\hat{q}, \alpha)$ on määritelty

- toisin sanoen,
  - D hyväksyy α:n jos ja vain jos α:n lukeminen vie D:n alkutilasta johonkin lopputilaan
  - D:n hyväksymä kieli on sen hyväksymien merkkijonojen joukko
- esimerkkejä
  - D_1 hyväksyy *ab*, *c* ja *cdd*
  - D_1 hylkää *a*, *abc* ja *cddc*
  - kaljasärky hyväks. “ε”,“kalja” ja “kalja kalja ⇑särky”
  - kaljasärky hylkää “kalja kalja”
- esimerkkejä (pitkät aakkoset lyhentäen)
  - $\mathcal{L}(D_1) = \mathcal{L}(ab \mid cd^*)$
  - $\mathcal{L}(D_2) = \mathcal{L}(0 \mid 1(0 \mid 1)^*)$
  - $\mathcal{L}(D_3) = \mathcal{L}(\varepsilon) = \{\varepsilon\}$
  - $\mathcal{L}$(pankkiautomaatti) = (ks np ((ek | yh) ku | ku ru))*
  - $\mathcal{L}$(kaljasärky) = ε | k | k k k* s k*
- näissä esimerkeissä automaatin hyväksymä kieli on säännöllinen
- luvussa 5.4 tulemme huomaamaan, että tämä ei ole sattumaa
- huom! tämän DFA: hyväksymä kieli ei ole 0 | 1(0 | 1)*, vaikka se hyväksyy kaikki sen merkkijonot!



  - kieli on (0 | 1)*

5.3 DFA:n operaatioita

Tilasiirtymäfunktion täydentäminen täydeksi

- edellä todettiin, että δ voi olla osittainen funktio
  - ts. $\delta(q, a)$:n ei tarvitse olla määritelty kaikille $q \in Q$ ja $a \in \Sigma$
- seuraava lause osoittaa, että tämä on vain automaattien piirtämistä ja lukemista helpottava käytäntö, eikä periaatteellisesti tärkeä:

  Olkoon D deterministinen äärellinen automaatti. On olemassa deterministinen äärellinen automaatti D' siten, että D':n tilasiirtymäfunktio on täysi, D':lla on korkeintaan yksi tila enemmän kuin D:llä, ja $\mathcal{L}(D') = \mathcal{L}(D)$.
- lauseen todistamiseksi määrittelemme operaation $full(D)$ siten, että $D' = full(D)$ täyttää vaatimukset
- jos D:n tilasiirtymäfunktio on täysi, valitsemme $full(D) = D$
- jos DFA:n $D = (Q, \Sigma, \delta, \hat{q}, F)$ tilasiirtymäfunktio ei ole täysi, $full(D)$ voidaan muodostaa
  - lisäämällä yksi uusi tila q_R, $q_R \notin Q$ ("R" = "reject")
  - vetämällä kaari (q_R, a, q_R) jokaiselle $a \in \Sigma$
  - vetämällä kaari (q, a, q_R) jokaiselle $q \in Q$ ja $a \in \Sigma$, joille $\delta(q, a)$ ei ole määritelty
- siis $full(D) := (Q', \Sigma, \delta', \hat{q}, F)$, missä
  - $Q' = Q \cup \{q_R\}$, $q_R \notin Q$
  - $\delta'(q, a) = \delta(q, a)$, jos $\delta(q, a)$ on määritelty
  - $\delta'(q, a) = q_R$, jos $\delta(q, a)$ ei ole määritelty, vaikka $q \in Q$ ja $a \in \Sigma$
  - $\delta'(q_R, a) = q_R$ kaikille $a \in \Sigma$

Täydennöksen todistaminen oikeaksi

- D' = $full(D)$ on DFA, koska
  - Q' ja Σ ovat äärellisiä (miksi?)
  - $\delta'(q, a) \in Q \cup \{q_R\} = Q'$
  - $\hat{q} \in Q \subseteq Q'$
  - $F \subseteq Q \subseteq Q'$
- D' matkii tarkasti kaikki ne siirtymät, jotka D voi tehdä
- D:n lopputilat ovat myös D':n lopputiloja

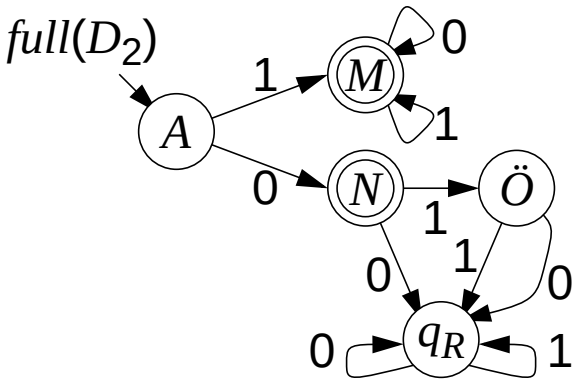
$\Rightarrow$ jos $\alpha \in \mathcal{L}(D)$, niin $\alpha \in \mathcal{L}(D')$

- D voi matkia kaikki muut D':n siirtymät paitsi q_R:ään vievät
- D' ei pääse q_R:stä mihinkään lopputilaan
- D':n lopputilat ovat myös D:n lopputiloja

$\Rightarrow$ jos $\alpha \in \mathcal{L}(D')$, niin $\alpha \in \mathcal{L}(D)$

- siis $\mathcal{L}(D') = \mathcal{L}(D)$
- väitteen muut osat on helppo tarkastaa ·/·

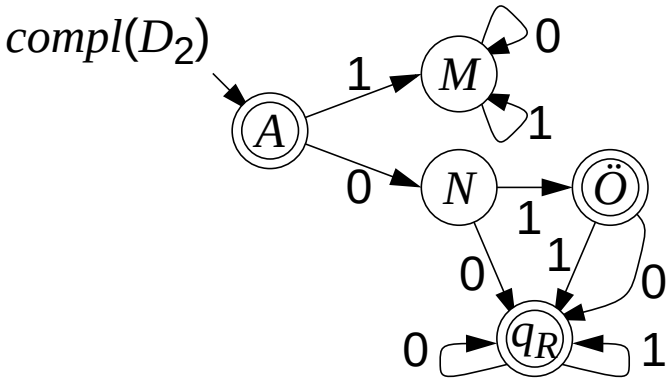
Esimerkki: D_2:n täydentäminen



DFA:n komplementointi

- aakkostosta Σ otetun kielen L komplementti $\Sigma^* - L$ on ne merkkijonot, jotka eivät kuulu L:ään
  - ts. $\Sigma^* - L = \{ \alpha \in \Sigma^* \mid \alpha \notin L \}$

- jos DFA:n $D = (Q, \Sigma, \delta, \hat{q}, F)$ tilasiirtymäfunktio on täysi, määrittelemme $compl(D) := (Q, \Sigma, \delta, \hat{q}, Q - F)$
  - ts. D siten muutettuna, että lopputilat on vaihdettu ei-lopputiloiksi ja päinvastoin
- jos D:n tilasiirtymäfunktio ei ole täysi, määrittelemme
  $$compl(D) := compl(full(D))$$
  - miksi tämä ei ole kehämääritelmä?
  - miten voi nähdä, että jokaiselle DFA:lle D pätee $compl(D) = compl(full(D))$?
- esimerkki



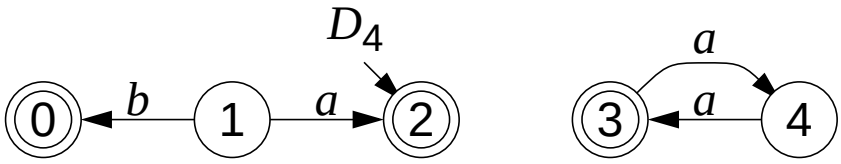
- on helppo nähdä, että $compl(D)$ on DFA, ja
  $$\mathcal{L}(compl(D)) = \Sigma^* - \mathcal{L}(D)$$
  - olkoon $D' = full(D)$

  $\Rightarrow$ $compl(D) = compl(D')$ ja $\mathcal{L}(D') = \mathcal{L}(D)$
  - koska D':n δ on täysi, on $\delta(\hat{q}, \alpha)$ aina määritelty
  - $\delta(\hat{q}, \alpha)$ on jokaiselle $\alpha \in \Sigma^*$ sama D':ssä ja $compl(D')$:ssä
  - $\delta(\hat{q}, \alpha)$ on lopputila $compl(D')$:ssä jos ja vain jos se ei ole lopputila D':ssä

  $\Rightarrow$ $compl(D) = compl(D')$ hyväksyy α:n jos ja vain jos D' ei hyväksy α:aa ·/·

Saavuttamattomien tilojen poisto

- joskus DFA:ssa on tiloja, joihin ei voi päästä alkutilasta



- sellaisten tilojen ja niihin liittyvien kaarten poisto ei muuta DFA:n hyväksymää kieltä
  - kaikki alkutilasta johonkin lopputilaan vievät polut jäävät jäljelle
  - poisto ei synnytä uusia alkutilasta lopputilaan vieviä polkuja

$\Rightarrow$ määrittelemme operaattorin “puhdista” (clean), joka poistaa ne

Jos $D = (Q, \Sigma, \delta, \hat{q}, F)$ *on DFA, niin*
$cln(D) := (Q', \Sigma, \delta', \hat{q}, F')$, *missä*
- $Q' = \{ q \in Q \mid \exists \alpha \in \Sigma^* : q = \delta(\hat{q}, \alpha) \}$.
- *Jos* $q \in Q'$ ja $a \in \Sigma$, *niin* $\delta'(q, a) = \delta(q, a)$.
- $F' = F \cap Q'$.

- nytkin on tarpeen tarkastaa, että lopputulos on DFA
  - Q' ja Σ ovat äärellisiä
  - $\hat{q} \in Q'$, koska $\hat{q} \in Q$ ja $\hat{q} = \delta(\hat{q}, \varepsilon)$
  - jos $q \in Q'$ niin $\delta'(q, a) = \delta(q, a) \in Q'$, koska silloin on α siten, että $q = \delta(\hat{q}, \alpha)$ ja $\delta(q, a) = \delta(\hat{q}, \alpha a)$
  - $F' = F \cap Q' \subseteq Q'$ ·/·
- kielen säilyminen tuli jo todettua

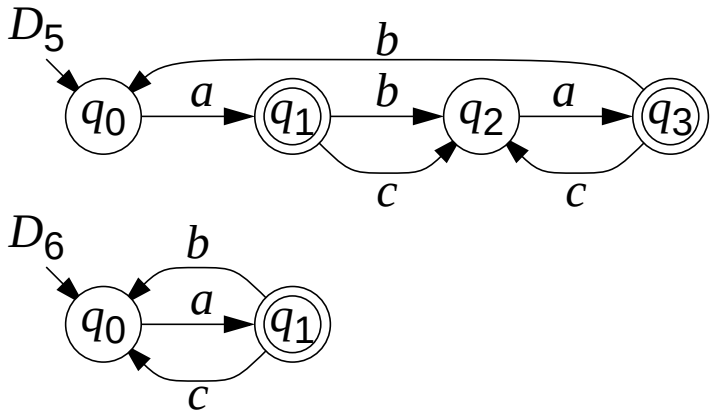
  Jos D on DFA, niin $\mathcal{L}(cln(D)) = \mathcal{L}(D)$.
- pätee (miksi?):

  Jos D:n tilasiirtymäfunktio on täysi, niin cln(D):n tilasiirtymäfunktio on täysi.

- samaan tapaan voi poistaa myös tilat, joista ei pääse mihinkään lopputilaan, kunhan varoo poistamasta alkutilaa
  - tällöin täysi tilasiirtymäfunktio voi muuttua aidosti osittaiseksi

Deterministisen äärellisen automaatin minimointi: johdanto

- saman kielen hyväksyviä deterministisiä äärellisiä automaatteja on äärettömän monta
  - voidaan lisätä loputtomasti eristettyjä tiloja
- kiintoisampi esimerkki



- joskus on tarpeen löytää *minimoitu* eli mahdollisimman pieni deterministinen äärellinen automaatti, joka hyväksyy annetun kielen
- saavuttamattomien tilojen poisto tekee osan minimoinnista, mutta ei kaikkea

$\Rightarrow$ tutkimme, miten minimoinnin voi saattaa loppuun

Lohkot

- olkoon $D = (Q, \Sigma, \delta, \hat{q}, F)$, missä δ on täysi
- jokaiselle $q \in Q$ voidaan määritellä q:n hyväksymä kieli seuraavasti:

  $$\mathcal{L}(q) := \{ \alpha \in \Sigma^* \mid \delta(q, \alpha) \in F \}$$

  – ts. kieli, jonka D hyväksyisi, jos alkutilaksi otettaisiin q
- tilan q määräämä *lohko* on niiden D:n tilojen joukko, jotka hyväksyvät saman kielen kuin q

  $$[[q]] := \{ q' \in Q \mid \mathcal{L}(q') = \mathcal{L}(q) \}$$
- jokainen $q \in Q$ kuuluu tasan yhteen lohkoon, ja

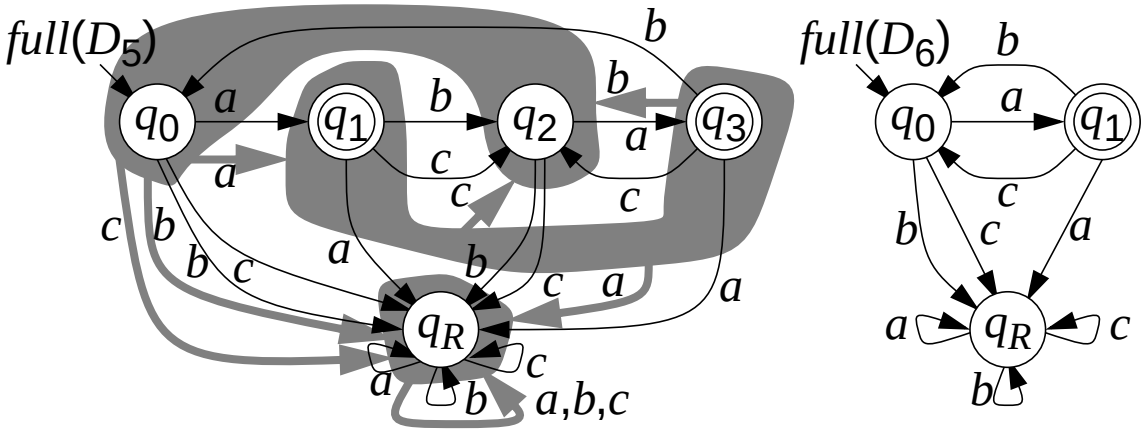
  $$q' \in [[q]] \Leftrightarrow [[q']] = [[q]] \Leftrightarrow \mathcal{L}(q') = \mathcal{L}(q)$$
- jos $q' \in [[q]]$, niin $q' \in F \Leftrightarrow q \in F$ (miksi?)
  – ts. saman lohkon kaikki tilat ovat lopputiloja, tai yksikään niistä ei ole lopputila
- jos $q' \in [[q]]$ ja $a \in \Sigma$, niin $[[\delta(q, a)]] = [[\delta(q', a)]]$ (miksi?)

Lohkottu deterministinen äärellinen automaatti

- olkoon $D = (Q, \Sigma, \delta, \hat{q}, F)$, missä δ on täysi

  *D:stä **lohkomalla** saatu DFA on*
  $[[D]] := (Q', \Sigma, \delta', \hat{q}', F')$, *missä*
  – $Q' = \{ [[q]] \mid q \in Q \}$
  – *jos* $q \in Q$ *ja* $a \in \Sigma$, *niin* $\delta'([[q]], a) = [[\delta(q, a)]]$
  – $\hat{q}' = [[\hat{q}]]$
  – $F' = \{ [[q]] \mid q \in F \}$

- ts. [[D]] on muodostettu D:stä
  - yhdistämällä lohkoiksi D:n tilat, jotka hyväksyvät saman kielen
  - kopioimalla D:n tilojen väliset siirtymät vastaavien lohkojen välisiksi siirtymiksi
  - ottamalla alkutilaksi se lohko, johon D:n alkutila kuuluu
  - ottamalla lopputiloiksi lohkot, joissa on ainakin yksi D:n lopputila
- miksi yllä δ' on täysi ja yksikäsitteinen?
- esimerkki



- vielä on osoitettava, että tulos on oikein!
  - [[D]] on DFA harjoitustehtäväksi (helppo)
  - lohkominen ei saa muuttaa hyväksyttyä kieltä
  - lopputuloksen oltava “pienin mahdollinen”
- itse asiassa [[D]] ei välttämättä ole minimaalinen, mutta osoitamme kohta, että [[$cln(D)$]] on

Hyväksytyn kielen säilyminen lohkonnassa

- δ^*:n ja [[D]]:n määritelmän vuoksi induktiolla
  - $\delta'([[q]], \varepsilon) = [[q]] = [[\delta(q, \varepsilon)]]$
  - $\delta'([[q]], \alpha a) = \delta'(\delta'([[q]], \alpha), a) = \delta'([[\delta(q, \alpha)]], a)$ $= [[\ \delta(\delta(q, \alpha), a)\ ]] = [[\ \delta(q, \alpha a)\ ]]$

$\Rightarrow$ saamme

Jos $q \in Q$ *ja* $\alpha \in \Sigma^*$, *niin* $\delta'([[q]], \alpha) = [[\delta(q, \alpha)]]$.

- koska $q \in F \Leftrightarrow [[q]] \in F'$, saamme
  $\delta(q, \alpha) \in F \Leftrightarrow [[\delta(q, \alpha)]] \in F' \Leftrightarrow \delta'([[q]], \alpha) \in F'$
  mistä seuraa
  $\mathcal{L}(q) = \mathcal{L}([[q]])$
- valitsemalla $q = \hat{q}$ saadaan seuraukseksi lause
  $\mathcal{L}([[D]]) = \mathcal{L}(D)$ ·/·

Edellisen todistuksen hyödyllisiä seurauksia

- lohkotun DFA:n eri tilat hyväksyvät eri kielet
  - todistus: jos $[[q_1]] \neq [[q_2]]$, niin
    $\mathcal{L}([[q_1]]) = \mathcal{L}(q_1) \neq \mathcal{L}(q_2) = \mathcal{L}([[q_2]])$
- jos D ei sisällä saavuttamattomia tiloja, niin $[[D]]$ ei sisällä saavuttamattomia tiloja
  - jokainen $[[D]]$:n tila on muotoa $[[q]]$, missä q on D:n tila
  - jos $q = \delta(\hat{q}, \alpha)$, niin $[[q]] = [[\delta(\hat{q}, \alpha)]] = \delta'([[\hat{q}]], \alpha) = \delta'(\hat{q}', \alpha)$

Lohkonta yhdessä saavuttamattomien tilojen poiston kanssa tuottaa minimin, ja se on yksikäsitteinen

- lause

  Olkoot $D_i = (Q_i, \Sigma, \delta_i, \hat{q}_i, F_i)$ *DFA:ita, missä* $i \in \{1, 2\}$ *siten, että*
  - δ_1 *ja* δ_2 *ovat täysiä funktioita,*
  - $\mathcal{L}(D_1) = \mathcal{L}(D_2)$, *ja*
  - $D_1 = [[cln(D)]]$ *jollekin DFA:lle* D.

  Pätee
  - $|Q_1| \leq |Q_2|$.
  - *Jos* $|Q_1| = |Q_2|$, *niin* D_1 *ja* D_2 *ovat isomorfiset eli tilojen nimiä lukuunottamatta samat.*

- jotta kielten vertaaminen olisi mielekästä, D_1:llä ja D_2:lla on sama aakkosto
- koska $D_1 = [[cln(D)]]$, ei D_1 sisällä saavuttamattomia tiloja

$\Rightarrow$ jokaiselle $q_1 \in Q_1$ on olemassa ainakin yksi $\alpha \in \Sigma^*$ siten, että $q_1 = \delta_1(\hat{q}_1, \alpha)$
  – valitaan jokin sellainen α

- olkoon $q_2 = \delta_2(\hat{q}_2, \alpha)$
- jos $\beta \in \mathcal{L}(q_1)$ ja $\beta \notin \mathcal{L}(q_2)$, niin $\alpha\beta \in \mathcal{L}(D_1)$ ja $\alpha\beta \notin \mathcal{L}(D_2)$
  $\Rightarrow$ $\lightning$ oletuksen $\mathcal{L}(D_1) = \mathcal{L}(D_2)$ kanssa
  $\Rightarrow$ $\mathcal{L}(q_1) \subseteq \mathcal{L}(q_2)$
- jos $\beta \in \mathcal{L}(q_2)$ ja $\beta \notin \mathcal{L}(q_1)$, niin …
  $\Rightarrow$ $\mathcal{L}(q_2) \subseteq \mathcal{L}(q_1)$
  $\Rightarrow$ $\mathcal{L}(q_1) = \mathcal{L}(q_2)$

$\Rightarrow$ jokaiselle $q_1 \in Q_1$ on olemassa ainakin yksi $q_2 \in Q_2$ siten, että $\mathcal{L}(q_1) = \mathcal{L}(q_2)$
  – valitaan niistä yksi ja merkitään sitä $f(q_1)$
  $\Rightarrow$ f on täysi funktio $Q_1 \to Q_2$, ja $\mathcal{L}(f(q_1)) = \mathcal{L}(q_1)$

- jos $q_{1,1} \in Q_1$ ja $q_{1,2} \in Q_1$ ovat eri tiloja, on aiemman tuloksen nojalla $\mathcal{L}(q_{1,1}) \neq \mathcal{L}(q_{1,2})$, koska D_1 on lohkottu
  $\Rightarrow$ $\mathcal{L}(f(q_{1,1})) = \mathcal{L}(q_{1,1}) \neq \mathcal{L}(q_{1,2}) = \mathcal{L}(f(q_{1,2}))$
  $\Rightarrow$ $f(q_{1,1}) \neq f(q_{1,2})$

$\Rightarrow$ $|Q_2| \geq |Q_1|$ $\cdot/\cdot$ (puolet väitteestä)

- viimeisen kohdan osoittamiseksi oletamme tästä eteenpäin, että $|Q_2| = |Q_1|$
- tällöin $Q_2 = \{ f(q_1) \mid q_1 \in Q_1 \}$
  – muuhun eivät Q_2:n tilat riitä

$\Rightarrow$ jos $q_{2,1} \in Q_2$, $q_{2,2} \in Q_2$ ja $q_{2,1} \neq q_{2,2}$, niin
- on olemassa $q_{1,1}$, $q_{1,2}$ siten, että $q_{2,i} = f(q_{1,i})$

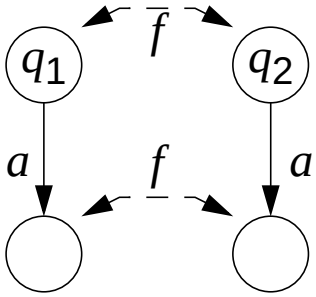
  $\Rightarrow$ $q_{1,1} \neq q_{1,2}$

  $\Rightarrow$ $\mathcal{L}(q_{2,1}) = \mathcal{L}(f(q_{1,1})) = \mathcal{L}(q_{1,1}) \neq \mathcal{L}(q_{1,2}) = \mathcal{L}(q_{2,2})$

- siis jos $q_{2,1} \neq q_{2,2}$, niin $\mathcal{L}(q_{2,1}) \neq \mathcal{L}(q_{2,2})$

$\Rightarrow$ $f(\hat{q}_1) = \hat{q}_2$, koska $\mathcal{L}(f(\hat{q}_1)) = \mathcal{L}(\hat{q}_1) = \mathcal{L}(\hat{q}_2)$

- koska $\mathcal{L}(f(q_1)) = \mathcal{L}(q_1)$, pätee
  - $q_1 \in F_1 \Leftrightarrow f(q_1) \in F_2$
  - jos $a \in \Sigma$, niin $\mathcal{L}(\delta_2(f(q_1), a)) = \mathcal{L}(\delta_1(q_1, a)) = \mathcal{L}(f(\delta_1(q_1, a)))$, joten $\delta_2(f(q_1), a) = f(\delta_1(q_1, a))$



$\Rightarrow$ Q_1 voidaan muuttaa Q_2:ksi korvaamalla järjestelmällisesti tilojen q nimet $f(q)$:lla ·/·

Entä jos tilasiirtymäfunktio ei ole täysi?

- olemme yllä koko ajan olettaneet, että tilasiirtymäfunktio on täysi
- jos emme halua sen olevan täysi, voimme koko ajan kuvitella, että on olemassa ylimääräinen tila q_R, johon puuttuvat tilasiirtymät vievät
  - sitä ei silti tarvitse algoritmissa toteuttaa
- q_R:n toteuttamatta jättämisestä on se hyöty, että siten saadaan usein merkittävästi vähennettyä kaarten määrää ja nopeutettua algoritmia
  - q_R:llä on usein hyvin paljon tulokaaria

- jos q_R olisi oikeasti mukana, se joutuisi lohkoon
  - jonka hyväksymä kieli on Ø
  - jonka lähtökaaret palaavat siihen itseensä
  - jossa ei ole lopputiloja
- jos tällainen lohko syntyy oikeasti
  - siihen liittyvät kaaret voi poistaa
  - sen itse voi poistaa paitsi jos se on alkutila

  ilman että hyväksytty kieli muuttuu
- näin $cln(D)$:stä tehty DFA on minimaalinen vaikka sallittaisiin vajaat tilasiirtymäfunktiot
  - vielä pienemmän täydentäminen täydeksi tuottaisi pienemmän täyden DFA:n kuin täysien DFA:iden minimointialgoritmi

Minimointialgoritmi

- $cln(D)$ on helppo laskea graafihakualgoritmilla
- $[[D]]$ näyttää vaikealta laskea
  - miten testata, hyväksyykö kaksi tilaa saman kielen?
- onneksi $[[D]]$:n laskemiseksi on olemassa tehokas algoritmi
  - tilojen joukko jaetaan pistevieraisiin lohkoihin
  - lohkoja jaetaan kahtia tietyn säännön mukaan
  - kun mikään lohko ei enää jakaannu, on $[[D]]$ valmis
  - nimi: *lohkomisalgoritmi*
- olkoon $D = (Q, \Sigma, \delta, \hat{q}, F)$
- algoritmin alussa on kaksi lohkoa: F ja $Q - F$
- merkitään lohkoa, johon tila q kuuluu $block(q)$

- lohko B jaetaan aakkosen $a \in \Sigma$ avulla seuraavasti:
  - jos $block(\delta(q, a))$ on sama kaikilla $q \in B$, niin B ei jakaannu a:n avulla
  - muutoin valitaan jokin $q \in B$, ja jaetaan B seuraavasti:

    $B_1 := \{ q' \in B \mid block(\delta(q', a)) = block(\delta(q, a)) \}$

    $B_2 := B - B_1$
- vaikka lohko ei jonain hetkenä jakaantuisi a:n avulla, se saattaa myöhemmin muuttua a:n avulla jakaantuvaksi kun muita lohkoja jaetaan

  $\Rightarrow$ lopettaa saa vasta kun on varmaa, että mikään lohko ei jakaannu minkään aakkosen avulla
- näillä keinoin on helppo suunnitella algoritmi, jonka ajan käyttö tavallisessa tietokoneessa on $O(|Q|^2 \cdot |\Sigma|)$
- tunnetaan keino järjestää lohkominen siten, että työ etenee “lohkomistarpeen” mukaisesti

  $\Rightarrow$ vielä tehokkaampi algoritmi

Lohkomisalgoritmin oikeellisuus

- alkujaossa ε kuuluu lohkon F tilojen kieliin, mutta ei lohkon $Q - F$
- jos B:tä jaettaessa $block(\delta(q_1, a)) \neq block(\delta(q_2, a))$, niin (induktio) on olemassa $\alpha \in \Sigma^*$ siten, että $\alpha \in \mathcal{L}(\delta(q_1, a))$ ja $\alpha \notin \mathcal{L}(\delta(q_2, a))$ tai päinvastoin, joten $a\alpha \in \mathcal{L}(q_1)$ ja $a\alpha \notin \mathcal{L}(q_2)$ tai päinvastoin

$\Rightarrow$ koko ajan pätee:

jos $block(q_1) \neq block(q_2)$, niin $\mathcal{L}(q_1) \neq \mathcal{L}(q_2)$

- jos $\mathcal{L}(q_1) \neq \mathcal{L}(q_2)$, niin on olemassa $a_1 a_2 \ldots a_n \in \Sigma^*$ siten, että $a_1 a_2 \ldots a_n \in \mathcal{L}(q_1)$ ja $a_1 a_2 \ldots a_n \notin \mathcal{L}(q_2)$ tai päinvastoin

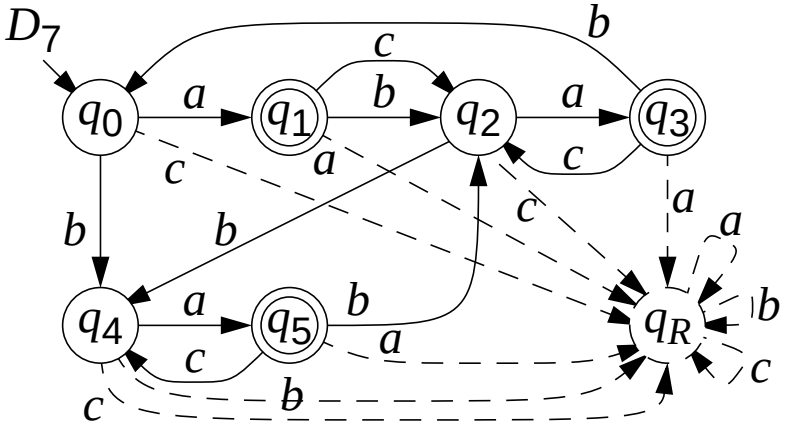
- todistamme induktiolla, että tällöin lopussa $block(q_1) \neq block(q_2)$
- pohja: $\varepsilon \in \mathcal{L}(\delta(q_1, a_1a_2\ldots a_n))$ ja $\varepsilon \notin \mathcal{L}(\delta(q_2, a_1a_2\ldots a_n))$, joten alkujaon vuoksi $block(\delta(q_1, a_1a_2\ldots a_n)) \neq block(\delta(q_2, a_1a_2\ldots a_n))$
- jos $block(\delta(q_1, a_1\ldots a_{i+1})) \neq block(\delta(q_2, a_1\ldots a_{i+1}))$ mutta $block(\delta(q_1, a_1\ldots a_i)) = block(\delta(q_2, a_1\ldots a_i))$, niin lohkomista ei ole saatettu loppuun

$\Rightarrow$ lopussa pätee:
jos $\mathcal{L}(q_1) \neq \mathcal{L}(q_2)$, niin $block(q_1) \neq block(q_2)$

- siis lopussa $\mathcal{L}(q_1) \neq \mathcal{L}(q_2) \Leftrightarrow block(q_1) \neq block(q_2)$, eli kaikille tiloille q pätee $block(q) = [[q]]$ ·/·

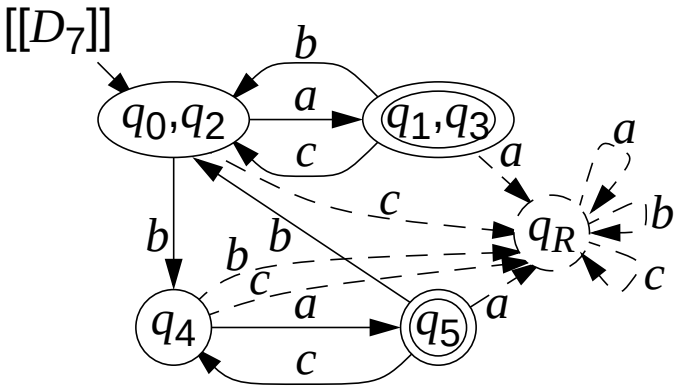
Esimerkki

- minimoitava D_7



- cln-operaatio ei muuta mitään
- aluksi lohkot ovat $\{q_1, q_3, q_5\}$ ja $\{q_0, q_2, q_4, q_R\}$
- $\{q_1, q_3, q_5\}$ ei jakaannu minkään aakkosen avulla
- $\{q_0, q_2, q_4, q_R\}$ jakaantuu a:n avulla lohkoiksi $\{q_R\}$ ja $\{q_0, q_2, q_4\}$
- $\{q_0, q_2, q_4\}$ jakaantuu b:n avulla $\{q_0, q_2\}$ ja $\{q_4\}$
- nyt $\{q_1, q_3, q_5\}$ jakaantuu c:n avulla $\{q_1, q_3\}$ ja $\{q_5\}$

- nyt mikään lohko ei enää jakaannu
  $\Rightarrow$ lopputulos on



Deterministisen äärellisten automaattien hyväksymien kielten vertailu

- lohkomisalgoritmin avulla on helppo testata, hyväksyykö kaksi DFA:ta saman kielen
  - pannaan DFA:t vierekkäin ikään kuin yhdeksi DFA:ksi, ajetaan lohkomisalgoritmi, ja katsotaan, päätyvätkö alkutilat samaan vai eri lohkoon

$\Rightarrow$ päätöstehtävä “On annettu DFA:t D_1 ja D_2 joilla on sama aakkosto; onko $\mathcal{L}(D_1) = \mathcal{L}(D_2)$?” voidaan ratkaista tietokoneella varsin nopeasti
  - polynomiaalisessa ajassa $|Q|$:n ja $|\Sigma|$:n suhteen

Tuloautomaatti

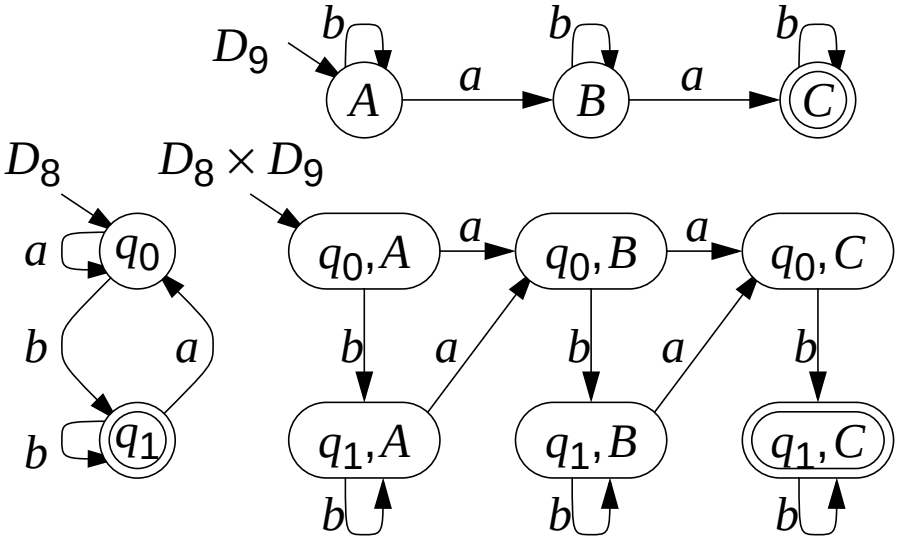
- tuloautomaatti on keino muodostaa DFA, jonka hyväksymä kieli on kahden DFA:n kielten leikkaus
  $\Rightarrow$ yhdistää osa-DFA:iden asettamat rajoitteet
- kertausta: joukkojen tulo määritellään

$$A \times B := \{ (a, b) \mid a \in A \wedge b \in B \}$$

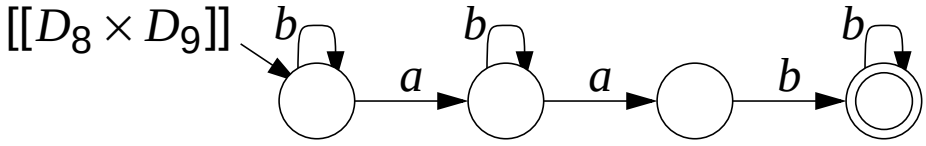
  - ts. kaikkien parien joukko, joiden 1. alkio on A:sta ja 2. alkio B:stä
  - esim. $\{0, 1\} \times \{a, b, c\} =$
    $\{ (0,a), (0,b), (0,c), (1,a), (1,b), (1,c) \}$

- määritelmä

  Olkoot $D_i = (Q_i, \Sigma, \delta_i, \hat{q}_i, F_i)$, $i \in \{1, 2\}$, *DFA:ita. Niiden* ***tulo*** $D_1 \times D_2 := (Q, \Sigma, \delta, \hat{q}, F)$, *missä*
  - $Q = Q_1 \times Q_2$,
  - $\delta((q_1, q_2), a) = \begin{cases} (\delta(q_1, a), \delta(q_2, a)), \textit{ jos } \delta(q_1, a) \textit{ ja } \delta(q_2, a) \textit{ on määritelty} \\ \textit{määrittelemätön, muutoin} \end{cases}$
  - $\hat{q} = (\hat{q}_1, \hat{q}_2)$, *ja*
  - $F = F_1 \times F_2$.
- nytkin vaaditaan, että D_1:llä ja D_2:lla sama aakkosto
- pätee: $\mathcal{L}(D_1 \times D_2) = \mathcal{L}(D_1) \cap \mathcal{L}(D_2)$
  - todistus sivuutetaan vaikka on helppo
- esimerkki
  - D_8 hyväksyy b:hen päättyvät ab-jonot
  - D_9 hyväksyy tasan 2 a:ta sisältävät ab-jonot
  - $D_8 \times D_9$ hyväksyy tasan 2 a:ta sisältävät b:hen päättyvät ab-jonot



- näytetään vielä $[[D_8 \times D_9]]$



Testi “onko $\mathcal{L}(D_1) \subseteq \mathcal{L}(D_2)$?”

- olkoot D_1 ja D_2 DFA:ita, joilla on sama aakkosto
- esimerkiksi ohjelmien verifioinnissa on joskus tarpeen testata, onko $\mathcal{L}(D_1) \subseteq \mathcal{L}(D_2)$
  - ts. hyväksyykö D_2 jokaisen merkkijonon, jonka D_1 hyväksyy
- jos D on DFA, on hyvin helppo tarkastaa, onko $\mathcal{L}(D) = \emptyset$
  - $\mathcal{L}(D) \neq \emptyset$, jos ja vain jos D:n alkutilasta on polku johonkin D:n lopputilaan
- joukko-opista tiedämme, että jos U on perusjoukko joka kattaa A:n ja B:n, niin

  $A \subseteq B \;\Leftrightarrow\; A \cap (U - B) = \emptyset$

$\Rightarrow$ $\mathcal{L}(D_1) \subseteq \mathcal{L}(D_2)$, jos ja vain jos

$\mathcal{L}(D_1) \cap (\Sigma^* - \mathcal{L}(D_2)) = \emptyset$

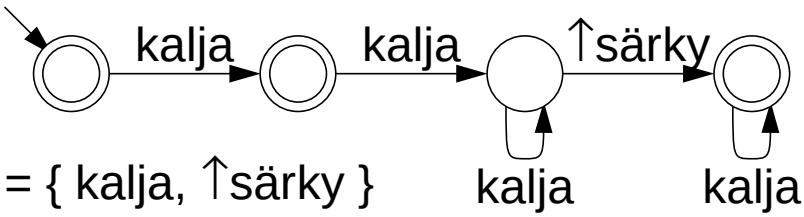
$\Rightarrow$ $\mathcal{L}(D_1) \subseteq \mathcal{L}(D_2)$, jos ja vain jos

$\mathcal{L}(D_1 \times compl(D_2)) = \emptyset$

- koska $compl(D)$, $D_1 \times D_2$ ja $\mathcal{L}(D) = \emptyset$ ovat helppoja laskea, on tämä keino tehokas ja helppo toteuttaa

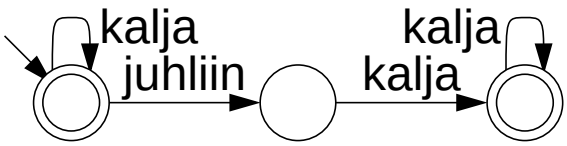
Esimerkki

- jos (ja vain jos) juo ≥ 2 kaljaa, niin alkaa särkeä päätä



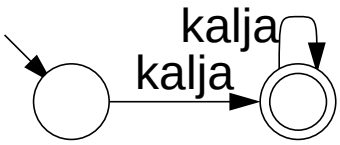
Σ = { kalja, ↑särky }

- kotonakin voi juoda, mutta juhlissa on pakko juoda ainakin yksi kalja

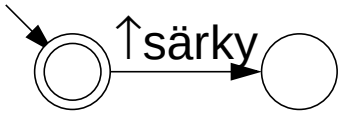


Σ = { kalja, juhliin }

- kaljaa halutaan, mutta päänsärkyä ei

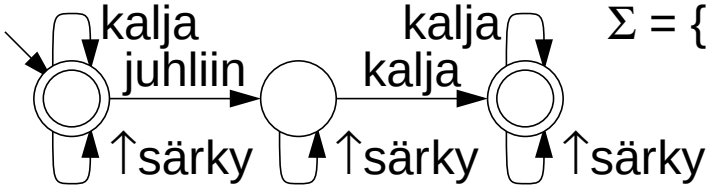


Σ = { kalja }



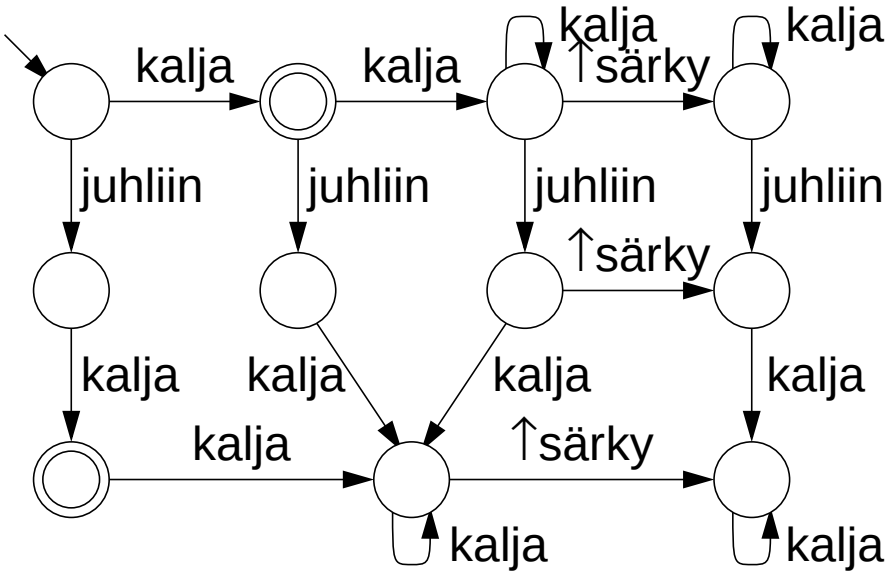
Σ = { ↑särky }

- mitä pitää tehdä?
- laajennetaan aakkostot samoiksi lisäämällä paikallissilmukoita oheisen esimerkin mukaan

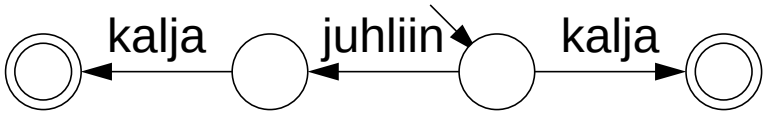


Σ = { kalja, juhliin, ↑särky }

- lasketaan *cln*(tuloautomaatti)



- poistetaan turhat tilat ja tilasiirtymät
  - turha = ei polulla alkutilasta lopputilaan



⇒ joko oltava ilta kotona yhden kaljan kanssa, tai juotava juhlissa yksi kalja ilman pohjia kotona

- lopputuloksen olisi saanut helpommin laskemalla tuloautomaatin vaiheittain ja poistamalla turhat tilat ja tilasiirtymät joka välissä
  - harjoitustehtävä
- kehittyneempiä tämäntapaisia menetelmiä käytetään mm. tietoliikenneprotokollien tutkimuksessa

5.4 Epädeterministiset äärelliset automaatit

Epädeterministinen äärellinen automaatti on muuten sama kuin deterministinen, mutta

- samalla tilalla ja aakkosella saa olla > 1 siirtymää
- sallitaan ns. ε-siirtymät, joissa ei lueta yhtään merkkiä

$\Rightarrow$ helpompi suunnitella kuin deterministisiä (vrt. harjoitustehtävät)

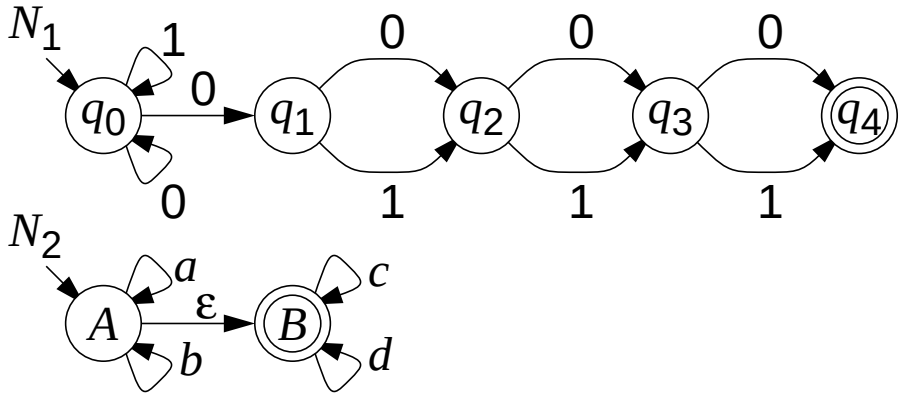
Määritelmä

- saadaan korvaamalla deterministisen äärellisen automaatin määritelmässä osittainen tilasiirtymäfunktio δ tilasiirtymärelaatiolla Δ

  Epädeterministinen äärellinen automaatti (nondeterministic finite automaton) (NFA)
  $(Q, \Sigma, \Delta, \hat{q}, F)$ *sisältää viisi osaa:*
  - Q: *äärellinen joukko* ***tiloja (state),***
  - Σ, *jolle* $\varepsilon \notin \Sigma$: *äärellinen* ***syöteaakkosto (input alphabet),***
  - $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$: ***tilasiirtymärelaatio (transition relation),***
  - $\hat{q} \in Q$: ***alkutila (initial state),*** *ja*
  - $F \subseteq Q$: ***lopputilat (final states).***

NFA:n voi esittää kuvana samaan tapaan kuin DFA:n

- esimerkkejä



Epädeterministisen äärellisen automaatin toiminta

- epädeterministisen äärellisen automaatin N toiminta on muuten samanlainen kuin deterministisen, mutta
  - jos luettu merkki on a ja nykyisestä tilasta on useita a-siirtymiä, N voi valita niistä minkä vain
  - jos nykyisestä tilasta on ε-siirtymä, N voi tehdä sen lukematta yhtään merkkiä
- määritellään tilasiirtymärelaation laajennos "—α→" merkkijonoille pienimpänä relaationa, joka täyttää seuraavat ehdot kaikille q, $q' \in Q$, $a \in \Sigma$ ja α, β ∈ Σ*:
  - q —ε→ q
  - jos $(q, \varepsilon, q') \in \Delta$, niin q —ε→ q'
  - jos $(q, a, q') \in \Delta$, niin q —a→ q'
  - jos on olemassa q_1 siten, että q —α→ q_1 ja q_1 —β→ q', niin q —αβ→ q'
- toisin sanoen, q —α→ q', jos ja vain jos tilasta q on polku tilaan q' siten, että kun polun varren aakkoset kirjoitetaan järjestyksessä peräkkäin, niin tulos on α
  - ε-symboleita ei kirjoiteta, koska ne edustavat tyhjää

- esimerkiksi N_1:ssä
  - $q_0 \text{—}\varepsilon\rightarrow q_0$
  - $q_0 \text{—}00\rightarrow q_0$
  - $q_0 \text{—}00\rightarrow q_1$
  - $q_0 \text{—}00\rightarrow q_2$
  - $q_0 \text{—}10111\rightarrow q_0$
  - $q_0 \text{—}10111\rightarrow q_4$
- esimerkiksi N_2:ssä
  - $A \text{—}\varepsilon\rightarrow B$
  - $A \text{—}a\rightarrow A$
  - $A \text{—}a\rightarrow B$
  - $A \text{—}abc\rightarrow B$
- kun puhutaan useasta NFA:sta yhtäaikaa, alaindekseillä voi osoittaa minkä "—α→"-relaatiosta on kyse
  - esim. $q_0 \text{—}10111\rightarrow_1 q_0$ ja $A \text{—}abc\rightarrow_2 B$

NFA:n hyväksymä kieli

- määritelmä

  Epädeterministinen äärellinen automaatti $N = (Q, \Sigma, \Delta, \hat{q}, F)$ ***hyväksyy (accepts)*** *merkkijonon* $\alpha \in \Sigma^*$, *jos ja vain jos on olemassa* $q \in F$ *siten, että* $\hat{q} \text{—}\alpha\rightarrow q$. *N:n* ***hyväksymä kieli*** *on*

  $$\mathcal{L}(N) := \{\ \alpha \in \Sigma^* \mid \exists\ q \in F\colon \hat{q} \text{—}\alpha\rightarrow q\ \}.$$

- siis jos α:lla voi jotenkin kulkea alkutilasta johonkin lopputilaan, hyväksyy N α:n, vaikka α:lla olisi mahdollista kulkea myös ei-lopputilaan
  - *ystävällinen* epädeterminismi
- esimerkiksi
  - $\mathcal{L}(N_1) = \mathcal{L}(\ (0 \mid 1)^*0(0 \mid 1)(0 \mid 1)(0 \mid 1)\ )$
  - $\mathcal{L}(N_2) = \mathcal{L}(\ (a \mid b)^*(c \mid d)^*\ )$

DFA:t voidaan tulkita NFA:iden erikoistapaukseksi

- $(Q, \Sigma, \delta, \hat{q}, F)$ muutetaan $(Q, \Sigma, \Delta, \hat{q}, F)$:ksi määrittelemällä

  $$(q, a, q') \in \Delta \;:\Leftrightarrow\; (\,\delta(q, a) \text{ on määritelty }) \wedge q' = \delta(q, a)$$

  – ts. täysin samat tilasiirtymät hieman eri formalismissa

$\Rightarrow$ jokaisen kielen, jonka voi hyväksyä DFA:lla, voi hyväksyä myös NFA:lla

ε-kaarten tarpeellisuus

- ε-kaaret voidaan poistaa NFA:sta muuttamatta sen hyväksymää kieltä siten, että vain tilasiirtymärelaatiota ja lopputilojen joukkoa tarvitsee muuttaa

  Olkoon $N = (Q, \Sigma, \Delta, \hat{q}, F)$ *epädeterministinen äärellinen automaatti. On olemassa epädeterminstinen äärellinen automaatti* $N' = (Q, \Sigma, \Delta', \hat{q}, F')$ *siten, että* $\Delta' \subseteq Q \times \Sigma \times Q$ *ja* $\mathcal{L}(N') = \mathcal{L}(N)$.

- todistus harjoitustehtäväksi

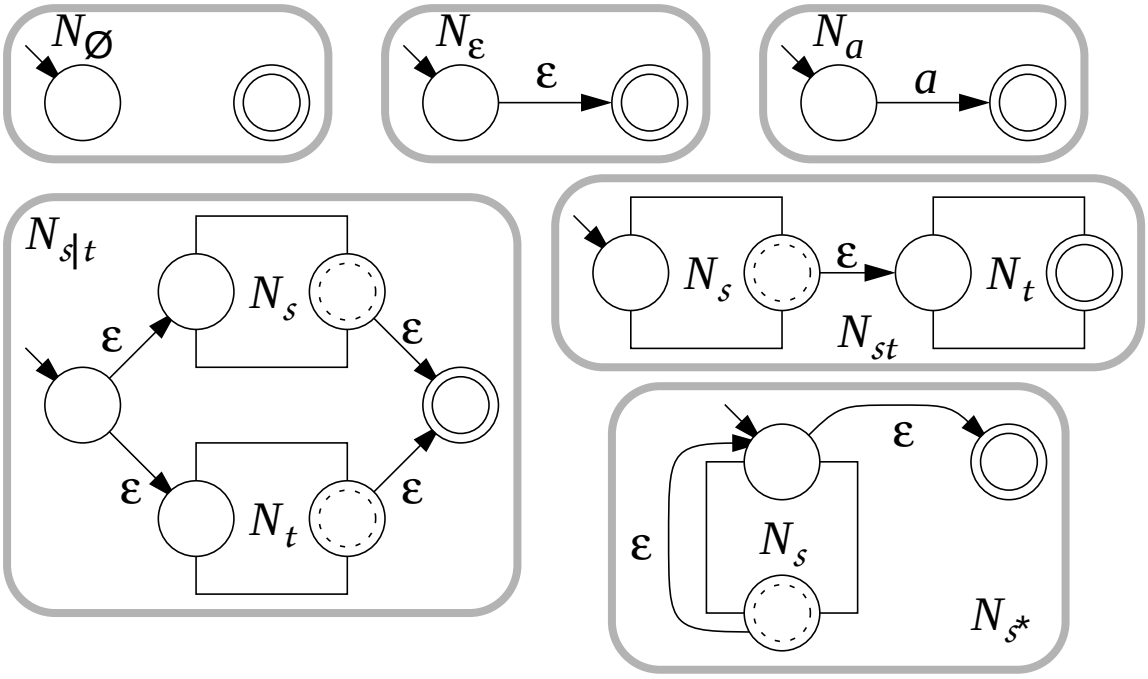
$\Rightarrow$ myös ε-kaaret ovat vain mukavuustekijä

Säännöllisten lausekkeiden ja epädeterminististen äärellisten automaattien välinen suhde, osa 1

- lause

  Olkoon r *säännöllinen lauseke. On olemassa epädeterminstinen äärellinen automaatti* N *siten, että* $\mathcal{L}(N) = \mathcal{L}(r)$, *ja lisäksi*
  – N*:llä on tasan yksi lopputila,*
  – N*:n tilojen määrä on* $\leq 2 \cdot |r|$, *ja*
  – N *voidaan muodostaa polynomiaalisessa ajassa.*

- todistus: eräs vaatimukset täyttävä N_r voidaan rakentaa r:n ohjaamana seuraavasti: ·/·



Säännöllisten lausekkeiden ja epädeterminististen äärellisten automaattien välinen suhde, osa 2

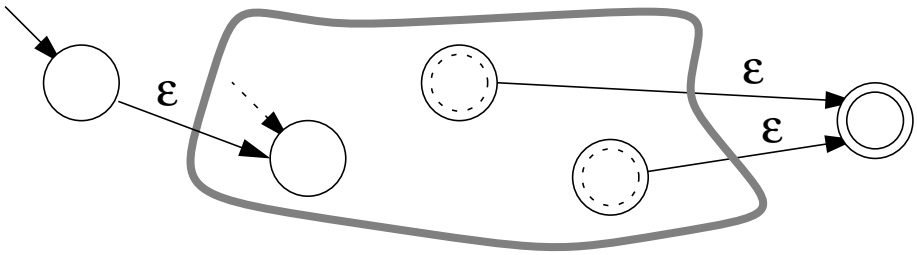
- lause

  Olkoon N epädeterministinen äärellinen automaatti. On olemassa säännöllinen lauseke r siten, että $\mathcal{L}(N) = \mathcal{L}(r)$.

- todistuksessa käytetään NFA:iden muunnosta, jossa kaaren nimenä saa olla yksittäisen aakkosen sijasta säännöllinen lauseke

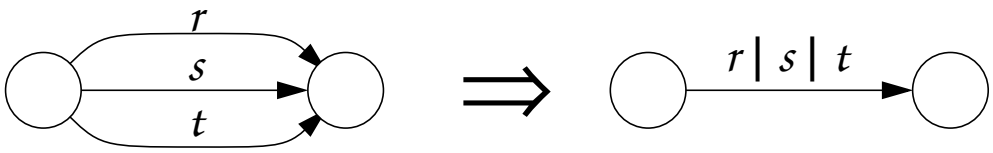
- N muutetaan tällaiseksi automaatiksi seuraavasti:

1. poistetaan N:stä kaikki tilat, jotka eivät ole jollain alkutilasta lopputilaan johtavalla polulla
2. luodaan N:lle uusi alkutila ja lopputila
   – uudesta alkutilasta vedetään ε-kaari vanhaan
   – vanhoista lopputiloista vedetään ε-kaaret uuteen
   – vanhat lopputilat muutetaan ei-lopputiloiksi

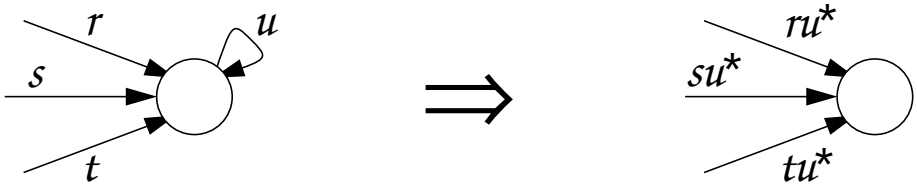


⇒ pätee
   – alkutilaan ei tule kaaria
   – on vain yksi lopputila, ja siitä ei lähde kaaria

3. jos samojen tilojen välillä on samaan suuntaan ≥ 2 kaarta, ne yhdistetään seuraavasti:

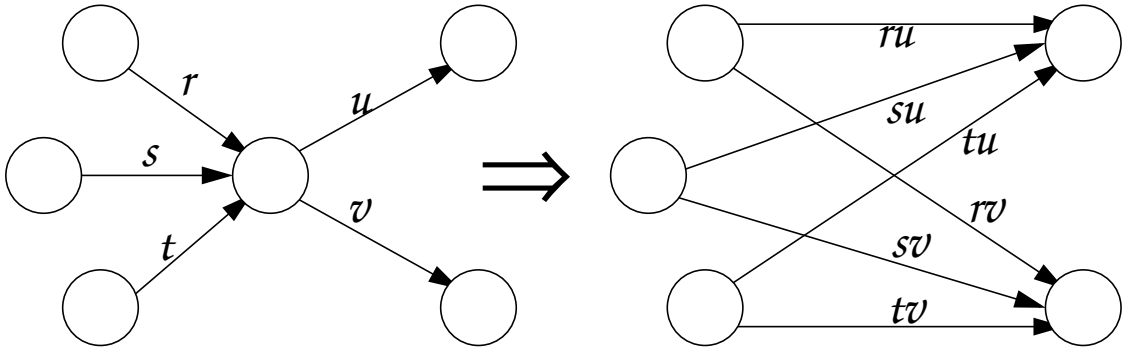


4. jos johonkin tilaan liittyy paikallinen silmukka, se hävitetään seuraavasti:

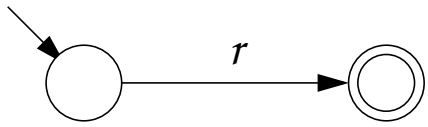


   – muista sulkujen lisäys sivulta 134
   – esim. jos $r = a$ ja $u = bc$, niin $ru^* = a(bc)^*$

5a. jos on olemassa tila, joka ei ole alku- eikä lopputila, jokin sellainen tila hävitetään seuraavasti ja palataan kohtaan 3:



5b. muutoin automaatti on kuvan muotoa, ja r määrittelee sen kielen, minkä alkuperäinen automaatti hyväksyy ·/·



Johtopäätös: säännölliset lausekkeet määrittelevät tarkalleen ne kielet, jotka NFA:t hyväksyvät

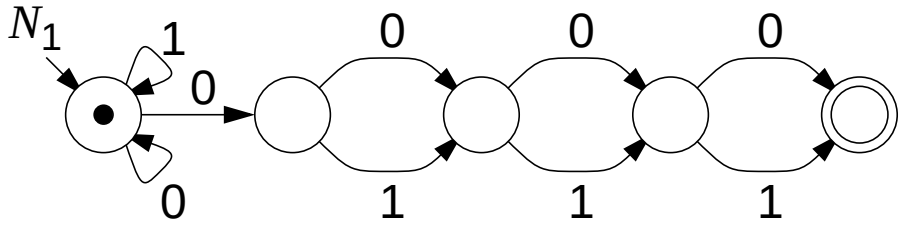
- tiedämme lisäksi, että NFA:t hyväksyvät ainakin kaikki ne kielet, jotka DFA:t hyväksyvät

$\Rightarrow$ hyväksyvätkö NFA:t enemmän kieliä kuin DFA:t?

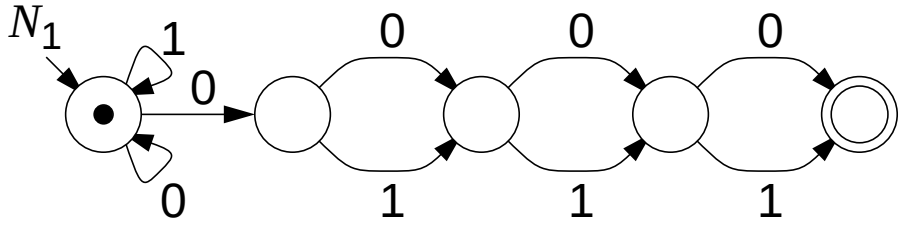
- käymme tutkimaan NFA:n determinististä *simulointia*

Epädeterministisen äärellisen automaatin toiminnan deterministinen simulointi

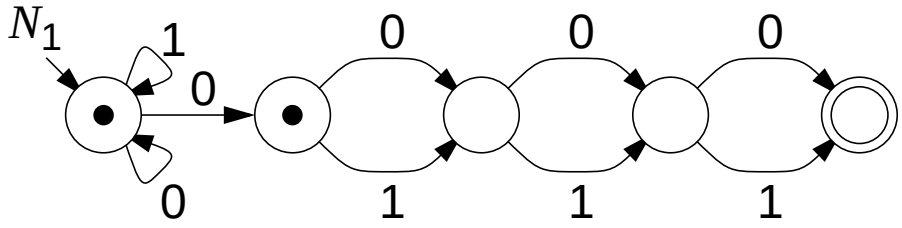
- päätöstehtävä NFA-HYVÄKSYNTÄ: “Hyväksyykö annettu epädeterministinen äärellinen automaatti annetun merkkijonon” saattaa tuntua vaikealta
- se on kuitenkin todellisuudessa helppo: simuloidaan NFA N:n toiminta siten, että pidetään kirjaa **kaikista** niistä tiloista, jossa N **saattaa olla** luettuaan tietyn merkkijonon
- esimerkki: hyväksyykö N_1 jonon 10100
- alkutilanne: luettu ε, lukematta 10100



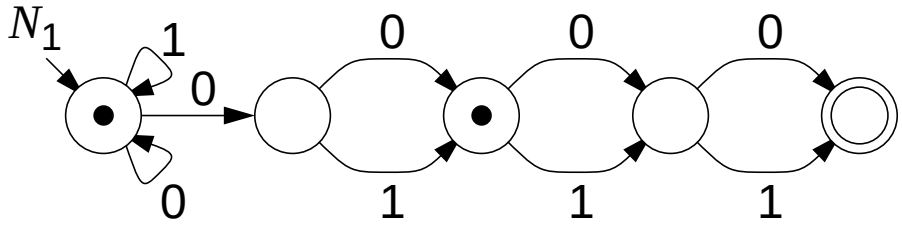
- luettu 1, lukematta 0100



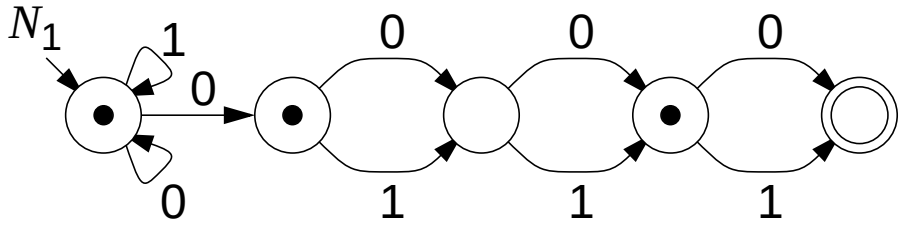
- luettu 10, lukematta 100



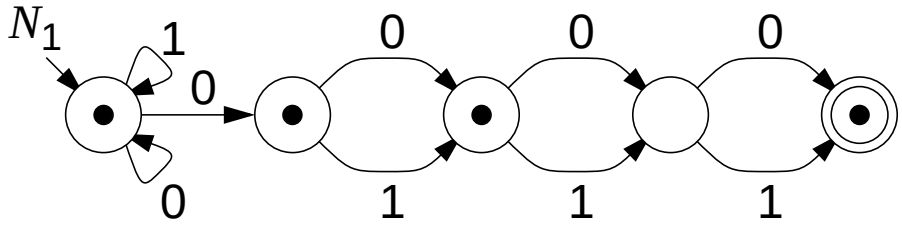
- luettu 101, lukematta 00



- luettu 1010, lukematta 0

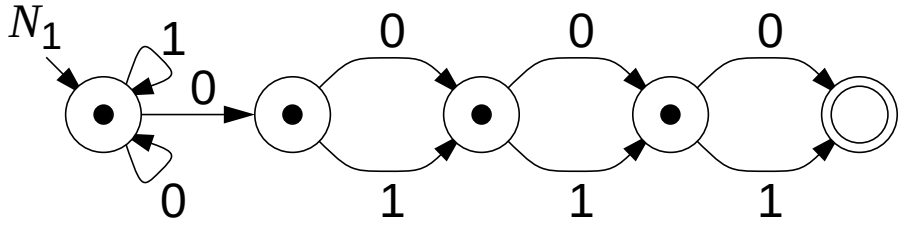


- luettu 10100, lukematta ε



$\Rightarrow$ 10100 $\in \mathcal{L}(N_1)$

- esimerkki: hyväksyykö N_1 jonon 101000
- luettu 101000, lukematta ε



$\Rightarrow$ 101000 $\notin \mathcal{L}(N_1)$

- tässä simuloinnissa automaatin tila korvautuu mahdollisten tilojen joukolla
  - Q:n osajoukko

Simuloinnin formalisointi

- edellä esitetty simulointi voidaan formalisoida määrittelemällä funktio, joka tuottaa niiden tilojen joukon, joissa NFA voi olla luettuaan tietyn merkkijonon
- määritelmästä on jatkossa enemmän hyötyä, jos sallimme simuloinnin lähtökohdaksi minkä tahansa tilan tai osajoukon tiloja
- määritelmä

  Olkoon $N = (Q, \Sigma, \Delta, \hat{q}, F)$ *NFA,* $q \in Q$, $\alpha \in \Sigma^*$ *ja* $K \subseteq Q$.
  Määrittelemme funktiot $\delta\colon Q \times \Sigma^* \to 2^Q$ *ja*
  $\delta\colon 2^Q \times \Sigma^* \to 2^Q$ *seuraavasti:*
  - $\delta(q, \alpha) := \{ q' \in Q \mid q \text{—}\alpha\to q' \}$
  - $\delta(K, \alpha) := \{ q' \in Q \mid \exists\, q \in K\colon q \text{—}\alpha\to q' \}$.
- tässä käytetään samaa nimeä "δ" kahdelle muodollisesti erille, mutta samantapaiselle funktiolle!
- δ:jen määritelmästä seuraa mm.
  - $\delta(q, \alpha) = \delta(\{q\}, \alpha)$
  - α:n simulointi alkutilasta vie tiloihin $\delta(\hat{q}, \alpha)$
  - $\alpha \in \mathcal{L}(N) \Leftrightarrow \delta(\hat{q}, \alpha) \cap F \neq \emptyset$
  - $\delta(K, \alpha\beta) = \delta(\, \delta(K, \alpha), \beta\,)$
- kaava $\delta(K, \alpha\beta) = \delta(\, \delta(K, \alpha), \beta\,)$ on
  - jatkossa tärkeä
  - vaikka ei yllättävä, ei silti ihan itsestään selvä

  $\Rightarrow$ todistamme sen huolellisesti
- jos $q_2 \in \delta(K, \alpha\beta)$, niin $\exists\, q_0 \in K\colon q_0 \text{—}\alpha\beta\to q_2$
  $\Rightarrow \exists\, q_1 \in Q\colon q_0 \text{—}\alpha\to q_1 \wedge q_1 \text{—}\beta\to q_2$
  - koska $q_0 \in K$, pätee $q_1 \in \delta(K, \alpha)$
  - koska $q_1 \in \delta(K, \alpha)$ ja $q_1 \text{—}\beta\to q_2$, pätee $q_2 \in \delta(\, \delta(K, \alpha), \beta\,)$

$\Rightarrow\ \delta(K, \alpha\beta) \subseteq \delta(\delta(K, \alpha), \beta)$

- jos $q_2 \in \delta(\delta(K, \alpha), \beta)$, niin $\exists\, q_1 \in \delta(K, \alpha)$: $q_1 \text{—}\beta\rightarrow q_2$
  - koska $q_1 \in \delta(K, \alpha)$, niin $\exists\, q_0 \in K$: $q_0 \text{—}\alpha\rightarrow q_1$
  $\Rightarrow\ \exists\, q_1$: $q_0 \text{—}\alpha\rightarrow q_1 \wedge q_1 \text{—}\beta\rightarrow q_2\ \Rightarrow\ q_0 \text{—}\alpha\beta\rightarrow q_2$
  - koska $q_0 \in K$, saamme $q_2 \in \delta(K, \alpha\beta)$

$\Rightarrow\ \delta(\delta(K, \alpha), \beta) \subseteq \delta(K, \alpha\beta)$ ja
$\delta(K, \alpha\beta) = \delta(\delta(K, \alpha), \beta)$ ·/·

Epädeterminististen äärellisen automaatin deterministisointi

- edellä esitelty simulointi sisältää
  - tilanteita, jotka ovat Q:n osajoukkoja
  - siirtymiä tilanteiden välillä luetun merkin mukaan

$\Rightarrow$ ehkäpä NFA:n käyttäytyminen **kaikilla** merkkijonoilla voitaisiin simuloida esittämällä ym. tilanteet ja siirtymät DFA:na?

- olkoon $N = (Q_N, \Sigma, \Delta_N, \hat{q}_N, F_N)$ NFA; tavoitteena on suunnitella DFA $Det(N) = (Q_D, \Sigma, \delta_D, \hat{q}_D, F_D)$, joka hyväksyy saman kielen kuin N
- sekaannuksen välttämiseksi merkitsemme N:ään liittyviä, joukkoja tuottavia tilasiirtymäfunktioita δ_N
- N:n simuloinnissa syntyvien mahdollisten tilanteiden joukko on
  $$Q_D = \{\, \delta_N(\hat{q}_N, \alpha) \mid \alpha \in \Sigma^* \,\}$$
- siis jos $q \in Q_D$, niin $q \subseteq Q_N$
- $a \in \Sigma$ vie tilanteesta $q_D \in Q_D$ tilanteeseen $\delta_N(q_D, a)$

$\Rightarrow$ $Det(N)$:n tilasiirtymärelaatio δ_D saadaan todella helposti:
  $$\delta_D(q_D, a) := \delta_N(q_D, a)$$
  - näin määritelty δ_D on täysi funktio

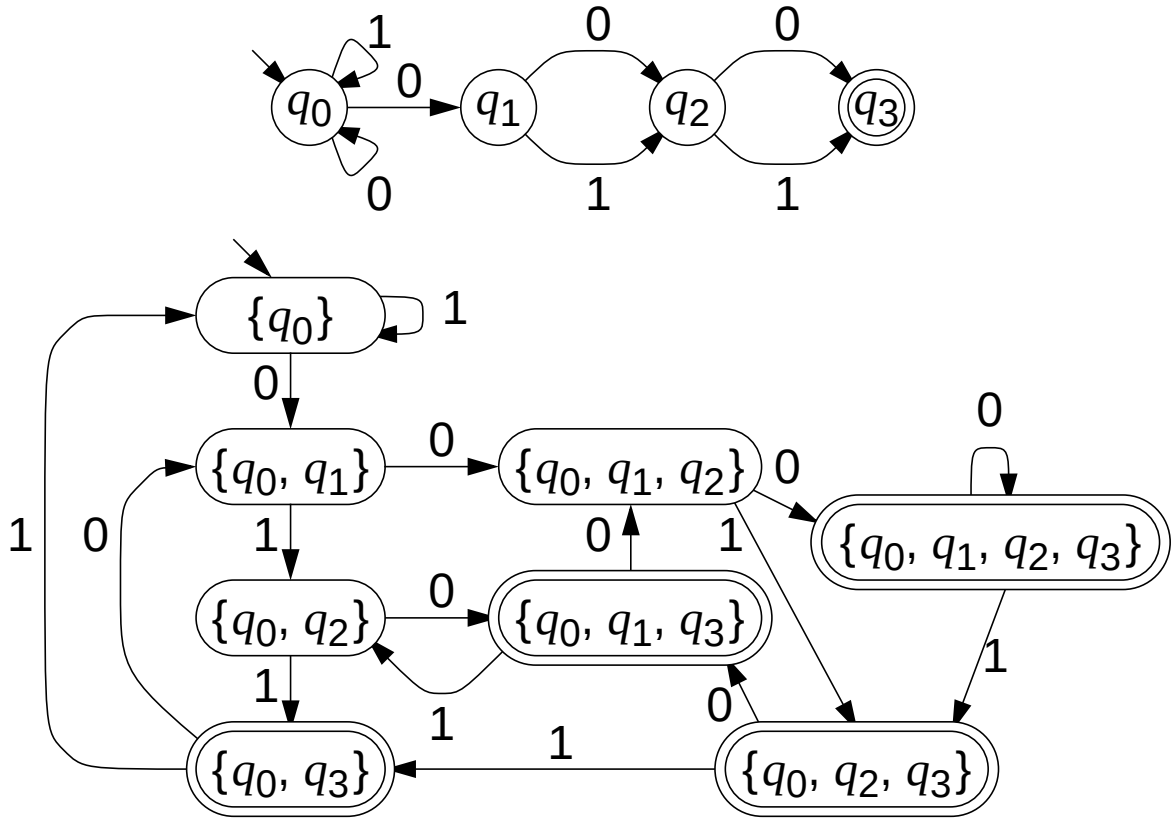
- $\hat{q}_D$ on tietysti ne tilat, joihin N pääsee alkutilastaan lukematta ainuttakaan merkkiä
  $\hat{q}_D := \delta_N(\hat{q}_N, \varepsilon)$
- N hyväksyy merkkijonon jos ja vain jos **jokin** laskenta johtaa lopputilaan

$\Rightarrow$ määrittelemme $F_D := \{ q_D \in Q_D \mid q_D \cap F_N \neq \emptyset \}$

- yhteenveto

  Jos $N = (Q_N, \Sigma, \Delta_N, \hat{q}_N, F_N)$ *on NFA, niin*
  Det$(N) := (Q_D, \Sigma, \delta_D, \hat{q}_D, F_D)$, *missä*
  - $Q_D = \{ \delta_N(\hat{q}_N, \alpha) \mid \alpha \in \Sigma^* \}$
  - $\forall\, q \in Q_D$: $\forall\, a \in \Sigma$: $\delta_D(q, a) = \delta_N(q, a)$
  - $\hat{q}_D = \delta_N(\hat{q}_N, \varepsilon)$
  - $F_D = \{ q_D \in Q_D \mid q_D \cap F_N \neq \emptyset \}$.
- *Det*(N):n osoittaminen DFA:ksi jätetään harjoitustehtäväksi

Esimerkki



Deterministisoinnin oikeellisuus

- vaikka edellä rakensimme DFA:n $Det(N) = D\ N$:stä simulointi-intuition ohjaamana, ei silti vielä ole varmaa, että ne hyväksyvät saman kielen

$\Rightarrow$ todistamme, että $\mathcal{L}(D) = \mathcal{L}(N)$

- simulaation oikeellisuuden avainasemassa on väite, että jokainen merkkijono α vie D:n juuri siihen tilaan, joka on niiden tilojen joukko, johon α veisi N:n
  - ts. $\forall\ \alpha \in \Sigma^*$: $\delta_D(\hat{q}_D, \alpha) = \delta_N(\hat{q}_N, \alpha)$
- todistamme tämän induktiolla α:n pituuden suhteen, eli todistamme, että
  - $\delta_D(\hat{q}_D, \varepsilon) = \delta_N(\hat{q}_N, \varepsilon)$
  - jos $\delta_D(\hat{q}_D, \alpha) = \delta_N(\hat{q}_N, \alpha)$ pätee ja $a \in \Sigma$, niin $\delta_D(\hat{q}_D, \alpha a) = \delta_N(\hat{q}_N, \alpha a)$
- induktion pohja

  $\delta_D(\hat{q}_D, \varepsilon)$ – DFA:lle $\delta(q, \varepsilon) = q$
  $= \hat{q}_D$ – $\hat{q}_D$:n määritelmä
  $= \delta_N(\hat{q}_N, \varepsilon)$
- induktioaskel

  $\delta_D(\hat{q}_D, \alpha a)$ – $\delta_D(q, \alpha a) = \delta_D(\delta_D(q,\alpha), a)$
  $= \delta_D(\delta_D(\hat{q}_D, \alpha), a)$ – induktio-oletus
  $= \delta_D(\delta_N(\hat{q}_N, \alpha), a)$ – δ_D:n määritelmä
  $= \delta_N(\delta_N(\hat{q}_N, \alpha), a)$ – $\delta_N(K, \alpha\beta) = \delta_N(\delta_N(K,\alpha), \beta)$
  $= \delta_N(\hat{q}_N, \alpha a)$
- nyt voimme viimeistellä todistuksen:

  $\alpha \in \mathcal{L}(D)$ – $\mathcal{L}(D)$:n määritelmä
  $\Leftrightarrow\ \delta_D(\hat{q}_D, \alpha) \in F_D$ – F_D:n määritelmä
  $\Leftrightarrow\ \delta_D(\hat{q}_D, \alpha) \cap F_N \neq \emptyset$ – äskeinen aputulos
  $\Leftrightarrow\ \delta_N(\hat{q}_N, \alpha) \cap F_N \neq \emptyset$ – δ_N:n määritelmä
  $\Leftrightarrow\ \exists\ q_N \in F_N$: $\hat{q}_N \text{—}\alpha\text{→}\ q_N$ – $\mathcal{L}(N)$:n määritelmä
  $\Leftrightarrow\ \alpha \in \mathcal{L}(N)$

$\Rightarrow$ $\mathcal{L}(D) = \mathcal{L}(N)$, ja saamme lauseen:

> *Jokaiselle epädeterministiselle äärelliselle automaatille N on olemassa deterministinen äärellinen automaatti D siten, että $\mathcal{L}(D) = \mathcal{L}(N)$.*

Yhdistämällä edellinen lause aikaisempiin saadaan:

> *Olkoon $L \subseteq \Sigma^*$.*
> *On olemassa säännöllinen lauseke r siten, että $L = \mathcal{L}(r)$*
> $\Leftrightarrow$ *on olemassa NFA N siten, että $L = \mathcal{L}(N)$*
> $\Leftrightarrow$ *on olemassa DFA D siten, että $L = \mathcal{L}(D)$.*

- toisin sanoen, säännölliset lausekkeet, NFA:t ja DFA:t määrittelevät / hyväksyvät täysin samat kielet
- epädeterministisyys ei siis laajenna “äärellisesti” hyväksyttävien kielten joukkoa; onko siitä mitään etua deterministisyyteen nähden?

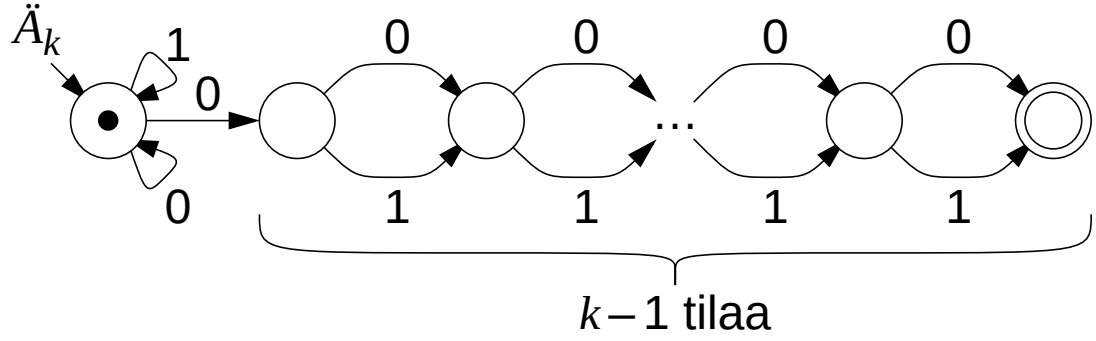
Deterministisen äärellisen automaatin koko

- tarkastellaan muunnosta $D = Det(N)$, missä
  - $N = (Q_N, \Sigma, \Delta_N, \hat{q}_N, F_N)$
  - $D = (Q_D, \Sigma, \delta_D, \hat{q}_D, F_D)$
  - $\mathcal{L}(D) = \mathcal{L}(N)$
- D:n tila on N:n tilojen joukko

$\Rightarrow$ $|Q_D|$:lle saadaan yläraja $2^{|Q_N|}$
  - aika suuri!

- voidaanko löytää muunnos, joka ei paisuta automaatin kokoa eksponentiaalisesti?

- kuvan NFA $\ddot{A}_k$ ($k \geq 2$)
  - hyväksyy kielen (0 | 1)*0(0 | 1)$^{k-2}$, ts. ne 0-1-jonot, joiden ($k-1$):s merkki lopusta on 0
  - sisältää k tilaa



- jos D on DFA joka hyväksyy saman kielen, niin D:ltä voi kysyä oliko i:s ($1 \leq i \leq k-1$) viimeksi luettu merkki 0 syöttämällä $k-i-1$ merkkiä ja katsomalla, päätyykö D lopputilaan

$\Rightarrow$ D:n täytyy “muistaa” ainakin $k-1$ bittiä

$\Rightarrow$ D:llä täytyy olla ainakin 2^{k-1} tilaa

$\Rightarrow$

Jokaiselle $k \in \boldsymbol{Z}^+$ on olemassa k-tilainen NFA siten, että jokaisessa saman kielen tunnistavassa DFA:ssa on ainakin 2^{k-1} tilaa.

- miksi jätimme lauseessa pois vaatimuksen $k \geq 2$?
- siis vaikka deterministiset automaatit hyväksyvät samat kielet kuin epädeterministiset, epädeterministiset voivat olla aika lailla pienempiä

NFA:iden hyväksymien kielten vertailu

- olkoot N_1 ja N_2 NFA:ita
- päätöstehtävän
  NFA-SAMUUS: “Onko $\mathcal{L}(N_1) = \mathcal{L}(N_2)$?”
  voi ratkaista ratkaisemalla
  “Onko $\mathcal{L}(Det(N_1)) = \mathcal{L}(Det(N_2))$?”
  – vertailu sujuu polynomiaalisessa ajassa
  – tiedämme varmasti, että deterministisointi vaatii joskus eksponentiaalisesti aikaa

$\Rightarrow$ hitaimmillaan eksponentiaalinen algoritmi

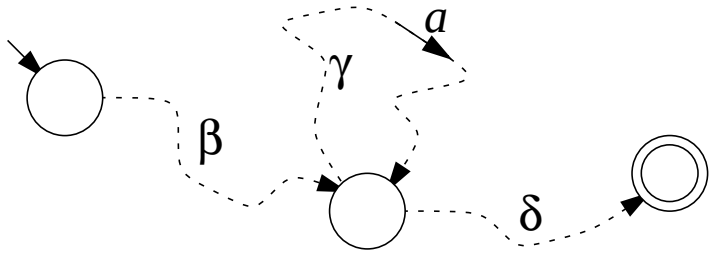
K: tuottaisiko jokin toinen lähestymistapa polynomiaalisen algoritmin?

V: kukaan ei tiedä täysin varmasti, mutta kurssilla OHJ-2300 Johdatus tietojenkäsittelyteoriaan paljastuu, että melkein varmasti “ei”

Pumppauslemma

- olkoon L säännöllinen kieli ja $N = (Q, \Sigma, \Delta, \hat{q}, F)$ sen hyväksyvä NFA
- jos $\alpha \in L$ ja $|\alpha| \geq |Q|$, niin viimeistään $|Q|$ ensimmäistä α:n merkkiä luettuaan N tulee uudelleen johonkin tilaan q, jossa se on käynyt aiemmin, ja jossa käytyään se on lukenut ≥ 1 merkin

- jaetaan α osiin $\beta\gamma\delta$ siten, että
  - N lukee β:n kulkiessaan $\hat{q}$:stä 1. kerran q:hun
  - N lukee γ:n kulkiessaan q:sta toistamiseen q:hun
  - N lukee δ:n kulkiessaan q:sta johonkin lopputilaan



- toistamalla q:sta q:hun kulkevan kierroksen nolla tai useampia kertoja N pystyy lukemaan ja hyväksymään kaikki merkkijonot muotoa $\beta\gamma^*\delta$, ts. jonot $\beta\delta$, $\beta\gamma\delta$, $\beta\gamma\gamma\delta$, $\beta\gamma\gamma\gamma\delta$, …

$\Rightarrow$ saamme nimellä *pumppauslemma* (*pumping lemma*) tunnetun tuloksen

> *Olkoon L säännöllinen kieli. On olemassa $k \in \boldsymbol{N}$ siten, että jos $\alpha \in L$ ja $|\alpha| \geq k$, niin on olemassa merkkijonot β, γ ja δ siten, että $\alpha = \beta\gamma\delta$, $\gamma \neq \varepsilon$, $|\beta\gamma| \leq k$, ja jokainen merkkijono muotoa $\beta\gamma^*\delta$ kuuluu L:ään.*

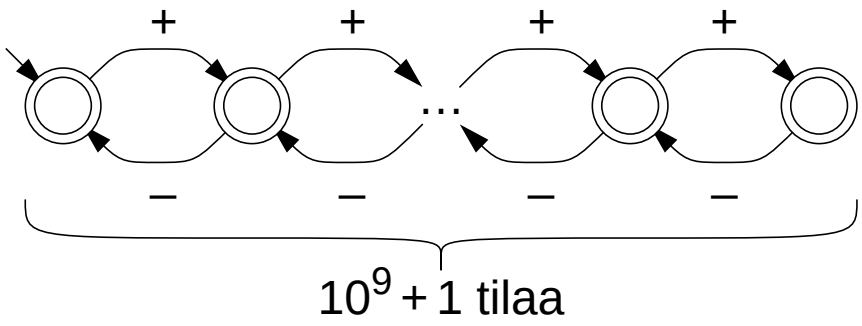
Pumppauslemman avulla voi todistaa, että jotkin kielet eivät ole säännöllisiä

- esimerkki: jos NFA N hyväksyy kaikki jonot muotoa $a^n b^n$ missä $n \in \boldsymbol{N}$, niin
  - valitsemalla $n = k$ saadaan $\beta = a^i$, $\gamma = a^j$ ja $\delta = a^{n-i-j}b^n$, missä $j > 0$ ja $i + j \leq k = n$

  $\Rightarrow$ $a^{n-j}b^n \in \mathcal{L}(N)$, vaikka se ei ole muotoa $a^n b^n$

  $\Rightarrow$ $\{ a^n b^n \mid n \in \boldsymbol{N} \}$ ei ole säännöllinen kieli

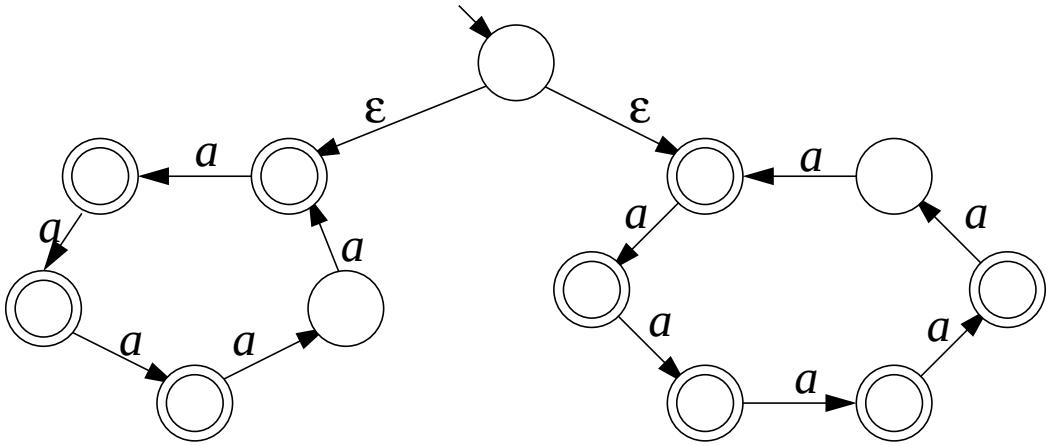
- merkintöjä lisäesimerkkejä varten
  - $\beta \leq \alpha$, jos merkkijono β on merkkijonon α alkuosa
  - $\#_\alpha(a)$ = merkin a esiintymiskertojen määrä merkkijonossa α
- lisää epäsäännöllisiä kieliä (“+”, “–” $\in \Sigma$):
  - $\{ \alpha \in \Sigma^* \mid \#_\alpha(\text{“–”}) = \#_\alpha(\text{“+”}) \}$
  - $\{ \alpha \in \Sigma^* \mid \#_\alpha(\text{“–”}) \leq \#_\alpha(\text{“+”}) \}$
  - $\{ \alpha \in \Sigma^* \mid \forall\, \beta\, ;\, \beta \leq \alpha : \#_\beta(\text{“–”}) \leq \#_\beta(\text{“+”}) \}$
  - $\{ \alpha \in \Sigma^* \mid \#_\alpha(\text{“–”}) \leq \#_\alpha(\text{“+”}) \leq \#_\alpha(\text{“–”}) + 10^9 \}$
- itse asiassa hyvin harvat kielet ovat säännöllisiä!
- kuitenkin esimerkiksi seuraava kieli on säännöllinen:
  $\{ \alpha \in \Sigma^* \mid \forall\, \beta\, ;\, \beta \leq \alpha : \#_\beta(\text{“–”}) \leq \#_\beta(\text{“+”}) \leq \#_\beta(\text{“–”}) + 10^9 \}$



Varoittava esimerkki

- tarkastellaan kieltä $L_{30} = \{ a^n \mid n \bmod 30 \neq 29 \}$
- voisi luulla, että koska lyhin hylätty merkkijono on 29 merkkiä pitkä, pienin L_{30}:n hyväksyvä NFA sisältää vähintään 29 tai 30 tilaa
- jos tarkkaan katsoo, huomaa kuitenkin, että pumppauslemma ei sovellu tilanteeseen
  - a^{29} on **hylätty** eikä hyväksytty merkkijono

- eikä syyttä: seuraava 12-tilainen NFA hyväksyy L_{30}:n!



⇒ intuitio “sen on osattava laskea ainakin 29:ään ⇒ tarvitaan ≥ 29 tilaa” ei ole pätevä

- sen sijaan **deterministisessä** tapauksessa a^n vie yksikäsitteiseen tilaan $q_n = \delta(q_i, a^n)$ kullakin $n \in \boldsymbol{N}$
  - q_{29} ei ole lopputila, mutta $q_0, q_1, \ldots, q_{28}$ ovat

  ⇒ voidaan päätellä $q_i \neq q_{29}$, kun $0 \leq i < 29$

  ⇒ jos $0 \leq i < j \leq 29$, niin
  $\delta(q_i, a^{29-j}) = q_{29+i-j} \neq q_{29} = \delta(q_j, a^{29-j})$

  ⇒ $q_i \neq q_j$

  ⇒ tiloja on ainakin 30

Onko DFA hyvä tietokoneen malli?

- voidaan väittää, että oikean tietokoneen muistin koolla on aina yläraja
  - lisämuistin ostoon varatut rahat loppuvat joskus
  - osoitteen sanapituus loppuu joskus
  - maapallolta loppu pii joskus
  - …

⇒ oikea tietokone on enintään DFA
  - “enintään”, koska oikea tietokone voi olla *vihamielisesti epädeterministinen*: epädeterminismi vähentää onnistumisia
  - (NFA on *ystävällisesti* epädeterministinen)

⇒ oikealla tietokoneella ei voi ratkaista tehtävää: onko annetussa merkkijonossa sama määrä “+”- ja “–”-merkkejä!

- kuitenkin seuraava ohjelma on helppo keksiä:

```
i := 0
while ¬ end-of-input do
    read( chr )
    if chr = “+” then i := i + 1
    elsif chr = “–” then i := i – 1
    endif
endwhile
if i = 0 then print “kyllä” else print “ei” endif
```

- selitys: tämä ohjelma aiheuttaa muistin ylivuodon esimerkiksi jos syötemerkkijonossa on hirmuisen paljon enemmän “+”- kuin “–”-merkkejä
  - voi tapahtua vain hirmuisen pitkillä syötteillä

⇒ oletuksella “tietokone on DFA” saadaan tuloksia, jotka periaatteessa pitävät paikkansa, mutta ovat käytännön kannalta irrelevantteja

- tulemme kurssilla OHJ-2300 huomaamaan, että oletuksella “tietokoneen muisti on rajaton” saadaan käytännön kannalta mielekkäämpiä tuloksia!

6 BNF JA JÄSENTÄMINEN

Tässä luvussa käsitellään erästä äärellisiä automaatteja voimakkaampaa tapaa määritellä kieliä

Tapa voidaan esittää neljässä eri muodossa

- BNF
- ratapihakaaviot
- yhteysriippumattomat kieliopit
- (yhden pinon koneet)

Muodot ovat

- ilmaisuvoimaltaan yhtäpitävät
- sijaitsevat eri kohdissa akselilla teoria $\leftrightarrow$ käytäntö

Luvussa myös

- käsitellään näin määritellyillä kielillä kirjoitettujen syötteiden jäsentämistä
- täydennetään lausekkeiden teoriaa

6.1 Backus-Naur form

Backus-Naur form (*BNF*) muunnelmineen on laajalti käytetty ohjelmointi- ja muiden formaalien kielten syntaksien määrittelyssä

- voimakkaampi kuin säännölliset kielet
- osittain käsiteltävissä tehokkaasti tietokoneella
- havainnollinen
  - varsinkin graafiset muodot

$\Rightarrow$ laajalti käytetty

- yleensä ihmisten luettavaksi tarkoitettu
- standardoitu versio: ISO/IEC 14977:1996(E)
- osaavin kirjoittajakunta (matemaatikot ja teoreettiset tietojenkäsittelyoppineet) suosii yksinkertaisia mutta käytännön kannalta kömpelöitä versioita
  - käsillä olevassa tilanteessa vähämerkitykselliset yksityiskohdat hoidettu huonosti tai ohitettu, vaikka kenties käytännössä hyvin tärkeitä

$\Rightarrow$ käytännön ohjelmointi- yms. kielten suunnittelijat laatineet omia murteita
  - jotkin hyviä
  - toiset epätäsmällisiä, jopa sisäisesti ristiriitaisia

- vrt. HTML:n tai eri ohjelmointikielten tilanne

$\Rightarrow$ BNF:n lukijan kannattaa aina tarkastaa miten kirjoittaja käyttää merkintöjä

- tällä kurssilla käytämme tiettyä tapaa
  - pyrimme ensisijaisesti täsmällisyyteen ja sitten käyttömukavuuteen
  - jälkeenpäin kukin saa käyttää mitä versiota haluaa, kunhan valitsee sen ja käyttää sitä hyvin

Kielen ja metakielen aakkosto

- kun määritellään kieltä formaalisti, läsnä on **kaksi** kieltä:
  - *kohdekieli* (se, jota määritellään)
  - *metakieli* (se, jolla määritellään)
- jotta määritelmä olisi täsmällinen ja yksikäsitteinen, on oltava selvillä, mitkä symbolit kuuluvat kohde- ja mitkä (ainoastaan) metakieleen
- matemaattisissa teksteissä tämä ratkaistaan usein ottamalla metakieleen ylimääräisiä symboleita, jotka eivät kuulu kohdekielen aakkostoon
  - mekin oletimme “Ø”, “ε”, “·”, “|”, “*”, “(”, “)” $\notin \Sigma$, kun määrittelimme säännölliset lausekkeet
  - käytämme tätä keinoa silloin kun meille sopii
- käytännössä usein halutaan, että kohdekielessä voi käyttää kaikkia tarjolla olevia merkkejä
  $\Rightarrow$ tarvitsemme myös toisen keinon

$\Rightarrow$ käytämme tarvittaessa tuttua keinoa: kohdekielen symbolit tai niiden jonot lainausmerkkien väliin “tällä tavalla”

- ongelma: miten esitetään kohdekielen loppulainausmerkki?
  - esim.

    ```
    erikoismerkkejä ovat “%”, “&”, “””
    ```
- otetaan monista ohjelmointikielistä tuttu ratkaisu: “pako”merkki (*escape*)
  - metakielen \” edustaa kohdekielen loppulainausmerkkiä
  - metakielen \\ edustaa kohdekielen merkkiä \
  - \:n mikä tahansa muu esiintymä on virhe metakieltä vastaan

- tämän formalisoimiseksi määrittelemme jos ”, \ ∈ Σ:

  $\Sigma^{\backslash} := (\Sigma - \{\text{”}, \backslash\}) \cup \{\backslash\text{”}, \backslash\backslash\}$

  ja *decode*: $\Sigma^{\backslash} \to \Sigma$:

  decode(\”) = ” ja *decode*(\\) = \

  decode(x) = x, kun $x \notin \{\backslash\text{”}, \backslash\backslash\}$

  – tapaus ” ∈ Σ ja \ ∉ Σ samaan tapaan, \ ei koodata
- *decode* laajennetaan merkkijonoille seuraavasti:

  $decode(a_1 \ldots a_n) = decode(a_1) \ldots decode(a_n)$

  – siis jokainen merkki korvataan erikseen
- koska \\ ja \” tulkitaan pareiksi, esimerkiksi \\\” ositetaan \\ \” eikä \ \\ ”
  – esim. *decode*(`backslash = \"\\\"`) = `backslash = "\"`

Kurssilla käytettävä laajennettu BNF BNF^{++}

- määritellään säännöllisten lausekkeiden yleistyksenä ottamalla käyttöön pari uutta operaattoria, sekä joukko *välisymboleita*
  – edustavat kokonaisia kieliä
  – saa käyttää rekursiivisesti
- BNF^{++}-määritelmä koostuu seuraavista osista:
  – äärellinen joukko Σ *loppusymboleita* (*terminal symbols*)
  – äärellinen joukko Ψ *välisymboleita* (*nonterminal symbols*) siten, että niiden nimet noudattavat myöhemmin annettavat ehdot
  – *alkusymboli* (*start symbol*) $\mathcal{S}$, $\mathcal{S} \in \Psi$
  – äärellinen joukko Π *sääntöjä* (*rules, productions*); säännöt on määritelty alla
- loppusymbolien joukko Σ on itse asiassa BNF^{++}-määritelmällä määriteltävän kohdekielen aakkosto

- sääntö on mikä tahansa kaava muotoa

  A ::= *lauseke*

  missä $A \in \Psi$, ja *lauseke* on mikä tahansa seuraavien vaihtoehtojen mukaan muodostettu merkkijono:
  - ε
  - "$a_1 a_2 \ldots a_n$", missä $a_1, a_2, \ldots, a_n \in \Sigma^{\setminus}$
  - B, missä $B \in \Psi$
  - *lauseke lauseke … lauseke*
  - *lauseke* | *lauseke* | … | *lauseke*
  - *lauseke**
  - *lauseke*$^+$
  - [*lauseke*]
  - (*lauseke*)
- jos Σ:n alkio *a* ei kuulu joukkoon Ψ, ja se noudattaa myöhemmin annettavat ehdot, niin vaihtoehdon "$a_1 a_2 \ldots a_n$" saa korvata vaihtoehdolla *a*
  - loppusymboleita saa silti pantua peräkkäin, koska *lauseke lauseke … lauseke* on yksi vaihtoehto
- sitovuussäännöt
  - * ja $^+$ sitovat voimakkaimmin, sitten peräkkäisyys ja lopuksi |
  - ε, "$a_1 a_2 \ldots a_n$" tai *a*, välisymboli, [*lauseke*] ja (*lauseke*) tulkitaan kukin sitovuuden kannalta yhdeksi kokonaisuudeksi
  - esim. A "*bb*" | "*abc*""*cba*"$^+$ tarkoittaa samaa kuin

    (A "*bb*") | ("*abc*" (("*cba*")$^+$))
- Π:ltä vaaditaan, että jokainen välisymboli esiintyy "::="-merkin vasemmalla puolen täsmälleen kerran

Toisto $n \ldots m$ kertaa

- joissakin notaatioissa voi määritellä toiston $n \ldots m$ kertaa tyyliin $lauseke^{n \ldots m}$, missä n ja m ovat luonnollisia lukuja
- emme määrittele sitä BNF^{++}:aan, koska
  - se ei ole kurssin asioiden kannalta tarpeen
  - se on helppo ymmärtää jos ymmärtää BNF^{++}:n
  - se vaikeuttaa ratkaisevasti osaa jatkossa tehtävistä päätelmistä tuomalla mukaan luvut
  - jokainen sitä hyödyntävä lauseke on toteutettavissa peräkkäisyyden ja “|” avulla (vaikka kömpelösti)
- tämä ei tarkoita väitettä, että toisto $n \ldots m$ kertaa olisi hyödytön operaatio!
  - päinvastoin, se aiheuttaisi jatkossa vaikeuksia juuri siksi, että sillä voi tehdä kätevästi asioita, jotka ilman sitä voi tehdä vain kömpelösti

$\Rightarrow$ tämän kurssin kannalta on parempi olla ilman sitä, mutta muualla tilanne voi olla toinen

Esimerkki: yksinkertaisen lausekelaskimen syöttökieli

- loppusymbolit
  Σ = { “+”, “–”, “*”, “/”, “(”, “)”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9” }
  - ehkä sama kuin laskimen näppäinten joukko ilman “=”-näppäintä, jota vastaa syötteen loppu?
- välisymbolit
  Ψ = { *lasku*, *termi*, *tekijä*, *atomilaus.*, *luku*, *numero* }
- alkusymboli
  $\mathcal{S}$ = *lasku*

- säännöt Π

  lasku ::= *termi* ((“+” | “–”) *termi*)*
  termi ::= *tekijä* ((“*” | “/”) *tekijä*)*
  tekijä ::= [“+” | “–”] *atomilaus.*
  atomilaus. ::= *luku* | “(” *lasku* “)”
  luku ::= *numero*$^{+}$
  numero ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

- esimerkiksi seuraavat ovat laskuja (myöhemmin näemme miksi)
  - 1, (1), ja ((1))
  - +000114
  - –3++000114
  - ((1))+25*(–3++000114)
- esimerkiksi seuraavat eivät ole laskuja
  - ε oltava $\geq$ 1 numero
  - ++114 enintään yksi etumerkki
  - 1+)3(sulut väärinpäin
  - 3+*4 etumerkki väärässä kohti
  - (1+2 sulut eivät tasapainossa
  - 1 + 2 aakkosto ei sisällä välilyöntiä

Merkkijonojen johtaminen

- BNF^{++}-määritelmien merkityksen määrittelemiseksi tarvitsemme merkkijonojen *tuottamisen* eli *johtamisen* (*derive*) käsitettä
- merkitsemme “lauseke *xxx* tuottaa merkkijonon α” lyhenteellä

  $xxx \Rightarrow^{*} \alpha$

  - sama lauseke voi tuottaa useita eri merkkijonoja

BNF^{++}-lauseke tuottaa ne ja vain ne merkkijonot, jotka voidaan johtaa seuraavien sääntöjen mukaan:

- $\varepsilon \Rightarrow^* \varepsilon$
  - siis lauseke ε tuottaa tyhjän merkkijonon
- $a \Rightarrow^* a$, jos $a \in \Sigma$ (lainausmerkitön tapaus), ja "$a_1 a_2 \ldots a_n$" $\Rightarrow^*$ $decode(a_1 a_2 \ldots a_n)$
  - siis "$a_1 a_2 \ldots a_n$" tuottaa $a_1 a_2 \ldots a_n$:n siten muutettuna, että pakomerkit on tulkittu
- jos $B \in \Psi$ on määritelty säännöllä B ::= *lauseke′*, ja *lauseke′* $\Rightarrow^* \alpha$, niin
  $B \Rightarrow^* \alpha$
  - siis lausekkeessa esiintyessään B edustaa kaikkia niitä merkkijonoja, jotka sen määrittelyssä käytetty lauseke voi tuottaa
  - B:n esiintymä on kuin aliohjelmakutsu
  - esim. *numero* $\Rightarrow^*$ 1 $\Rightarrow$ *luku* $\Rightarrow^*$ 1 $\Rightarrow$ *atomilaus.* $\Rightarrow^*$ 1 $\Rightarrow$ *tekijä* $\Rightarrow^*$ 1 $\Rightarrow$ *termi* $\Rightarrow^*$ 1 $\Rightarrow$ *lasku* $\Rightarrow^*$ 1
- jos *lauseke*$_1 \Rightarrow^* \alpha_1$, *lauseke*$_2 \Rightarrow^* \alpha_2$, … ja *lauseke*$_n \Rightarrow^* \alpha_n$, niin
  lauseke$_1$ *lauseke*$_2$ … *lauseke*$_n$ $\Rightarrow^* \alpha_1 \alpha_2 \ldots \alpha_n$
  - osalausekkeiden tuottamat merkkijonot pannaan peräkkäin
  - esim. koska *lasku* $\Rightarrow^*$ 1, niin "(" *lasku* ")" $\Rightarrow^*$ (1)
- jos *lauseke*$_i \Rightarrow^* \alpha$ jollakin $1 \le i \le n$, niin
  lauseke$_1$ | *lauseke*$_2$ | … | *lauseke*$_n$ $\Rightarrow^* \alpha$
  - valitaan jonkin osalausekkeen tuottama merkkijono
  - esim. "0" | "1" | "2" | "3" $\Rightarrow^*$ 2

- jos *lauseke* ⇨* α_1, *lauseke* ⇨* α_2, … ja *lauseke* ⇨* α_n missä $n \geq 0$, niin
  *lauseke** ⇨* $\alpha_1\alpha_2\ldots\alpha_n$
  - pannaan peräkkäin nolla tai useampia merkkijonoja, jotka osalauseke voi tuottaa
  - $\alpha_1, \alpha_2, \ldots, \alpha_n$ saavat olla samoja tai eri merkkijonoja
  - esim. (“0” | “1”)* ⇨* ε ja (“0” | “1”)* ⇨* 1011
- jos *lauseke* ⇨* α_1, *lauseke* ⇨* α_2, … ja *lauseke* ⇨* α_n missä $n \geq 1$, niin
  lauseke$^+$ ⇨* $\alpha_1\alpha_2\ldots\alpha_n$
  - pannaan peräkkäin yksi tai useampia merkkijonoja, jotka osalauseke voi tuottaa
  - esim. *numero*$^+$ ⇨* 1917, mutta *numero*$^+$ ⇨̸* ε
- jos *lauseke* ⇨* α, niin
  [*lauseke*] ⇨* ε ja [*lauseke*] ⇨* α
  - hakasulut ilmaisevat vapaaehtoisuutta
  - esim. [“+” | “–”] “1” ⇨* 1 ja [“+” | “–”] “1” ⇨* –1
- jos *lauseke* ⇨* α, niin
  (*lauseke*) ⇨* α
  - sulut eivät vaikuta tuotettuun merkkijonoon
  - tarpeen lausekkeiden ryhmittelemiseksi
  - esim. (“0”“1”)* ⇨* 0101, mutta “0”“1”* ⇨̸* 0101

Nyt voidaan määritellä

- $\mathcal{L}(lauseke) := \{ \alpha \in \Sigma^* \mid lauseke \Rightarrow^* \alpha \}$
- jos $A \in \Psi$ on määritelty säännöllä A ::= *lauseke*, niin
  $\mathcal{L}(A) = \mathcal{L}(lauseke)$
- jos $\mathcal{B} = (\Psi, \Sigma, \Pi, \mathcal{S})$ on BNF^{++}-määritelmä, niin
  $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{S})$
  - ts. alkusymbolin määrittelemä kieli

Lausekelaskinesimerkissä

- koska *lasku* ⇨* 1, niin *atomilaus.* ⇨* (1) ⇒ …
  ⇒ *lasku* ⇨* (1) ⇒ … ⇒ *lasku* ⇨* ((1)) ⇒ …
- koska *numero* ⇨* 0, *numero* ⇨* 1 ja *numero* ⇨* 4, niin *numero*$^{+}$ ⇨* 000114
  ⇒ *luku* ⇨* 000114 ⇒ *atomilaus.* ⇨* 000114
  ⇒ *tekijä* ⇨* +000114 ⇒ … ⇒ *lasku* ⇨* +000114
- jne.

Rekursiivisten määritelmien tulkinnasta

- yllä BNF^{++}-määritelmille määritelty merkitys lienee suurimmalta osin helppo ymmärtää
- rekursiivisten määritelmien merkitys voi kaivata lisäkommentteja
  - esim. A ::= ε | A “a”
  - esim. B ::= B | “a”
  - esim. C ::= C
  - esim. D ::= D “a” D
- asia ratkeaa, kun luetaan huolella “… ne **ja vain ne** merkkijonot, jotka **voidaan johtaa** …”
- esimerkki: A ::= ε | A “a”
  - ε | A “a” ⇨* ε ⇒ A ⇨* ε
  ⇒ A “a” ⇨* εa = a ⇒ ε | A “a” ⇨* a ⇒ A ⇨* a
  ⇒ ε | A “a” ⇨* aa ⇒ A ⇨* aa
  ⇒ A ⇨* aaa ⇒ A ⇨* $aaaa$ ⇒ …
  ⇒ $\mathcal{L}(A)$ = {ε, a, aa, aaa, …}

- esimerkki: B ::= B | “a”
  - selvästi $a \in \mathcal{L}(B)$
  - kun sijoitetaan a B:n paikalle oikealla puolen, saadaan a

  $\Rightarrow$ ei voida johtaa muita merkkijonoja kuin a

  $\Rightarrow$ $\mathcal{L}(B) = \{a\}$
- esimerkki: C ::= C
  - ei voida johtaa mitään

  $\Rightarrow$ $\mathcal{L}(C) = \emptyset$
- samoin $\mathcal{L}(D) = \emptyset$

Käytännöllisiä sopimuksia

- tavallisesti Σ ja Ψ käyvät ilmi säännöistä

  $\Rightarrow$ turha luetella erikseen

  $\Rightarrow$ sallimme niiden luettelematta jättämisen, jolloin

  Ψ = sääntöjen vasempien puolten joukko

  Σ = säännöistä löytyvien loppusymbolien joukko (tunnistetaan lainausmerkeistä tai siitä, että eivät Ψ:ssä)
- tavallisesti $\mathcal{S}$ on selvä asiayhteydestä

  $\Rightarrow$ sallimme jättää pois
  - jollei muuta sanota, niin

    $\mathcal{S}$ = ensimmäisen säännön vasen puoli
- nyt laskun määritelmäksi riittää

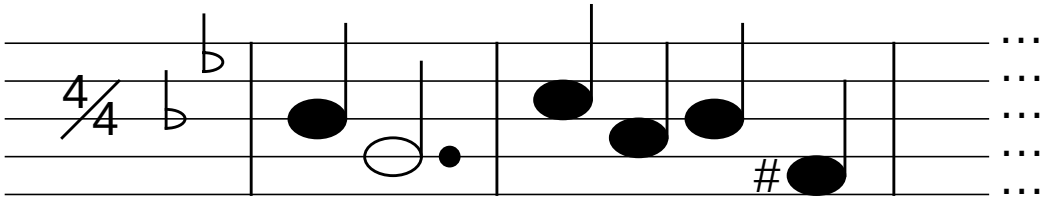
lasku ::= *termi* ((“+” | “–”) *termi*)*
termi ::= *tekijä* ((“*” | “/”) *tekijä*)*
tekijä ::= [“+” | “–”] *atomilaus.*
atomilaus. ::= *luku* | “(” *lasku* “)”
luku ::= *numero*$^+$
numero ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

Kuinka “alas” määritelmä viedään?

- toisin sanoen, mitä oletetaan kohdekielen aakkostosta?
- lausekelaskimen tapauksessa oletimme sen olevan konkreettinen merkistö
  $\Rightarrow$ 1 + 2 ei ole lasku koska siinä on välilyöntejä
- usein on tarkoituksenmukaista jättää määritelmä vähän korkeammalle tasolle
  – tällöin usein $\Sigma \cap \Psi = \emptyset$ ja Σ:n alkiot erottuvat selvästi metakielen merkeistä ε, |, *, $^+$, [,], (,)
  $\Rightarrow$ lainausmerkkien käyttö loppusymbolien osoittamiseksi ei tarpeen
- määritelmä voi olla tarkoitettukin abstraktioksi, tai muuten kuin merkeillä toteutettavaksi

Esimerkki: nuottikirjoitus

sävelmä ::= [*tahtilaji*] *sävellaji tahti*$^{+}$
tahtilaji ::= *luonn.luku luonn.luku*
sävellaji ::= (*alennus korkeus*)*
| (*ylennys korkeus*)*
korkeus ::= *C* | *D* | *E* | *F* | *G* | *A* | *H*
tahti ::= (*nuotti* | *tauko*)$^{+}$
nuotti ::= [*siirto*] *oktaavi korkeus*
kesto [*pidennys*]
oktaavi ::= *luonn.luku*
siirto ::= *alennus* | *ylennys* | *palautus*
kesto ::= *koko* | *puoli* | *osa4* | *osa8* | *osa16*
| *osa32*
tauko ::= *kesto* [*pidennys*]



- voimme päätellä, että
  - Σ = { *luonn.luku*, *C*, *D*, *E*, *F*, *G*, *A*, *H*, *alennus*, *ylennys*, *palautus*, *koko*, *puoli*, *osa4*, *osa8*, *osa16*, *osa32*, *pidennys* }
  - Ψ = { *sävelmä*, *tahtilaji*, *sävellaji*, *korkeus*, *tahti*, *nuotti*, *oktaavi*, *siirto*, *kesto*, *tauko* }
  - $\mathcal{S}$ = *sävelmä*

Tarvittaessa voimme tietysti viedä määritelmän esimerkiksi ISO 8859-1 merkkien tasolle

- esimerkki: tekstuaalinen nuottikirjoituksen kuvauskieli

sävelmä ::= [*tahtilaji*] *sävellaji* “|” (*tahti* “|”)$^+$
tahtilaji ::= *luonn.luku* “/” *luonn.luku*
sävellaji ::= (*alennus korkeus*)*
| (*ylennys korkeus*)*
alennus ::= “b”
ylennys ::= “#”
palautus ::= “!”
korkeus ::= “C” | “D” | “E” | “F” | “G” | “A” | “H”
tahti ::= (*nuotti* | *tauko*)$^+$
nuotti ::= “d” [*siirto*] *oktaavi korkeus kesto* [“.”]
oktaavi ::= *luonn.luku*
siirto ::= *alennus* | *ylennys* | *palautus*
kesto ::= “1” | “2” | “4” | “8” | “16” | “32”
tauko ::= “=” *kesto* [“.”]
luonn.luku ::= (“0” | “1” | “2” | “3” | “4” | “5” | “6” |
“7” | “8” | “9”)$^+$

- esimerkissä tuli taas epäsuorasti kiellettyä välilyöntien käyttö
- välilyöntien lisääminen sääntöihin on usein hyvin kömpelöä

⇒ seuraava asia

Leksikaaliset ja syntaktiset säännöt

- ohjelmointikielten syntaksien määrittelyssä on tapana lopettaa määritelmä yksittäisiä merkkejä ylemmällä tasolla
  - loppusymboleita ovat avainsanat kuten `switch`, tunnisteet kuten `laskuri`, literaalivakiot kuten `12.34` ja `"kukkuu!"`, väli- ja operaattorimerkit kuten +, ++, ., &&, …
  - *tekstialkiot* (*token*)
- tekstialkioiden muodostuminen merkeistä määritellään erikseen
  - esim. *tunniste* ::= *kirjain* (*kirjain* | *numero* | “_”)*
  - yleensä säännölliset lausekkeet riittävät tähän
  - ⇒ ohjelmakoodi pilkottavissa nopeasti tekstialkioihin äärellisillä automaateilla
- ohjelman muodostuminen tekstialkioista määritellään BNF:n tapaisilla keinoilla
  - esim. pieni osa C++:n kielioppia:

    for-lause ::=
    “for” “(” *for-alku* [*ehto*] “;” [*lauseke*] “)” *lause*
    for-alku ::= *lauseke-lause* | *yksinkert.julistus*
    lauseke-lause ::= [*lauseke*] “;”
- tavallisesti vaaditaan, että tekstialkioiden
  - sisällä ei saa olla ylimääräisiä merkkejä
  - välissä saa olla kommentteja ja tiettyjä merkkejä, kuten välilyönti, rivin vaihto ja tabulaattori
- sääntöjä, jotka määrittelevät, miten sisään tuleva merkkijono ryhmittyy tekstialkioiden jonoksi, kutsutaan *leksikaalisiksi* säännöiksi
- *syntaksi* määrittelee, missä järjestyksessä tekstialkioita saa panna peräkkäin

- silloin kun leksikaalitaso on määritelty, se on eri taso kuin syntaksi
  - leksikaalitaso on “nippeliteknisempi”
  - ⇒ usein se jätetään epäolennaisena määrittelemättä

Mitä leksikaalisia sääntöjä sovimme BNF^{++}:lle?

- olennaista on vain, että lukija pystyy
  - näkemään, missä kukin BNF^{++}-sääntö alkaa ja loppuu
  - erottamaan loppusymbolit, välisymbolit ja BNF^{++}:n metasymbolit (kuten $^{+}$) toisistaan

⇒ ei ole tarpeen määritellä kovin tiukkoja sääntöjä

BNF^{++}:n leksikaaliset säännöt

- BNF^{++}:n tekstialkiot ovat
  - välisymbolin nimi
  - lainausmerkein rajattu loppusymbolien jono \”- ja \\-koodauksin
  - loppusymbolin nimi, jos sitä ei rajattu lainausmerkein
  - ::=, ε, |, *, $^{+}$, [,], (,)
  - seuraavat merkkijonot (“a.s.” ~ “avainsana”):

loppus.j.a.s. ::= “loppusymbolit:” | “Loppusymbolit:” | “LOPPUSYMBOLIT:”

välis.j.a.s. ::= “välisymbolit:” | “Välisymbolit:” | “VÄLISYMBOLIT:”

alkus.a.s. ::= “alkusymboli:” | “Alkusymboli:” | “ALKUSYMBOLI:”

sääntöa.s. ::= “säännöt:” | “Säännöt:” | “SÄÄNNÖT:”

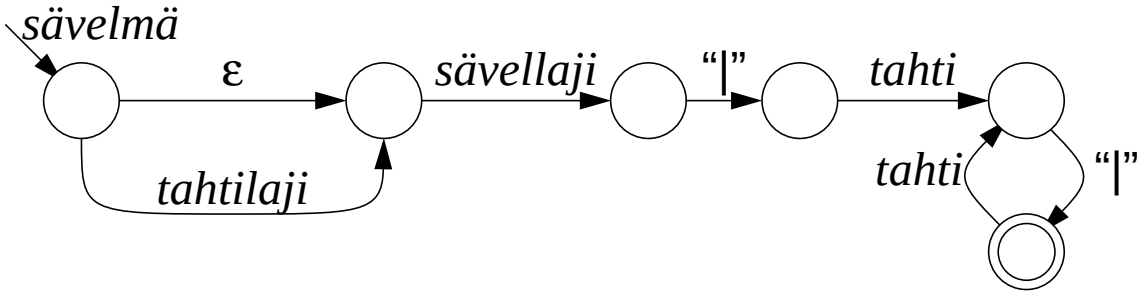
- välisymbolin nimi
  - tulee sisältää ainakin yksi merkki
  - saa sisältää kaikkia merkkejä paitsi näkymättömiä tai tyhjänä näkyviä merkkejä (esim. välilyönti), ohjausmerkkejä (esim. rivinsiirto ja äänimerkki ISO-Latin-1 7), ja seuraavia: ε, ”, |, *, $^{+}$, [,], (,), :
- sama vaaditaan loppusymboleilta, jos niiden rajaamiseen ei käytetä lainausmerkkejä
- isot ja pienet kirjaimet tulkitaan eri merkeiksi nimissä
- jos kaksi välisymbolia ja / tai lainausmerkitöntä loppusymbolia on peräkkäin, niiden välissä tulee olla tyhjää
- kahden säännön välissä tulee olla tyhjää
  - mieluiten joka sääntö aloitetaan omalta riviltä
- sääntökokoelman alku ja loppu on osoitettava esimerkiksi sisentämällä koko sääntökokoelma
- sääntökokoelma tulee asemoida siten, että normaaliin tapaan ylhäältä alas ja vasemmalta oikealle etenevä lukija kohtaa tekstialkiot tarkoitetussa järjestyksessä

BNF^{++}:n syntaksi

määritelmä ::= [*loppus.j.*] [*välis.j.*] [*alkus.*] *säännöt*
loppus.j. ::= *loppus.j.a.s.* “{” *loppusymboli*
(“,” *loppusymboli*)* “}”
välis.j. ::= *välis.j.a.s.* “{” *välisymboli*
(“,” *välisymboli*)* “}”
alkus. ::= *alkus.a.s. välisymboli*
säännöt ::= [*sääntöa.s.*] *sääntö*$^{+}$
sääntö ::= *välisymboli* “::=” *lauseke*
lauseke ::= *tulo* (“|” *tulo*)*
tulo ::= *toisto*$^{+}$
toisto ::= *atomilauseke* [“*” | “$^{+}$”]
vapaaehtoinen ::= “[” *lauseke* “]”
atomilauseke ::= “ε” | *loppusymboli* | *välisymboli* |
vapaaehtoinen | “(” *lauseke* “)”

Graafiset syntaksikaaviot

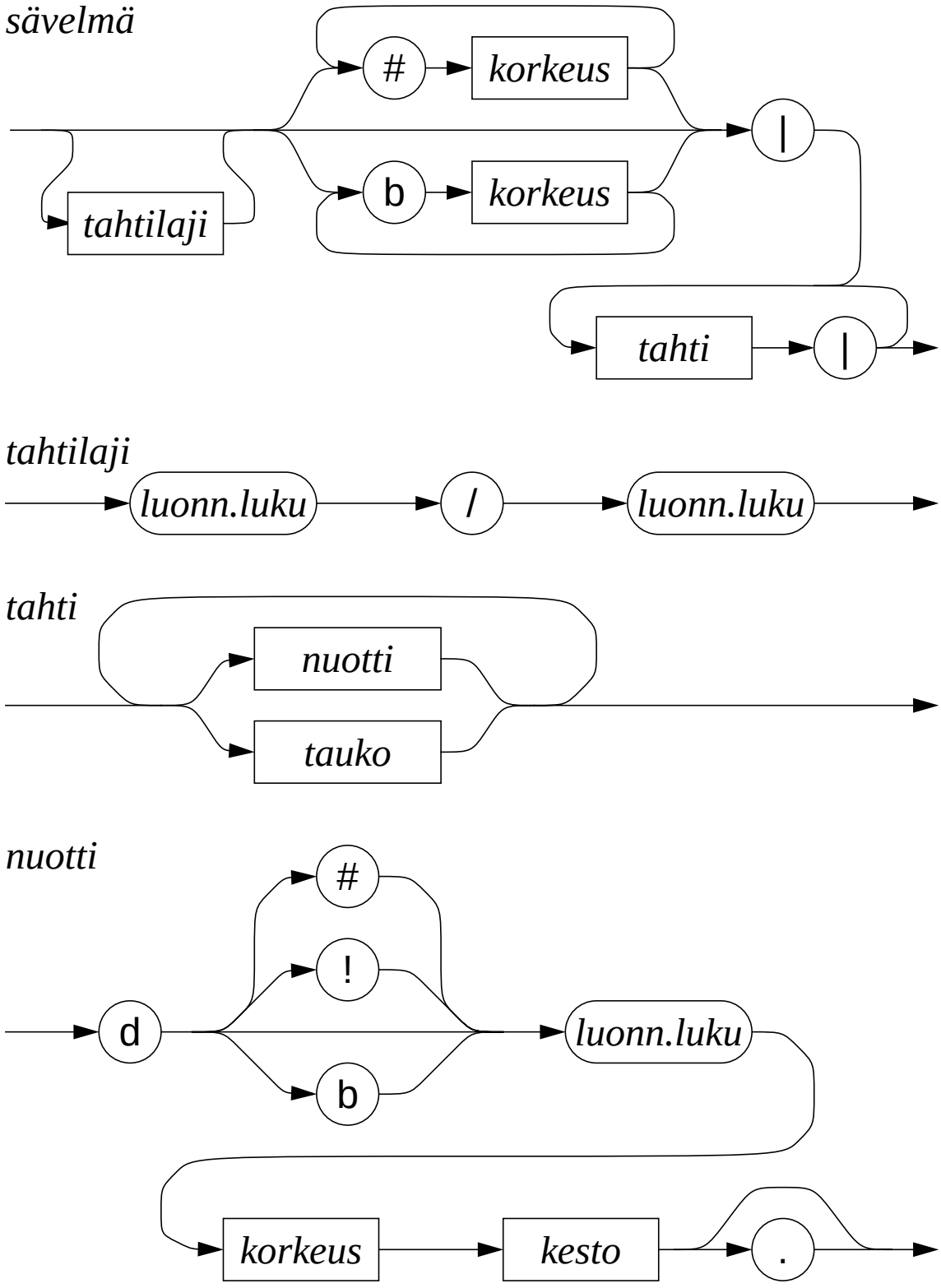
- aivan kuten säännölliset lausekkeet voi esittää NFA:ina, BNF^{++}-säännöt voisi esittää laajennettuina NFA:ina, joissa kaarten merkintöinä sallittaisiin myös välisymbolit

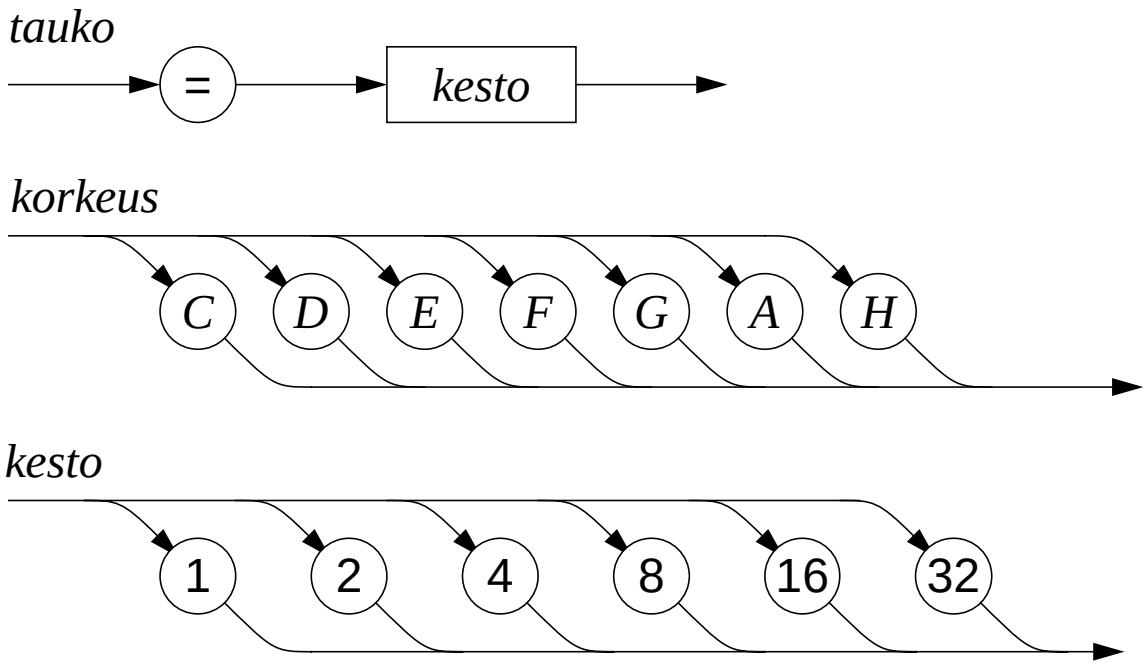


- tavaksi on kuitenkin muodostunut esittää sama informaatio toisella tavalla piirrettynä
- väli- ja loppusymbolit esitetään “laatikoina”
  – välisymbolit: laatikko on suorakaide
  – loppusymbolit: laatikon päät pyöristetty

- peräkkäisyys, vaihtoehtoisuus, vapaaehtoisuus ja toisto esitetään laatikoiden välisinä nuolina
- alkukohta esitetään tyhjästä alkavana ja loppukohta tyhjään päättyvänä nuolena
- yhteen laatikkoon tulee tasan yksi ja lähtee tasan yksi nuoli
  - nuolet yhdistetään ennen laatikkoon tuloa
  - lähtevä nuoli haarautetaan tarpeen mukaan
  - yhdistäminen ja haarautuminen tehdään “pehmeästi”, jotta nuolen suunta olisi aina selvä
- tyyliseikkoja
  - jokaisessa silmukassa oltava ≥ 1 laatikko
  - vain yksi alku- ja yksi loppunuoli
  - alkunuoli vasemmassa yläkulmassa
  - loppunuoli oikeassa reunassa
- miten tyyliseikkavaatimuksiin tulisi suhtautua?
  - kuvan merkitys usein selvä, vaikka niitä rikottaisiin $\Rightarrow$ tyylisäännöt pohjimmiltaan makuasioita
  - kannattaa noudattaa, jotta kaaviotekniikka säilyisi yhtenäisenä
  - vrt. ohjelmointikielten sisennyssäännöt
- koko määritelmä on joukko kaavioita
  - yksi kaavio esittää yhden välisymbolin määritelmän, ja esittää sen kokonaisuudessaan
  - välisymbolin nimi kaavion otsikoksi tai alkunuolen viereen
- välisymboleita tarvitaan yleensä vähemmän kuin laajennettuja BNF:iä käytettäessä
- tällaisia kaavioita kutsutaan joskus myös “ratapihakaavioiksi”

Esimerkki: nuottikirjoitus graafisilla syntaksikaavioilla olettaen, että luonnollinen luku on tekstialkio

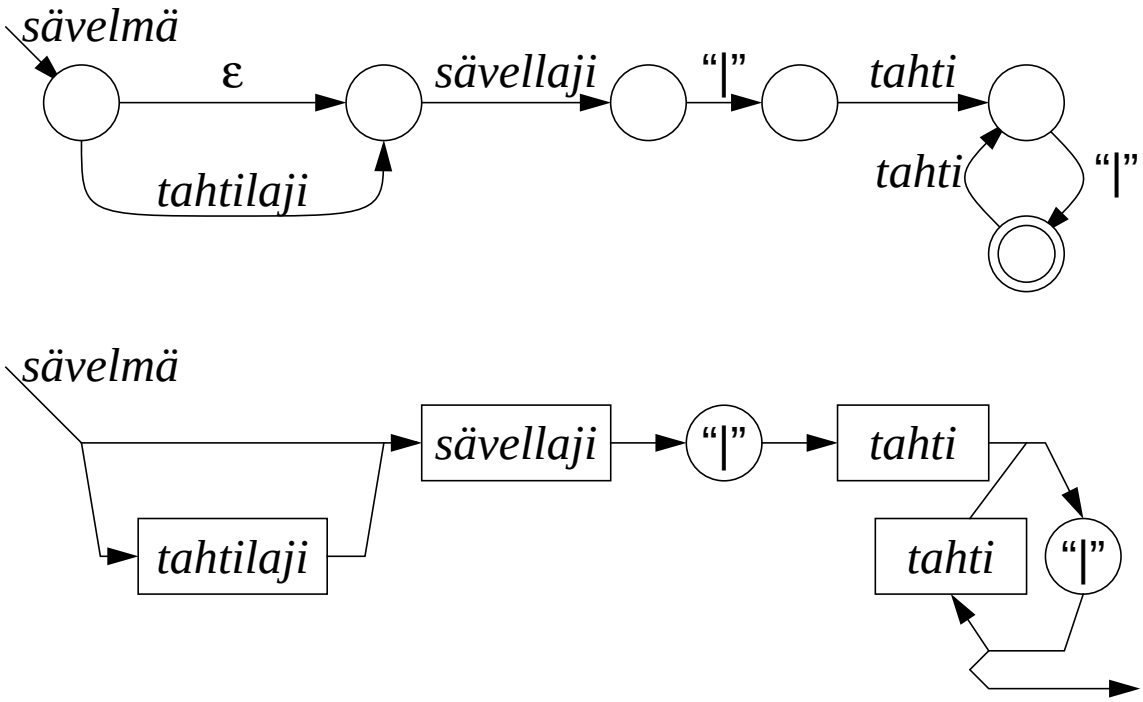




Graafisten syntaksikaavioden ilmaisuvoima BNF^{++}:aan verrattuna

- jokaisesta BNF^{++}-määritelmästä on helppo muodostaa saman kielen esittävä graafinen syntaksikaavio
  - muuntamalla se ensin NFA:ksi, jossa kaaren nimenä saa olla myös välisymboli
  - piirtämällä jokaiseen ei-ε-kaareen pyöreä- tai teräväkulmainen laatikko, ja siirtämällä kaaren nimi sinne
  - poistamalla ε-kaarten nimet
  - korvaamalla tilat nuolten yhdistämisillä ja haarautuksilla; lopputiloista yhteinen nuoli ulos

- esimerkki:



- vastaavasti graafiset syntaksikaaviot voi helposti muuntaa NFA:iksi, joissa kaaren nimenä saa olla myös välisymboli
  - perusidea: lisätään nuolten sisään tarpeellinen määrä tiloja

⇒ graafiset syntaksikaaviot määrittelevät samat kielet kuin NFA:t, joissa kaaren nimenä saa olla myös välisymboli

- NFA:n, jossa kaaren nimenä saa olla myös välisymboli, voi muuntaa BNF^{++}-määritelmäksi antamalla tiloille nimet, ja käyttämällä tilojen nimiä välisymboleina
  - tilan määrittelevät säännöt ovat muotoa
    $tila_1$ ::= $nimi_{12}\ tila_2$ | $nimi_{13}\ tila_3$ | … | $nimi_{1n}\ tila_n$
    missä $nimi_{ij}$ = kaaren ($tila_i$, $tila_j$) nimi
  - lopputilalle lisäksi … | ε

⇒ graafiset syntaksikaaviot määrittelevät samat kielet kuin BNF^{++}

Tavallinen (laajentamaton) BNF

- BNF^{++}:n “$^{+}$” ei lisää ilmaisuvoimaa, koska jokaisen kaavan muotoa
  lauseke^{+}
  voi kielen muuttumatta korvata kaavalla
  (*lauseke lauseke**)
- myös “*” voidaan poistaa, jos otetaan käyttöön yksi välisymboli lisää:
  lauseketähti ::= ε | *lauseke lauseketähti*
- sulut voidaan poistaa korvaamalla
  (*xxx* … *yyy*)
  uudella välisymbolilla
  sulkuhäkkyrä ::= *xxx* … *yyy*
- sama vapaaehtoisuudelle [*xxx* … *yyy*]
  vapaahäkkyrä ::= ε | *xxx* … *yyy*
- nyt kaikki säännöt ovat muotoa
  kieli ::= *xxx* … *xxx* | *xxx* … *xxx* | … | *xxx* … *xxx*
  missä *xxx* on loppusymboli, välisymboli tai ε
- alkuperäinen BNF oli jokseenkin tarkalleen tällainen
  – välisymbolien nimet ympäröitiin kulmamerkeillä, esim. ⟨*kieli*⟩, ja saivat sisältää tyhjää
  – loppusymboleita ei ympäröity lainausmerkeillä
  – ε esitettiin ⟨*empty*⟩
- alkuperäinen BNF: Naur, P. (toim.): Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, Vol. 3, No. 5, pp. 299–314, 1960.

Yhteysriippumattomat kieliopit

- jos sallitaan välisymbolin määrittely > 1 säännöllä, jotka tulkitaan vaihtoehtoisiksi, tullaan toimeen myös ilman valintaoperaattoria “|”:
  kieli ::= $lauseke_1$ | $lauseke_2$ | … | $lauseke_n$
  korvataan
  kieli ::= $lauseke_1$
  kieli ::= $lauseke_2$
  …
  kieli ::= $lauseke_n$
- ε on tarpeeton tulossa xxx_1 xxx_2 … xxx_n, paitsi jos se on tulon ainoa tekijä
  ⇒ jätämme turhat ε:t pois
- näiden muutosten jälkeen jokainen sääntö on muotoa
  kieli ::= *lauseke*
  missä *lauseke* on joko ε tai epätyhjä jono loppu- ja / tai välisymboleita
- välisymboli voi olla kokonaan ilman sääntöä
  – tuottaa kieleksi Ø
  – saataisiin myös säännöllä tyyppiä A ::= A
- jos leksikaalinen taso unohdetaan, voimme olettaa, että Σ ja Ψ ovat vain kaksi äärellistä joukkoa symboleita
  – koska olemme vaatineet, että loppusymbolit erottuvat välisymboleista, pätee $\Sigma \cap \Psi = \emptyset$
- koska “::=” on vain välimerkki ja ε esittää tyhjää merkkijonoa, on sääntö itse asiassa vain pari muotoa (A, σ), missä $A \in \Psi$ ja $\sigma \in (\Sigma \cup \Psi)^*$

- formalismimme on nyt muotoa
  $(\Psi, \Sigma, \Pi, \mathcal{S})$
  missä
  – Σ ja Ψ ovat äärellisiä joukkoja ja $\Sigma \cap \Psi = \emptyset$
  – $\Pi \subseteq \Psi \times (\Sigma \cup \Psi)^*$
  – $\mathcal{S} \in \Psi$
- tällainen formalismi on *yhteysriippumaton kielioppi* (*context-free grammar*, *CFG*)
- varoitus: osien tunnusmerkit vaihtelevat eri kirjoittajilla, ja Ψ:n tilalla saattaa olla $\Sigma \cup \Psi$
- helppouden vuoksi sallimme merkinnän
  $kieli$ ::= $lauseke_1$ | $lauseke_2$ | … | $lauseke_n$
  käytön lyhennemerkintänä säännöille
  $(kieli, lauseke_1) \in \Pi, \ldots, (kieli, lauseke_n) \in \Pi$

Yhteysriippumattomat kielet

- yhteysriippumattomalla kieliopilla määriteltävissä oleva kieli on *yhteysriippumaton kieli* (*context-free language*, *CFL*)
- jos leksikaalinen taso unohdetaan, yhteysriippumattomat kieliopit ovat yhtä vahva formalismi kuin BNF^{++}
  – näimme jo, miten BNF^{++}-määritelmä voidaan muuntaa saman kielen muodostavaksi yhteysriippumattomaksi kieliopiksi
  – jokainen yhteysriippumaton kielioppi voidaan muuntaa BNF^{++}-määritelmäksi liittämällä saman välisymbolin vaihtoehtoiset määritelmät yhdeksi määritelmäksi “|”:n avulla (ym. lyhennemerkintä)

$\Rightarrow$ BNF^{++} kykenee määrittelemään kaikki yhteysriippumattomat kielet, ja vain ne

Yhteysriippumattomat kieliopit ovat hyvin yksinkertainen formalismi

- käytännön määrittelytyössä kömpelö
- kätevä kääntäjien rakentamisessa (seuraava alaluku)
- sopiva teoreettisen tutkimuksen kohteeksi
  - siten saadut yhteysriippumattomia kieliä koskevat tulokset pätevät myös BNF^{++}:lle ja graafisille syntaksikaavioille

Yhteysriippumattomien kielten ilmaisuvoima

- jokainen säännöllinen lauseke voidaan tulkita BNF^{++}-lausekkeeksi

$\Rightarrow$ säännölliset kielet $\subseteq$ yhteysriippumattomat kielet

- näimme edellä, että $\{ a^n b^n \mid n \in \boldsymbol{N} \}$ ei ole säännöllinen kieli
- on helppo nähdä, että se on yhteysriippumaton:
  A ::= ε | “a” A “b”

$\Rightarrow$ säännölliset kielet $\neq$ yhteysriippumattomat kielet

6.2 Jäsentämisestä

Yhteysriippumattoman kieliopin määrittelemä kieli

- edellä määrittelimme, että

  ***Yhteysriippumaton kielioppi** (**context-free grammar**, **CFG**) on nelikko* $(\Psi, \Sigma, \Pi, \mathcal{S})$*, missä*
  - Σ *ja* Ψ *ovat äärellisiä joukkoja ja* $\Sigma \cap \Psi = \emptyset$*,*
  - $\Pi \subseteq \Psi \times (\Sigma \cup \Psi)^*$*, ja*
  - $\mathcal{S} \in \Psi$*.*
- merkkijonon johtaminen määritellään hieman eri tavalla kuin BNF^{++}:n yhteydessä:

  Olkoon $G = (\Psi, \Sigma, \Pi, \mathcal{S})$ *yhteysriippumaton kielioppi ja* α*,* $\beta \in (\Sigma \cup \Psi)^*$*. Relaatiot* $\Rightarrow$ *ja* $\Rightarrow^* \subseteq (\Sigma \cup \Psi)^* \times (\Sigma \cup \Psi)^*$ *määritellään seuraavasti:*
  - $\alpha \Rightarrow \beta$ *jos ja vain jos on olemassa* $\beta_1, \beta_2, \beta_3 \in (\Sigma \cup \Psi)^*$ *ja* $A \in \Psi$ *siten, että* $\alpha = \beta_1 A \beta_3$*,* $\beta = \beta_1 \beta_2 \beta_3$*, ja* $(A, \beta_2) \in \Pi$*.*
  - $\alpha \Rightarrow^* \beta$ *jos ja vain jos on olemassa* $n \geq 0$ *ja* $\alpha_0, \ldots, \alpha_n$ *siten, että* $\alpha = \alpha_0$*,* $\alpha_n = \beta$*, ja* $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$*.*

  Terminologiaa:
  - *Jos* $\alpha \Rightarrow \beta$*, niin* α ***tuottaa suoraan** (**johtaa suoraan**, **derives directly**)* β*:n.*
  - *Jos* $\alpha \Rightarrow^* \beta$*, niin* α ***tuottaa** (**johtaa**, **derives**)* β*:n.*
  - *Jos* $\mathcal{S} \Rightarrow^* \beta$*, niin* β *on **lausejohdos** (**sentential form**).*
  - *Jos* $\mathcal{S} \Rightarrow^* \beta$ *ja* $\beta \in \Sigma^*$*, niin* β *on **lause** (**sentence**).*

  *Kieliopin G **tuottama kieli** (**language generated by** G) on G:n lauseiden joukko, eli*

  $$\mathcal{L}(G) := \{ \alpha \in \Sigma^* \mid \mathcal{S} \Rightarrow^* \alpha \}$$

  ***Yhteysriippumaton kieli** (**context-free language**, **CFL**) on kieli, jonka voi tuottaa yhteysriippumattomalla kieliopilla.*

- siis
  - johdetussa merkkijonossa saa nyt olla myös välisymboleita
  - johtaminen on nyt (ehkä merkkijonojen sisällä olevien) välisymbolien korvaamista merkkijonoilla
  - johtaminen etenee nyt “ylhäältä alas”
- eroista huolimatta määritelty kieli ei muutu
  - jos $\alpha \in \Sigma^*$ ja $A \in \Psi$, niin $A \Rightarrow^* \alpha^*$ uuden määritelmän mukaan jos ja vain jos $A \Rightarrow^* \alpha^*$ BNF^{++}:n määritelmän mukaan

(Transitiiviset sulkeumat

- binäärirelaatio (= 2-paikkainen relaatio) $\Rightarrow^*$ on määritelty binäärirelaation $\Rightarrow$ *refleksiivisenä transitiivisena sulkeumana*
- = relaatio, joka saadaan alkuperäisestä “toistamalla” nolla tai useampia kertoja
- matematiikassa on tapana merkitä minkä tahansa binäärirelaation ~ refleksiivistä transitiivista sulkeumaa symbolilla ~*
  - $a \sim^* b \;:\Leftrightarrow\; \exists\, n;\, n \geq 0\colon \exists\, a_0, \ldots, a_n\colon a = a_0 \wedge a_n = b \wedge a_0 \sim a_1 \wedge a_1 \sim a_2 \wedge \ldots \wedge a_{n-1} \sim a_n$
- ~:n *transitiivinen sulkeuma* $\sim^+$ saadaan “toistamalla” yksi tai useampia kertoja
  - $a \sim^+ b \;:\Leftrightarrow\; \exists\, n;\, n \geq 1\colon \exists\, a_0, \ldots, a_n\colon a = a_0 \wedge a_n = b \wedge a_0 \sim a_1 \wedge a_1 \sim a_2 \wedge \ldots \wedge a_{n-1} \sim a_n$)

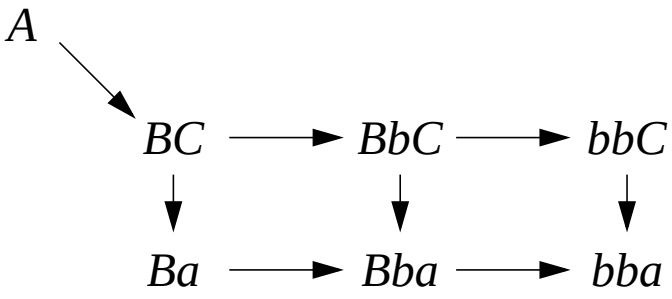
Samalle merkkijonolle on usein useita erilaisia tapoja johtaa se

- esimerkki:

  $A ::= BC$
  $B ::= b \mid Bb$
  $C ::= a \mid c$

  – $A \Rightarrow BC \Rightarrow BbC \Rightarrow bbC \Rightarrow bba$
  – $A \Rightarrow BC \Rightarrow BbC \Rightarrow Bba \Rightarrow bba$
  – $A \Rightarrow BC \Rightarrow Ba \Rightarrow Bba \Rightarrow bba$

- kuvana



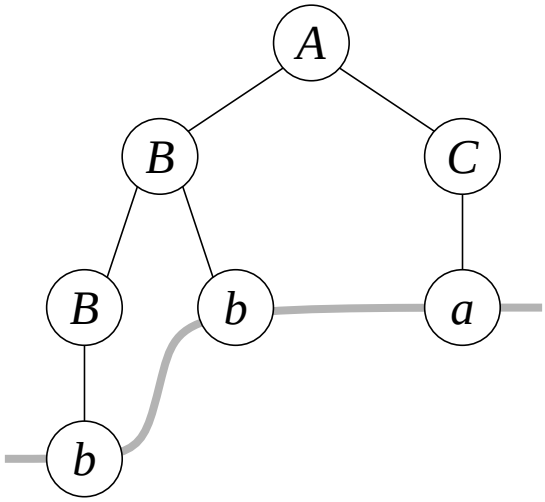
- tässä esimerkissä eri tavat erosivat vain sen suhteen, missä järjestyksessä saman merkkijonon välisymbolit käsiteltiin
- järjestys on lähes aina täysin epäolennainen

⇒ otamme käyttöön toisen tavan esittää merkkijonon johtaminen, joka ei ota kantaa järjestykseen

Jäsennyspuu

- esittää merkkijonon johtamisen CFG:stä (Ψ, Σ, Π, $\mathcal{S}$) ottamatta kantaa askelien järjestykseen
- puun solmut on nimetty väli- tai loppusymboleilla tai ε:lla
  – eri solmuilla saa olla sama nimi
  – lehtien ja vain lehtien niminä on loppusymboleita tai ε

- juuren nimenä on alkusymboli $\mathcal{S}$
- jos lehtien nimet vasemmalta oikealle ovat a_1, a_2, …, a_n, niin johdettu merkkijono on $a_1a_2\ldots a_n$ turhat ε:t poistettuna
- jos solmun nimi on A ja sen lasten nimet vasemmalta oikealle ovat X_1, …, X_n, niin $(A, X_1X_2\ldots X_n) \in \Pi$
  - $(A, X_1X_2\ldots X_n)$ on johtamisessa käytetty sääntö
- esimerkki: $A \Rightarrow^* bba$



Samalla kielellä voi olla monta eri kielioppia

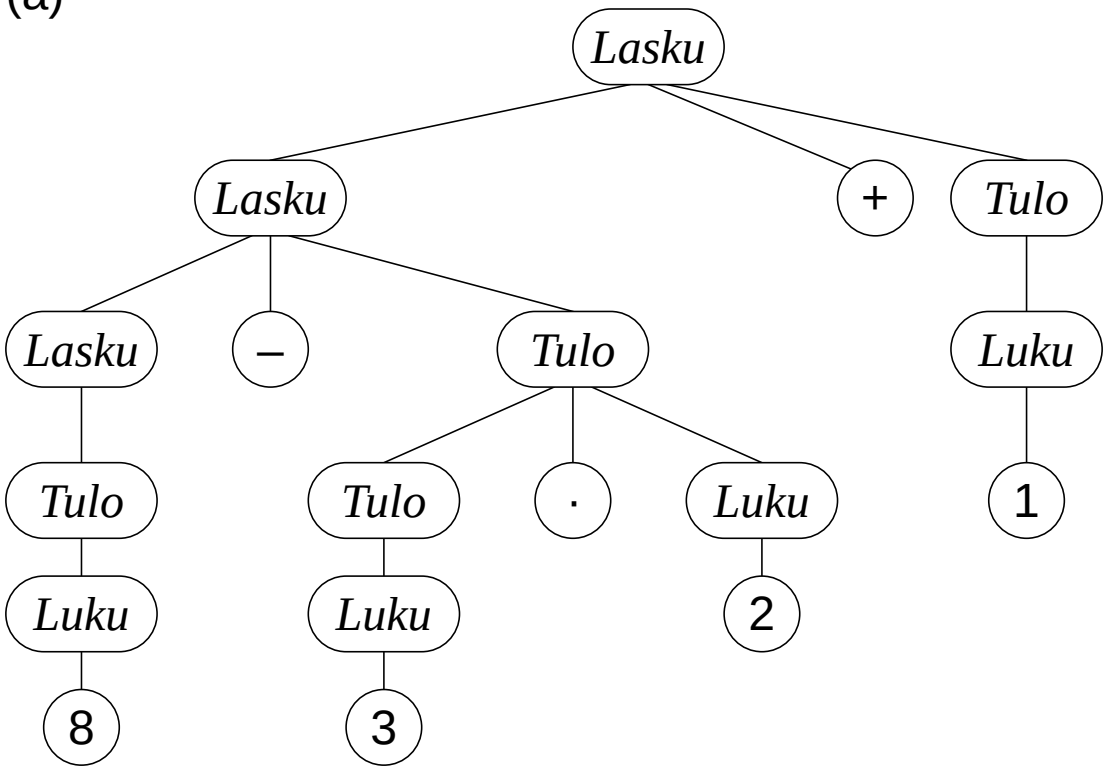
- (yksinkertaistettu) esimerkki:

(a) *Lasku* ::= *Tulo* | *Lasku* “+” *Tulo* | *Lasku* “–” *Tulo*
Tulo ::= *Luku* | *Tulo* “·” *Luku* | *Tulo* “/” *Luku*
Luku ::= “0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”

(b) *Lasku* ::= *Summa* | *Lasku* “·” *Summa* |
Lasku “/” *Summa*
Summa ::= *Luku* | *Summa* “+” *Luku* | *Summa* “–” *Luku*
Luku ::= “0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”

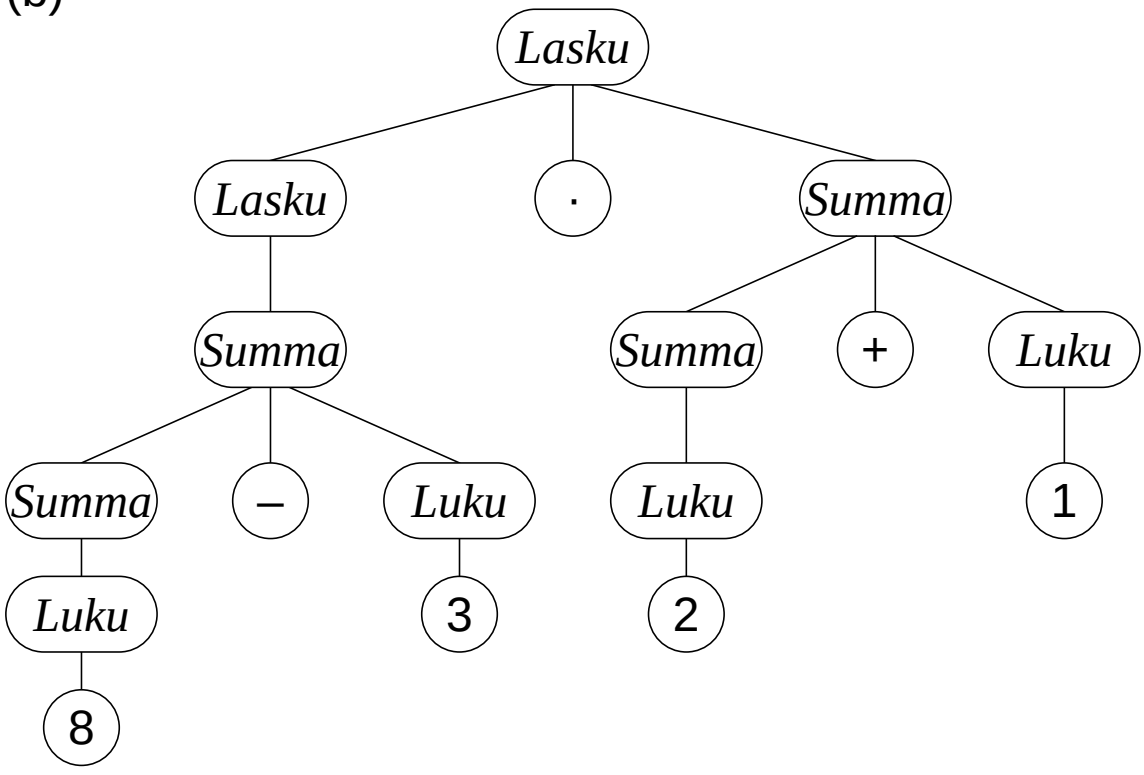
(c) *Lasku* ::= *Luku* | *Lasku* “+” *Luku* | *Lasku* “–” *Luku* |
Lasku “·” *Luku* | *Lasku* “/” *Luku*
Luku ::= “0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”

- jäsennyspuita merkkijonolle 8–3·2+1
  (a)



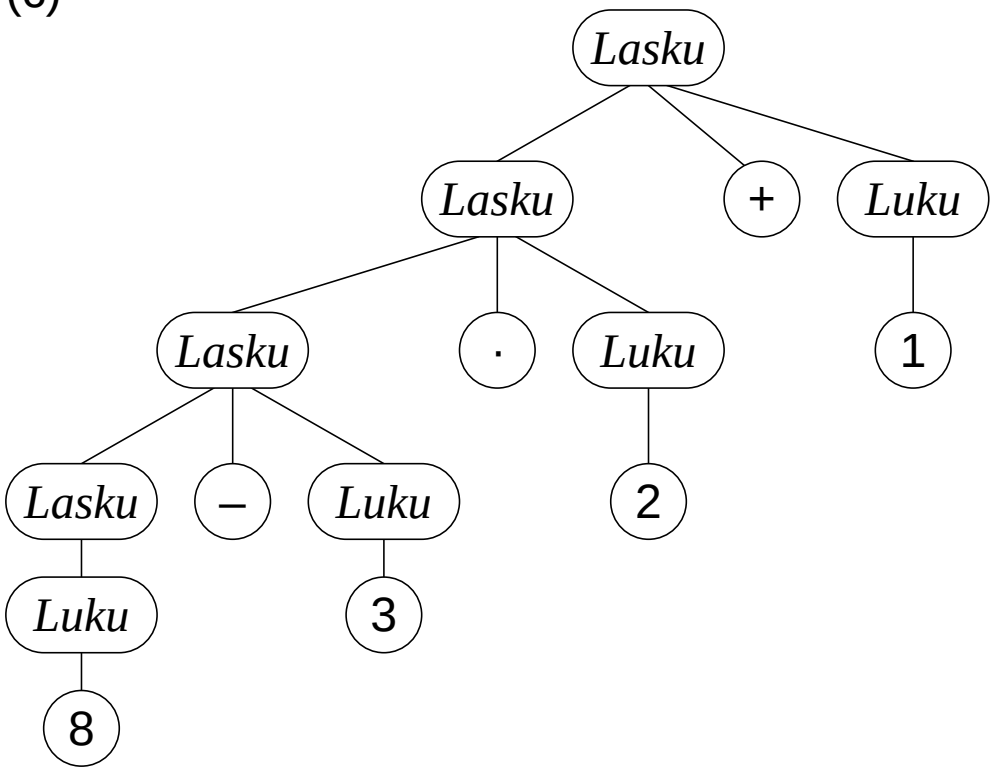
  – vrt. 8 – 3·2 + 1 = 8 – 6 + 1 = 2 + 1 = 3

  (b)



  – vrt. 8–3 · 2+1 = 5 · 3 = 15

(c)



– vrt. 8–3 · 2 + 1 = 5 · 2 + 1 = 10 + 1 = 11

⇒ eri kieliopit tuottavat usein samalle merkkijonolle erilaisia jäsennyspuita

Huom! jäsennyspuun rakenne ei välttämättä määrää laskujärjestystä

- ensiksi “operaattori” ja “laskujärjestys” eivät aina ole mielekkäitä käsitteitä
  - kaikki kielet eivät muodostu (vain) lausekkeista!
  - *desim.luku* ::= *etumerkki numerojono* “.” *numerojono*
    numerojono ::= *numero* | *numero numerojono*
    etumerkki ::= ε | “+” | “–”
- toiseksi on mahdollista jäsentää lauseke eri järjestyksessä kuin se lasketaan
  - jäsentäminen on helpompaa oikealle kuin vasemmalle sitovien operaattoreiden tapauksessa
  - ⇒ lausekelaskin voi jäsentää toisin kuin laskee

- silti jäsennyspuun rakenne antaa usein vihjeitä kielelle tarkoitetusta merkityksestä (eli semantiikasta)
  ⇒ jäsennyspuita käytetään hyväksi merkkijonojen käsittelyssä, esim. kääntämisessä
  – kohta tulee esimerkki: *attribuuttikieliopit*
- kuitenkaan kielioppien erot eivät tee kielistä eri kieliä, jos johdettavissa olevat merkkijonojen joukot säilyvät samoina!
  – määritelmä ⇒ kieli on **vain** Σ*:n osajoukko
- käyttämämme kielen määritelmä ottaa huomioon vain syntaksin
  – määritelmä pitää (ja voidaan) vaihtaa jos semantiikkakin halutaan ottaa huomioon

⇒ jäsennyspuiden erot eivät liity itse kieleen, vaan sen tuottamisessa käytettyyn kielioppiin

Attribuuttikieliopit

- *attribuuttikieliopit* (*attribute grammars*) ovat yksinkertainen ja kätevä tapa määritellä syntaksin ohjaamaa laskentaa
- idea: liitetään väli- ja loppusymboleihin *attribuutteja* ja niiden laskentasääntöjä
- attribuutit ovat ikään kuin tietueiden kenttiä
- laskentasäännöissä saa viitata vain tietueen ja sen lasten attribuutteihin
- laskentajärjestyksen on vastattava jäsennyspuuta

- esimerkki:

$Lasku ::= Tulo_1$	$Lasku.x := Tulo_1.x$
$Lasku ::= Lasku_1 \text{“+”} Tulo_1$	$Lasku.x := Lasku_1.x + Tulo_1.x$
$Lasku ::= Lasku_1 \text{“–”} Tulo_1$	$Lasku.x := Lasku_1.x - Tulo_1.x$
$Tulo ::= Luku_1$	$Tulo.x := Luku_1.x$
$Tulo ::= Tulo \text{“·”} Luku$	$Tulo.x := Tulo_1.x \cdot Luku_1.x$
$Luku ::= Nro_1$	$Luku.x := Nro_1.x$
$Luku ::= Luku_1\ Nro_1$	$Luku.x := 10 \cdot Luku_1.x + Nro_1.x$
$Nro ::= \text{“0”}$	$Nro.x := 0$
…	
$Nro ::= \text{“9”}$	$Nro.x := 9$

- kätevä toteuttaa, kun jäsennyspuu (tai vastaavat tiedot) ovat käytettävissä

⇒ kätevä niille kieliopeille, jotka vastaavat oikeaa laskujärjestystä ja joiden mukaan on helppo jäsentää

Lausekkeiden rakenteen kertausta

- kuten luvussa 4 todettiin, matematiikka ja tietojenkäsittelytiede ovat tulvillaan kieliä, joissa on lausekkeita, jotka koostuvat
  - operaattoreista (esim. “+”, “·”, “∧”, “¬”) ja
  - operandeista (esim. muuttujat, luvut, **F**, **T**, osalausekkeet)
- jos moniosainen operaattorisymboli ympäröi operandinsa kokonaan, on laskujärjestys ilman muuta selvä
  - esim. Ada: **if** … **then** … **elsif** … **else** … **end if**
  - kampamainen rakenne
- samoin jos operaattorin yhteydessä on pakko käyttää riittävästi välimerkkejä kuten “(”, “)” ja “,”
  - matemaatikot suosivat tätä formalismeja esitellessään

- muussa tapauksessa tarvitaan lisäsääntöjä laskujärjestyksen määräämiseen
  - usein sitovuustasot ja sitovuuden suunnat
  - vrt. edellä 8–3·2+1

Sitovuustasot

- esimerkiksi matematiikassa "·" sitoo voimakkaammin kuin "+"
  - esim. 8–3·2+1
- sitovuustasot voidaan ottaa huomioon kieliopissa
  - kääntämisen aikana helppo noudattaa sitä
  - esim.

(a) *Lasku* ::= *Tulo* | *Lasku* "+" *Tulo* | *Lasku* "–" *Tulo*
Tulo ::= *Luku* | *Tulo* "·" *Luku* | *Tulo* "/" *Luku*
Luku ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

(b) *Lasku* ::= *Summa* | *Lasku* "·" *Summa* |
Lasku "/" *Summa*
Summa ::= *Luku* | *Summa* "+" *Luku* | *Summa* "–" *Luku*
Luku ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

(c) *Lasku* ::= *Luku* | *Lasku* "+" *Luku* | *Lasku* "–" *Luku* |
Lasku "·" *Luku* | *Lasku* "/" *Luku*
Luku ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

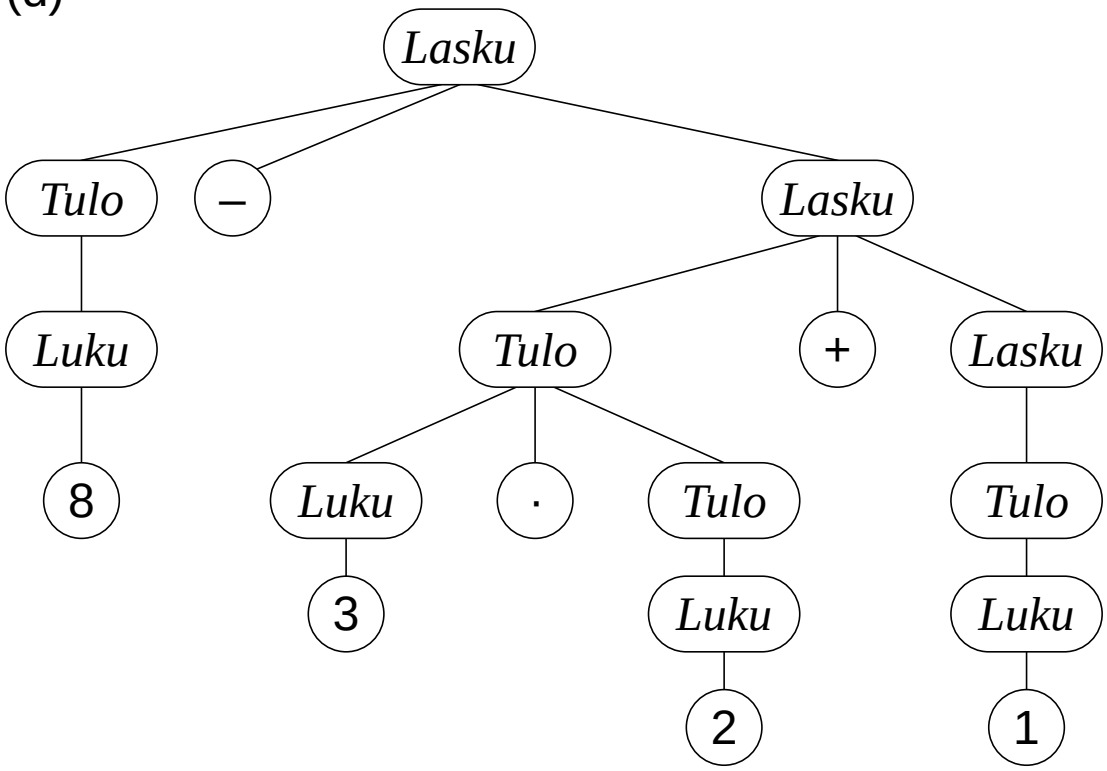
Sitovuuden suunta

- laskentajärjestys saman sitovuustason operaattorien välillä on myös tärkeä
  - esim. 8 – 6+1 = 8 – 7 = 1
    ≠ 8–6 + 1 = 2 + 1 = 3
- sitovuuden suuntakin voidaan ottaa huomioon kieliopissa (jos osataan kääntää ko. kieliopin mukaan):

(a) *Lasku* ::= *Tulo* | *Lasku* “+” *Tulo* | *Lasku* “–” *Tulo*
Tulo ::= *Luku* | *Tulo* “·” *Luku* | *Tulo* “/” *Luku*
Luku ::= “0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”

(d) *Lasku* ::= *Tulo* | *Tulo* “+” *Lasku* | *Tulo* “–” *Lasku*
Tulo ::= *Luku* | *Luku* “·” *Tulo* | *Luku* “/” *Tulo*
Luku ::= “0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”

(d)



8 – 3·2 + 1 = 8 – 6 + 1 = 8 – 7 = 1

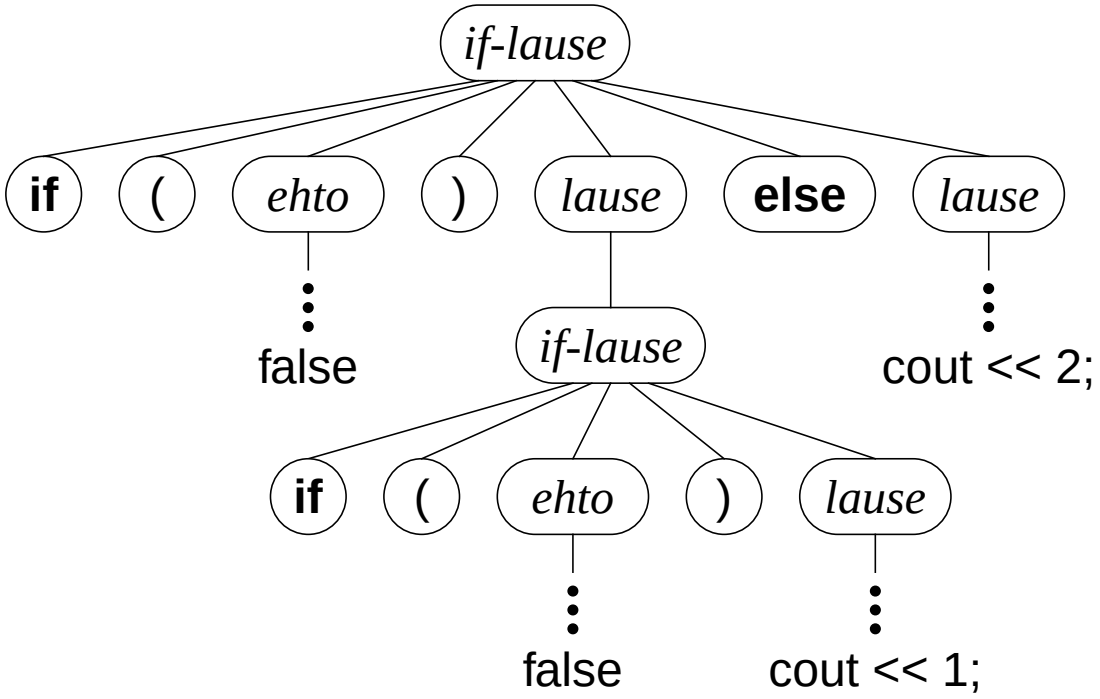
Moniselitteiset kieliopit

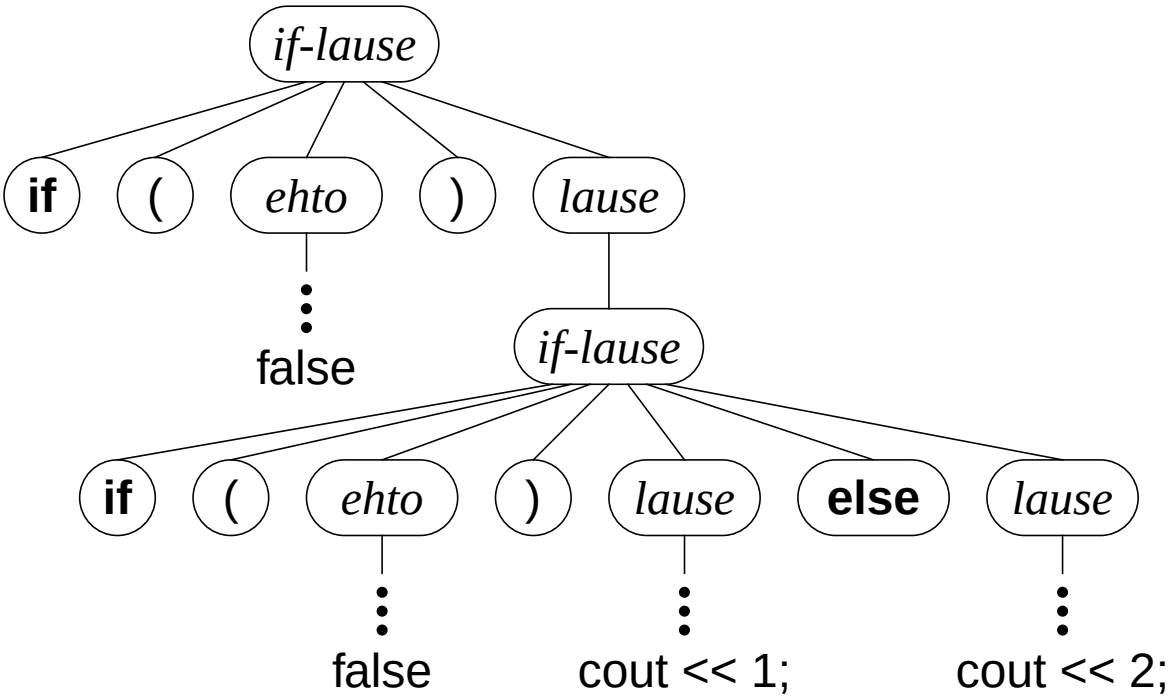
- joskus samalle loppusymbolien jonolle voidaan antaa kaksi eri jäsennyspuuta samassa kieliopissa
- tällöin sanotaan, että kielioppi on *moniselitteinen* (*ambiguous*)
- aiheuttaa väärän tulkinnan vaaran
  – varsinkin jos lukija ei huomaa moniselitteisyyttä!

Klassinen esimerkki C++:n avulla esitettynä:

if-lause ::= “**if**” “(” *ehto* “)” *lause* |
“**if**” “(” *ehto* “)” *lause* “**else**” *lause*

- mitä seuraava tulostaa?
  if(false) **if**(false) cout << 1; **else** cout << 2;





- ohjelmointikielten määrittelyissä tämä esimerkki on tapana ratkaista sanallisesti
- esimerkki: The Annotated C++ Reference Manual, 1994, sivu 85:

> *The* `else` *ambiguity is resolved by connecting an* `else` *with the last encountered* `else`*-less* `if`.

Jos itse pääsee suunnittelemaan kielen ja kieliopin, kannattaa moniselitteisyyttä välttää

- esimerkki: Ada:
  if *ehto* **then** *lause*$^+$ (**elsif** *ehto* **then** *lause*$^+$)*
  [**else** *lause*$^+$] **end if**;
- valitettavasti kieliopista on joskus aika vaikea nähdä, onko se moniselitteinen
  – algoritmisesti ratkeamaton tehtävä
- sitäpaitsi on olemassa yhteysriippumattomia kieliä, joille on vain moniselitteisiä kielioppeja
  – esim. $\{ a^i b^j c^k \mid i = j \vee j = k \}$ on yhteysriippumaton

$$\begin{array}{llll} \ddot{O} & ::= EC \mid AF & & \\ A & ::= \varepsilon \mid aA & C & ::= \varepsilon \mid cC \\ E & ::= \varepsilon \mid aEb & F & ::= \varepsilon \mid bFc \end{array}$$

  – $a^n b^n c^n$ voidaan johtaa sekä EC:stä että AF:stä
  – voidaan todistaa, että ko. kielen kaikki muutkin kieliopit ovat moniselitteisiä

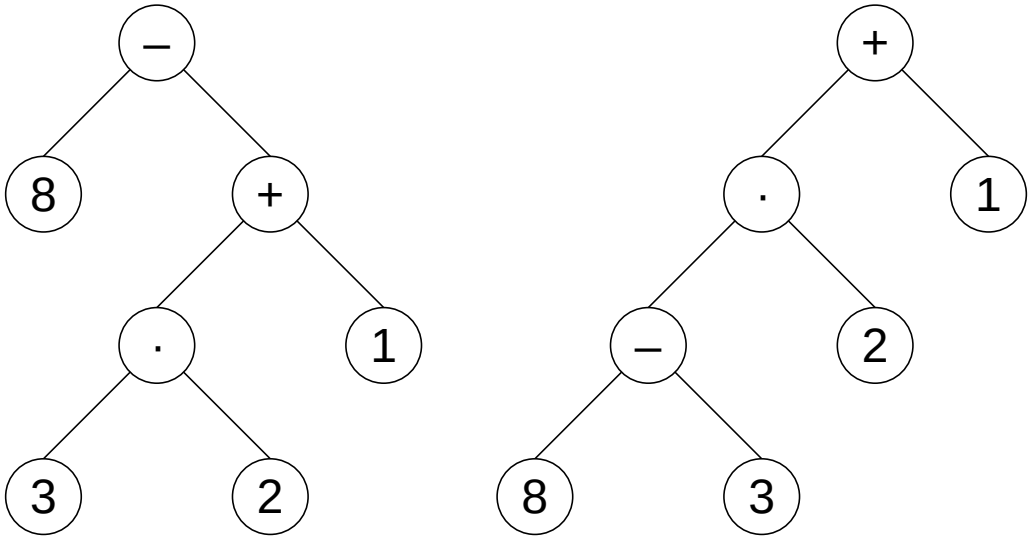
Lausekkeen esittäminen abstraktina puuna (lausekepuu)

- jäsennyspuissa lausekkeiden esittämistä tarkasteltiin konkreettisen syntaksin tasolla
  – sulut, operaattorisymbolin sijainti operandeihin nähden yms. näkyvät
  – jäsennyspuissa paljon välisymboleiden nimiä

- usein lausekkeita ajatellaan esitystavasta riippumattomasti matemaattisina abstraktioina
  - operaattori edustaa funktiota
  - lauseke on rykelmä sisäkkäisiä funktiokutsuja

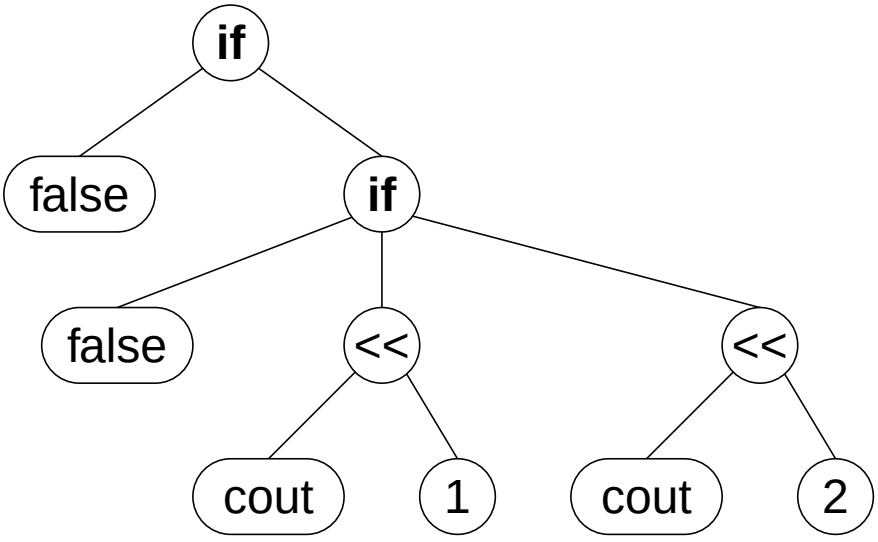
⇒ saadaan jäsennyspuita tiiviimpi puuesitys
  - luvun 4.1 esitys

- esimerkki (matematiikasta tutuilla sitovuussäännöillä): $8 - (3 \cdot 2 + 1)$ ja $(8-3) \cdot 2 + 1$



- esimerkki (ohjelmointikielen lause tulkittuna lausekkeeksi):

  if(false) **if**(false) cout << 1; **else** cout << 2;



- puut ovat käteviä lausekkeiden käsittelyssä
  - voidaan jopa ajatella, että puu on asian ydin, ja lauseke on vain keino esittää se tekstuaalisesti
- jos kielioppi jäsentää lausekkeet halutulla tavalla, lauseketta esittävä puu saadaan jäsennyspuusta
  - siirtämällä välisymbolisolmuun riittävät tiedot operaation luonteesta (esim. “+”, “**if**”)
  - jättämällä turhat välisymbolit pois (esim. “*tulo*”)
  - jättämällä välimerkit yms. pois (esim. sulut)

Pinoautomaatit: johdanto

- näimme edellä, että jokaiselle säännölliselle kielelle on olemassa äärellinen automaatti, joka hyväksyy siihen kuuluvat merkkijonot ja vain ne
- äärelliset automaatit (varsinkin deterministiset) ovat käytännössä käteviä tarkastettaessa merkkijonon kuulumista annettuun säännölliseen kieleen
- usein on tarpeen selvittää, kuuluuko annettu loppusymbolien jono annettuun BNF^{++}:lla määriteltyyn (eli yhteysriippumattomaan) kieleen
  - esim. onko merkkijono syntaktisesti oikeaa C++:aa
- jos vastaus on “kyllä”, usein tarvitaan myös jäsennyspuu (tai sama informaatio muussa muodossa)
  - ts. syöte halutaan *jäsentää* (*parse*)
- **huom!** “parsing” on yhtä paljon “parsimista” kuin “apple” on “appelsiini”!
  - “parsiminen” sopisi paremmin esim. termin “backpatching” (tai “patching”?) suomennokseksi

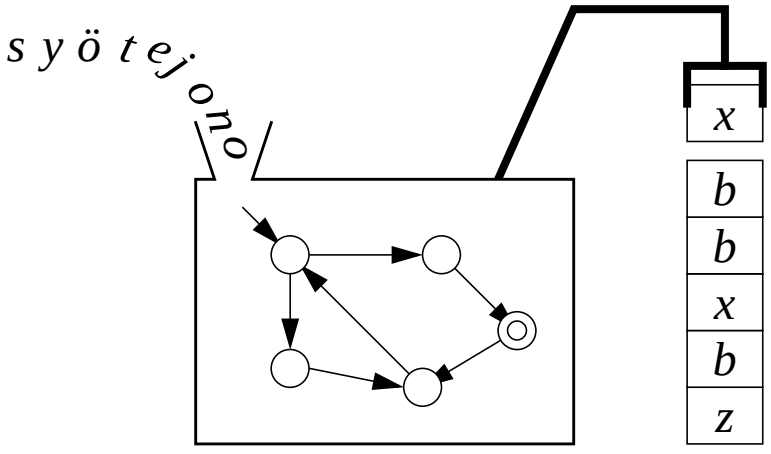
$\Rightarrow$ automaattiluokka yhteysriippumattomille kielille olisi hyödyllinen

- huomasimme, että jokaisen yhteysriippumattoman kielen voi esittää äärellisen automaatin yleistyksellä, missä kaaren nimenä saa olla myös välisymboli
- tällainen automaatti ei kuitenkaan sellaisenaan kykene lukemaan merkkijonoja
  – miten toteutetaan siirtymät välisymboleilla nimettyjä kaaria pitkin?

$\Rightarrow$ määrittelemme toisenlaisen automaatin yhteysriippumattomien kielten hyväksymistä varten

Pinoautomaatin rakenne

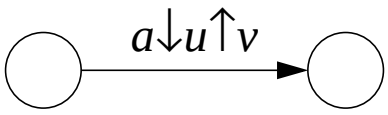
- sopiva kone eli *pinoautomaatti* saadaan lisäämällä äärelliseen automaattiin rajaton pino
- jokaisessa tilasiirtymässä pinoautomaatti voi
  – lukea ja poistaa pinon ylimmän merkin
  – kirjoittaa pinon päällimmäiseksi merkin
- pinoa voi siis käsitellä vain päällimmäisen merkin kohdalta!
  – pinoautomaatti ei pääse esim. kurkkaamaan pinon pohjalle muuten kuin purkamalla koko pinon
- pinon merkit kuuluvat omaan *pinoaakkostoonsa*
  – saa olla sama, osittain sama tai täysin eri kuin syöteaakkosto



Pinoautomaattien määritelmä

- pinoautomaattien määritelmien yksityiskohdat vaihtelevat kirjallisuudessa varsin paljon
- yksi mahdollinen määritelmä

  ***Pinoautomaatti** eli **yhden pinon kone** (**pushdown automaton**) (**PDA**)* $(Q, \Sigma, \Gamma, \Delta, \hat{q}, F)$ *sisältää kuusi osaa:*
  - Q: *äärellinen joukko* ***tiloja** (**state**),*
  - Σ: *äärellinen* ***syöteaakkosto** (**input alphabet**),*
  - Γ: *äärellinen* ***pinoaakkosto** (**stack alphabet**),*
  - $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q \times (\Gamma \cup \{\varepsilon\})$: ***tilasiirtymärelaatio** (**transition relation**),*
  - $\hat{q} \in Q$: ***alkutila** (**initial state**), ja*
  - $F \subseteq Q$: ***lopputilat** (**final states**).*
- Q, Σ, $\hat{q}$ ja F ovat kuten NFA:ssa
- tilasiirtymä koostuu
  - tilasta, josta siirtymä alkaa
  - merkistä, joka luettiin syötteestä tai ε:sta, jos siirtymä tehdään syötettä lukematta
  - merkistä, joka nostettiin pinosta tai ε:sta, jos siirtymä tehdään pinoa katsomatta
  - tilasta, johon siirtymä päättyy
  - merkistä, joka lisätään pinon huipulle tai ε:sta, jos ei lisätä mitään
- tilasiirtymän voisi esittää piirroksessa vaikka seuraavasti:
  - a on syötteestä luettu merkki tai ε
  - u on pinosta luettu merkki tai ε
  - v on pinoon lisätty merkki tai ε



Pinoautomaatin hyväksymä kieli

- pinoautomaatti hyväksyy merkkijonon, jos se voi lukea sen siten, että
  - lukemisen alkaessa pino on tyhjä
  - lopuksi automaatti päätyy johonkin lopputilaansa
- tämä määritelmä sallii, että lopuksi pinoon saa jäädä merkkejä
  - ei olennaista, koska automaatin “loppuun” voidaan lisätä pinon tyhjentäminen lukemalla siitä merkkejä niin kauan kuin niitä riittää
- pinoautomaatin hyväksymä kieli on tietysti sen hyväksymien merkkijonojen joukko
  - Σ^*:n osajoukko (eikä välttämättä Γ^*:n)
- tärkeä lause:

  Kieli on yhteysriippumaton, jos ja vain jos on olemassa pinoautomaatti, joka hyväksyy sen.
- pinoautomaatit voivat olla epädeterministisiä
  $\Rightarrow$ eivät käytännössä (helposti) toteutettavissa

Deterministiset pinoautomaatit

- pinoautomaatti on deterministinen, jos jokaisessa tilanteessa sillä on enintään yksi mahdollinen siirtymä
- näimme, että jokaiselle NFA:lle on olemassa saman kielen hyväksyvä DFA, mutta pieninkin DFA voi olla aika lailla suurempi kuin alkuperäinen NFA
- PDA:iden tapauksessa epädeterminismin tuoma lisävoima on vielä selvempi:

  On olemassa yhteysriippumaton kieli siten, että ei ole olemassa determinististä PDA:ta, joka hyväksyy sen.

- esimerkiksi seuraava kieli on tällainen:
  $\{ a_1 a_2 \ldots a_n a_n a_{n-1} \ldots a_1 \mid n \in \boldsymbol{N} \wedge a_1, a_2, \ldots, a_n \in \{a, b\} \}$
- yhteysriippumaton kieli on *deterministinen*, jos on olemassa deterministinen PDA, joka hyväksyy sen
- yhteysriippumattomista kieliopeista on olemassa alaluokkia, jotka määrittelevät vain deterministisiä yhteysriippumattomia kieliä
  - $LL(k)$- ja $LR(k)$-kieliopit
  - jäsennettävissä hyvin tehokkaasti
  - $\Rightarrow$ tärkeitä kääntäjätekniikassa
- $LR(1)$ määrittelee tarkalleen ne kielet, jotka voi hyväksyä deterministisillä PDA:illa

Yhteysriippumattomien kielten pumppauslemma

- yhteysriippumattomille kielille pätee pumppauslemma, joka muistuttaa säännöllisten kielten vastaavaa:

  Olkoon L yhteysriippumaton kieli. On olemassa $k \in \boldsymbol{N}$ *siten, että jos* $\alpha \in L$ *ja* $|\alpha| \geq k$, *niin on olemassa merkkijonot* $\alpha_1, \beta_1, \gamma, \beta_2$ *ja* α_2 *siten, että*
  - $\alpha_1\beta_1\gamma\beta_2\alpha_2 = \alpha$,
  - $\beta_1\beta_2 \neq \varepsilon$,
  - $|\beta_1\gamma\beta_2| \leq k$, *ja*
  - *jokainen merkkijono muotoa* $\alpha_1\beta_1^{\,n}\gamma\beta_2^{\,n}\alpha_2$ *kuuluu L:ään.*
- siis osia β_1 ja β_2 voi toistaa minkä tahansa määrän kertoja, kunhan molempia toistetaan yhtä monta kertaa

- tämän avulla voi mm. todistaa, että $\{ a^n b^n c^n \mid n \in \boldsymbol{N} \}$ ei ole yhteysriippumaton
  - jos valitaan $n \geq k$, on $\beta_1\gamma\beta_2$:ssa enintään kahta eri merkkiä
  - $\Rightarrow$ pumppaus ei säilytä merkkien a, b ja c lukumääriä keskenään yhtäsuurina

$\Rightarrow$ vaikka yhteysriippumattomilla kielillä voi laskea n:ään kahdesti, niin ei enää kolmatta kertaa!

$\Rightarrow$ yhteysriippumattomatkaan kielet eivät riitä määrittelemään esimerkiksi ohjelmointikieltä tarkasti

Kielten määrittelemisestä

- deterministiset yhteysriippumattomat kielet ovat jäsennettävissä tehokkaasti
- epädeterministisille yhteysriippumattomille kielille ei tunneta yhtä tehokkaita jäsentämisalgoritmeja

$\Rightarrow$ kielen syntaksi kannattaa määritellä deterministisen kielen tuottavan kieliopin (esimerkiksi $LR(k)$) avulla

- (determinististen) yhteysriippumattomien kielten ilmaisuvoima ei riitä kovin kummoisiin semanttisiin tarkastuksiin

$\Rightarrow$ kannattaa menetellä seuraavasti:
  - valitaan deterministinen yhteysriippumaton kieli, joka rajaa halutun kielen ylhäältä päin
  - määritellään joukko lisäehtoja sille, että loppusymbolien jono kuuluu kieleen

- usein määritellään lisäehtoja, jotka ovat tarkastettavissa melko helposti sopivan kirjanpidon avulla
  - esim. tyyppitarkastukset, muuttujat määriteltävä ennen käyttöä, aliohjelmakutsussa oltava oikea määrä ja oikean tyyppisiä parametreja, sävelmän tahdin kokonaispituuden oltava oikea, …
  - *staattiset tarkastukset*
- ohjelmointikieliin liittyy lisäksi sääntöjä, jotka voidaan tarkastaa käytännössä vasta suorituksen aikana
  - esim. nollalla ei saa jakaa, taulukon indeksin oltava määritellyissä rajoissa, …
  - *dynaamiset tarkastukset*
- eli meillä on nyt neljä tasoa ohjelmointikieliin liittyviä sääntöjä:
  - leksikaaliset säännöt — äärelliset automaatit
  - syntaksi — (deterministiset) yhteysriippumattomat kielet
  - staattiset tarkastukset — käännösaikaisia
  - dynaamiset tarkastukset — suoritusaikaisia

Hieman syntaksin määrittelemisestä

- syntaksin määritteleminen ei rajoitu ohjelmointikielten suunnitteluun, vaan sitä tarvitaan määriteltäessä mm. seuraavien rakennetta:
  - monimutkaista dataa sisältävät tiedostot
  - protokollat
  - linjalla kulkeva data
- tarkemmat tiedot LR-kieliopeista, jäsentämisestä niiden mukaan jne. kuuluvat kääntäjätekniikan kurssiin

- helposti jäsennettävän syntaksin suunnittelu onnistuu kyllä pelkällä maalaisjärjelläkin, jos on lupa lisätä jäsentämistä ohjaavia välimerkkejä mielin määrin
  - esim. runsas avainsanojen ja erikoismerkkien käyttö ohjelmointikielissä (vrt. Lisp!)
  - esim. nuottikirjoituksen nuotin- ja tauonaloitusmerkit “d” ja “=”
  - esim. kampamaiset rakenteet kuten Adan
    if … **then** … **elsif** … **else** … **end if**;

BNF, laajennettu BNF ja graafiset syntaksikaaviot ovat

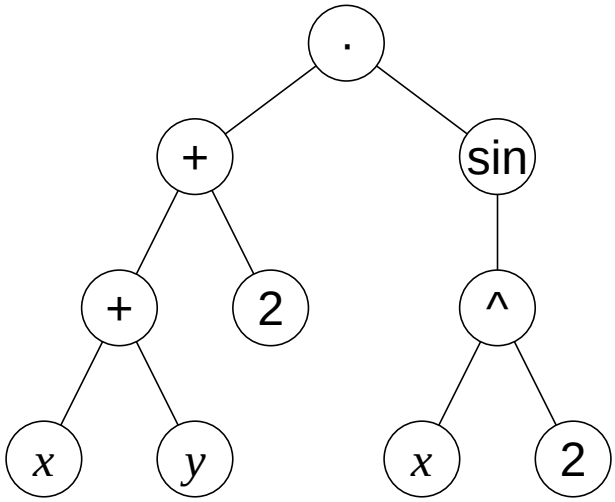
- syntaksin määrittelyyn hyvin riittäviä, jos kaikkia tarkastuksia ei yritetä kaataa syntaksin vastuulle
- olennaisesti täsmällisempiä kuin vaikkapa luonnollinen kieli + esimerkit
- helppoja
- havainnollisia
- tunnettuja jo vuosikymmenet

⇒ ***syntaksimäärittelyn jättäminen epäformaaliksi on anteeksiantamatonta ammattitaidottomuutta!***

6.3 Esimerkki: reaalifunktioiden kieli

Luvussa 4.2 esitettiin yksinkertainen kieli reaalilukujen funktioiden esittämistä varten

- esimerkkifunktio: $(x + y + 2) \cdot \sin x^2$



- funktio esitetään lausekkeena
- operaattoreiden sitovuustasot ja sitovuuden suunnat määriteltiin antamalla seuraava taulukko:

nimi	^	+ ja –	· ja /	sin, cos	+ ja –
par	2	1	2	1	2
taso	5	4	3	2	1
suunta	o	o	v	o	v

Tavoite: tuottaa lausekkeesta puu

- jäsennin palauttaa osoittimen puun solmuun
- kuten edellä, oletetaan, että *uplus_os* osoittaa siihen operaattorin lajitietueeseen, joka edustaa etumerkki-plussaa, jne.
  - yksi tietue operaattoria kohti riittää
  - vakioita on yksi tietue jokaista vakion arvoa kohti
  - muuttujia yksi tietue muuttujaa kohti

- oletetaan, että “uusi(*laji*, *vasen*, *oikea*)” varaa tai löytää solmutietueen ja palauttaa sen osoitteen
  - *oikea*n voi jättää unaarioperaattoreilla pois, myös *vase*mman voi jättää pois vakioilla ja muuttujilla

Sitovuustasojen ja sitovuuden suuntien mukainen kielioppi

- ilman toistoa
  - ts. yhteysriippumattomana kielioppina

Lauseke ::= *Summa*
Summa ::= *Tulo* | *Summa* “+” *Tulo*
| *Summa* “–” *Tulo*
Tulo ::= *Potenssi* | *Tulo* “*” *Potenssi*
| *Tulo* “/” *Potenssi*
Potenssi ::= *Atomilaus.*
| *Atomilaus.* “^” *Potenssi*
Atomilaus. ::= *Muuttuja* | *Lukuvakio*
| “+” *Potenssi* | “–” *Potenssi*
| “sin” *Tulo* | “cos” *Tulo*
| “(” *Lauseke* “)”

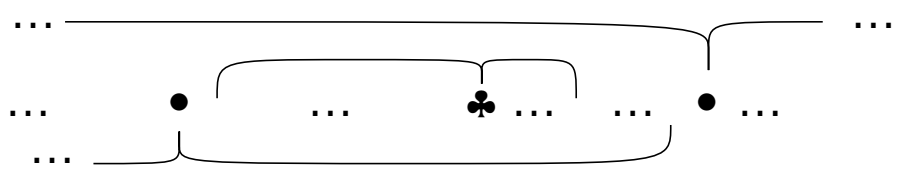
- käyttäen toistoa “*” rekursion sijaan

Lauseke ::= *Summa*
Summa ::= (*Tulo* (“+” | “–”))* *Tulo*
Tulo ::= (*Potenssi* (“*” | “/”))* *Potenssi*
Potenssi ::= *Atomilaus.* (“^” *Atomilaus.*)*
Atomilaus. ::= *Muuttuja* | *Lukuvakio*
| (“+” | “–”) *Potenssi*
| (“sin” | “cos”) *Tulo*
| “(” *Lauseke* “)”

- säännöt “*Lauseke* ::= *Summa*” ovat tietysti tarpeettomia
  - tekevät tarvittaessa helpommaksi lisätä kielioppiin summaa ylempi taso

Sitovuustasojen ja sitovuuden suuntien toteutuminen kieliopissa

- sitovuustasoja ja sitovuuden suuntia tarvitaan kertomaan, kumpaan operaattoriin osalauseke liittyy, jos se on kahden operaattorin välissä



⇒ uusi “paljas” operaattori lopettaa vanhan operaattorin vaikutusalueen, jos ja vain jos
  - (“paljas” = ei sulkujen sisällä, sinin argumenttina tms.)
  - uuden sitovuustaso < vanhan sitovuustaso; tai
  - sitovuustasot samat, ja suunnat vasemmalle
- binäärioperaattoreiden sitovuustasot “^” > “*”, “/” > “+”, “–” on toteutettu kokoamalla *Tulo Potenss*eista ja *Summa Tulo*ista
- binäärioperaattoreiden sitovuuden suuntien ottaminen huomioon näkyy vertaamalla *Potenssi*a *Summa*an ja *Tulo*on
  - rekursio tai toisto “jälkikäteen” tai “etukäteen”
  - ts. esim. *Summa*n vasemmalla puolella kokonainen *Summa*, oikealla pelkkä *Tulo*
- etumerkkien vaikutusalue on määritelty kattamaan vain *Potenssi*n
  ⇒ sitovuustaso < “^”, > muut

- sinin ja kosinin vaikutusalue kattaa vain *Tulo*n
  ⇒ sitovuustaso > binääri “+” ja “–”, < muut
- sulut osoittavat vaikutusalueensa lopun itse
  ⇒ vaikutusaluetta ei tarvitse määritellä
  – kuitenkin on määriteltävä, mitä sulkujen sisällä saa olla: mitä tahansa mitä lausekkeessa saa olla
- sulut, muuttujat, etumerkit jne. on sijoitettu alimmalle tasolle, koska ne sallitaan minkä tahansa osalausekkeen sisällä

Leksikaalinen taso

- oletamme, että käytössä on aliohjelma

  etene_alkio,

  joka etsii ja tunnistaa seuraavan tekstialkion, ja sijoittaa sen koodin muuttujaan

  seuraava_alkio
- etene_alkio ilmaisee tiedoston loppumisen asettamalla

  seuraava_alkio := *tiedoston_loppu*

Jäsentämisen toteutus: johdanto

- ns. “rekursiivisen laskeutumisen” (*recursive-descent*) menetelmä
  – kutakin välisymbolia vastaa aliohjelma
  – aliohjelmat voivat kutsua toisiaan rekursiivisesti
- etuja
  – hyvin helppo toteuttaa
  – toimii aina kun kielessä on riittävästi jäsentämistä ohjaavia avainsanoja ja välimerkkejä

- ongelma: kun *Summa* alkaa, ei tiedetä onko se *Summa* (“+” | “–”) *Tulo* vai pelkkä *Tulo*
  ⇒ ei tiedetä, kumpaa aliohjelmaa pitäisi kutsua
  – jos aina kutsuttaisiin *Summa*a, saataisiin ikuinen rekursio

⇒ on vaikea jäsentää ilman toistoa kirjoitetun kieliopin mukaan

⇒ jäsennämme toistoa “*” käyttävän kieliopin mukaan
  – *Summa*n alussa kutsutaan aina *Tulo*a

- usein välisymbolin vaikutusalueen loppuminen nähdään vasta kun on luettu ainakin yksi tekstialkio sen perästä
  – esim. *Tulo* 2 * 3 **+** 4

⇒ osalausekkeen ensimmäinen tekstialkio voi olla luettuna “etukäteen”

- sekaannusten välttämiseksi on helpointa pitää koko ajan yksi tekstialkio luettuna “etukäteen”

⇒ koko lauseke jäsennetään kutsumalla

```
etene_alkio          ⇨ lukee 1. tekstialkion
jäs_koko_syöte
```

Atomilausekkeen jäsentäminen

```
function jäs_atomilauseke : ^solmu
    if seuraava_alkio = muuttujan_nimi then
        etene_alkio
        return uusi( muuttuja_os( nimen_sisältö ) )
    else if seuraava_alkio = numerojono then
        etene_alkio
        return uusi( vakio_os( luvun_sisältö ) )
    else if seuraava_alkio = miinus_symboli then
        etene_alkio
        return uusi( umiinus_os, jäs_potenssi )
    else if seuraava_alkio = plus_symboli then
        …                          ⇨ kuten etumerkki-miinus
    else if seuraava_alkio = sin_symboli then
        etene_alkio
        return uusi( sin_os, jäs_tulo )
    else if …                                  ⇨ cos, tan jne.
    else if seuraava_alkio = sulku_auki then
        etene_alkio
        tulos := jäs_lauseke
        if seuraava_alkio ≠ sulku_kiinni then
            virhe( “loppusulku puuttuu” )
        else
            etene_alkio; return tulos
    else virhe( “odotettiin atomilauseketta” )
    endif
endfunc
```

Potenssilaskun jäsentäminen

- sitoo oikealle
  - *Potenssi* ::= *Atomilaus.* (“^” *Atomilaus.*)*
  - 2^3^4 = 2^(3^4)

⇒ helppo toteuttaa rekursiolla

– toteutus oikeastaan

Potenssi ::= *Atomilaus.* [“^” *Potenssi*]

```
function jäs_potenssi : ^solmu
    tulos := jäs_atomilauseke
    if seuraava_alkio = pot_symboli then
        etene_alkio
        return uusi( pot_os, tulos, jäs_potenssi )
    else return tulos
    endif
endfunc
```

Kerto- ja jakolaskun jäsentäminen

- sitovat vasemmalle
  – *Tulo* ::= [*Tulo* (“*” | “/”)] *Potenssi*
  – 12 / 4 / 2 = (12 / 4) / 2

⇒ rekursiivinen toteutus johtaisi ikuiseen rekursioon

- onneksi tämä on helppo toteuttaa silmukalla

```
function jäs_tulo : ^solmu
    tulos := jäs_potenssi
    while
        seuraava_alkio ∈ {kerto_symboli, jako_symboli}
    do
        if seuraava_alkio = kerto_symboli then
            etene_alkio
            tulos := uusi( kerto_os, tulos, jäs_potenssi )
        else                ⇨ = jako_symboli
            …
        endif
    endwhile
    return tulos
endfunc
```

Yhteen- ja vähennyslaskun jäsentäminen

- kuten kerto- ja jakolasku

```
function jäs_summa : ^solmu
    tulos := jäs_tulo
    while … ∈ {plus_symboli, miinus_symboli} do
        if seuraava_alkio = plus_symboli then
            etene_alkio
            tulos := uusi( bplus_os, tulos, jäs_tulo )
        else …
    endwhile
    return tulos
endfunc
```

Lausekkeen jäsentäminen

- pelkkä läpikutsu
  - ⇒ hyvä kääntäjä saattaa optimoida pois
  - – esim. C++ `inline` lisää optimoinnin todennäköisyyttä

```
function jäs_lauseke : ^solmu
    return jäs_summa
endfunc
```

Koko syötteen jäsentäminen

- vain tarkastaa, että lausekkeen loputtua syötekin loppuu

```
function jäs_koko_syöte
    tulos := jäs_lauseke
    if seuraava_alkio ≠ tiedoston_loppu then
        virhe( “lauseke loputtua syötteessä roskaa” )
    endif
    return tulos
endfunc
```

Arviointia

- tällä tavalla on helppo toteuttaa jäsennin monille kieliopeille
- näin tehty jäsennin saadaan tulostamaan selkeät syntaksivirheilmoitukset
- jos kieleen voi vaikuttaa, sen voi muuttaa tällä tavalla jäsentyväksi lisäämällä avainsanoja ja välimerkkejä
- käännösaikaiset käsittelyoperaatiot on helppo lisätä jäsentämisen lomaan
  - tässä käsittely oli puun rakentaminen
  - esim. käsittely on laskentaa:

```
function laske_summa : luku
    tulos := laske_tulo
    while … ∈ {plus_symboli, miinus_symboli} do
        if seuraava_alkio = plus_symboli then
            etene_alkio
            tulos := tulos + laske_tulo
        else …
    endwhile
    return tulos
endfunc
```

- näin tehty jäsennin ei ole tehokkain mahdollinen
  - paljon aliohjelmakutsuja
- kääntämiseen sisältyy yleensä kaikkiaan niin paljon työtä, että aliohjelmakutsujen aiheuttama ajan kulutuksen lisäys ei ole olennainen (tai ainakaan sietämätön)
  - tekstialkioiden tunnistus (etene_alkio)
  - lopputuloksen muodostaminen

- rakenteita ei yleensä panna sisäkkäin miten paljon tahansa
  - muutoin esim. ohjelmakoodin sisennys karkaisi ohi paperin oikean reunan

$\Rightarrow$ aliohjelmakutsujen aiheuttama muistin kulutuksen lisäyskään ei ole olennainen (tai ainakaan sietämätön)

- kun rekursiivinen laskeutuminen on liian tehoton tai ei pure kielioppiin, voidaan käyttää muita menetelmiä
  - esim. LR-jäsentäminen (joka on erikoistapaus “shift-reduce” -jäsentämisestä)
- on olemassa automaatteja, jotka tekevät kieliopista jäsentimen
  - esim. Unixin Yacc tuottaa LR-jäsentimen ns. LALR-menetelmällä

Tehon parantaminen aliohjelmakutsuja vähentämällä

- aliohjelmakutsuja voi vähentää kirjoittamalla aliohjelmia “auki” toistensa sisään
- haitta: koodin selkeys kärsii
- joskus kääntäjälle voi ehdottaa, että se kirjoittaa aliohjelman auki
  - C++ `inline`
- auki kirjoittamiselle on eduksi, että aliohjelma kutsuu toista vain yhdessä kohti
  - muutoin koodin kopioiden määrä kasvaa eksponentiaalisesti kutsutasojen määrän suhteen

⇒ voi olla eduksi kirjoittaa silmukoita toiseen muotoon

```
function laske_summa : luku
    eka_kierros := true
    repeat
        if ¬ eka_kierros then
            apu_alkio := seuraava_alkio
            etene_alkio
        endif
        tulos2 := laske_tulo
        if eka_kierros then
            tulos1 := tulos2; eka_kierros := false
        else
            if apu_alkio = plus_symboli then
                tulos1 := tulos1 + tulos2
            else
                tulos1 := tulos1 – tulos2
            endif
        endif
    until seuraava_alkio ∉
        {plus_symboli, miinus_symboli}
    return tulos1
endfunc
```

- auki kirjoittaminen käy hankalaksi, kun aliohjelma kutsuu itseään muuten kuin häntärekursiolla
- esimerkki: *Lauseke* ::= … "(" *Lauseke* ")"
  - jokaisesta sulkutasosta on pidettävä kirjaa missä asiayhteydessä se esiintyy, esim. 1 + (2 * (3 + 4))

  ⇒ jossakin tarvitaan pinomainen tietovarasto
  - jos se ei ole aliohjelmien rekursiopino, niin sitten se on tehtävä itse

- samasta syystä rekursion poisto oikealle sitovan operaattorin aliohjelmasta edellyttää keskeneräisiin lausekkeisiin liittyvän tiedon talletusta jonnekin
  - lausekkeen 5^3^6^… laskemista ei voi (mielekkäästi) aloittaa ennen kuin tiedetään mitä "…" on

7 LOPUKSI

Kävimme läpi niitä matemaattisia rakennelmia, joita käytetään eniten ohjelmistotyössä

- logiikka: esimerkiksi ohjelman tilan määrittely
- abstraktien rakenteiden muodostaminen joukko-opillisesti
- automaatit ja BNF kielten käsittelyssä
- tilakoneet

Tavoitteena oli

- kehittää silmää yksityiskohdille
- oppia keinoja olla täsmällinen siellä, missä siitä on hyötyä
- kehittää matemaattista ajattelua

Kurssin asioiden opiskelua voi jatkaa

- ohjelmien todistaminen
- lausekielten toteutustekniikka
- tilakoneet
- tietojenkäsittelyteoria
- (tietorakenteet ja algoritmit)
- ((formaali spesifiointi))

Mutta sekin on jo hyvä, jos kurssin ansiosta teet vähemmän bugeja spesifikaatioissa ja koodissa!

Onnea tenttiin!

LIITE A: Kurssin pseudokoodi

Noudattaa pitkälti kirjan

> Cormen — Leiserson — Rivest:
> *Introduction to Algorithms*

käytäntöjä

- joissakin kohdin pyritty tutumpiin merkintöihin

for- yms. rakenteellisten lauseiden rajaus osoitetaan avainsanalla **endfor** tms. sekä sisennyksillä

“➪” aloittaa rivin loppuun ulottuvan kommentin

Sijoitusoperaattorina on “:=”

- kuten ohjelmointikielissä Algol, Pascal, Ada, …
- “=” on yhtäsuuruuden vertaaminen
- CLR-kirjassa sijoitusoperaattorina on “←”

+=, –=, ·=… kuten C++

Ellei toisin sanota, kaikki muuttujat ovat paikallisia

Taulukkoon liittyviä oheistietoja esitetään muuttujilla, jotka alkavat taulukon nimellä

- esimerkiksi taulukon A pituus: $A\text{-}length$
  - siis jos A on $A[1 \ldots n]$, niin $A\text{-}length = n$
  - CLR-kirjassa $length[A]$

Tietueen (tai olion) kenttiä osoitetaan pisteen avulla

- esimerkiksi $opiskelija.nimi$, $opiskelija.numero$

- vrt. Pascal

```
PROGRAM TietueKoe( Input, Output );

CONST maxLen = 15;

VAR A : RECORD
    length : Integer;
    contents : PACKED ARRAY[ 1..maxLen ] OF Char
END; (* A *)

BEGIN (* pääohjelma *)
    A.length := 8;
    A.contents := 'Lahtinen       ';
    WITH A DO BEGIN
        contents[2] := 'e';
        writeln( A.contents )
    END (* with *)
END. (* pääohjelma *)
```

- sama C++-kielen merkinnöin
  - todellisuudessa C++-ohjelmoija ei loisi kenttää `length` vaan käyttäisi `contents.length()`

```
#include <iostream>
#include <string>

struct Atype{
    int length;
    std::string contents;
};

int main(){  // pääohjelma
    Atype A;
    A.length = 8; A.contents = "Lahtinen";
    A.contents[1] = 'e';
    std::cout << A.contents << std::endl;
}
```

Osoittimen *x* osoittamaa tietuetta merkitään *x*^

⇒ sen kenttiä merkitään esimerkiksi *x*^.*avain*

- sama tapa kuin esimerkiksi Pascalissa
- C ja C++: `(*x).avain` tai `x->avain`

- CLR-kirjassa *avain*[*x*]
- CLR-kirjan merkintätapa *oheistieto*[*kokonaisuus*] vastaa toteutusta, missä
  - jokaisella tietueella on numero
  - tietueen kenttiä vastaavat ko. numerolla indeksoitavat taulukot
  - esim.

```
int length[ maxLength ],
    contents[ maxLength ];
```

Taulukoilla ja / tai osoittimilla kootun kokonaisuuden nimi tarkoittaa **viitettä** ko. kokonaisuuteen

- sijoituksen A := B jälkeen tietysti $A[i] = B[i]$ kun $1 \leq i \leq A\text{-}length$
- sijoitusten B := A; A[1] := 0; B[1] := 1 jälkeen A[1] = 1 !!
- vrt. C++ &-tyypit

Osoitin tai viite voi kohdistua myös ei minnekään: NIL

Aliohjelmien parametrien välitys

- yksittäiset tiedot: arvon välitys eli kopiointi sisään tultaessa (call-by-value)
  - sama kuin Pascalin, C:n ja C++:n perusmekanismi
- tietokokonaisuudet: viitteen välitys (call-by-reference)
  - sama kuin Pascalin **var**- ja C++:n “&”-parametrit
  - matkittavissa C:n osoittimilla

Loppu

LIITE B: HAKEMISTO

Symbolit

0.–9.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Y

Ä

SISÄLLYSLUETTELO