C-ohjelmointiopas

© 1992 Kenneth S. A. Oksanen,
cessu@cs.hut.fi

ANSI – POSIX –versio

Sisällys

1 Esipuhe **3**

2 Johdanto **4**

3 Sisennys **6**

4 Tunnukset **9**

5 Yksinkertaista! **13**
5.1 Sievennä monimutkaisia lausekkeita . . . 13
5.2 Valitse sopiva rakenne . . . 14
5.3 Muita keinoja yksinkertaistaa . . . 17

6 Funktiot **18**
6.1 Hyvän funktion tunnusmerkkejä . . . 18
6.2 Makrot . . . 20

7 Käyttöjärjestelmän palvelut **22**
7.1 `system()` vs. systeemikutsut . . . 22
7.2 Yleisimpiä systeemikutsut . . . 23

8 Muistin käsittely **24**
8.1 `malloc(), calloc(), realloc(), free()` . . . 24
8.2 Pinon käyttö . . . 25

9 Syötteen lukeminen ja jäsentäminen **27**
9.1 Syötteen varmistettu lukeminen . . . 27
9.2 Syötteen "ymmärtäminen", `lex` ja `yacc` . . . 29

10 Ohjelmointivirheet **41**
10.1 C:n ansoja . . . 41
10.2 `assert()` . . . 43
10.3 `gdb` . . . 43

11 Projekti **45**
11.1 Koodin jakaminen tiedostoihin . . . 45
11.2 Tiedostot valmiissa ohjelmassa . . . 45
11.3 Työkaluajattelu . . . 46
11.4 `make` . . . 46
11.5 `rcs` . . . 47

12 Selväkieliset selostukset **50**
12.1 Koodin kommentointi . . . 50

12.2 Ohjelman dokumentointi 51
12.3 Käyttöohje 51
12.4 UNIXin manuaalisivu 52

13 Yhteenveto 53

A Kirjallisuus 54

B Tiivistelmä 55

1 Esipuhe

• *Tarkoitus*

Aloitteleville ohjelmoijille tarkoitettujen oppikirjojen ja edes välttävän ohjelmointitaidon välillä on huomattava aukko, jota tämä opas yrittää täyttää. Oppaan tarkoitus on lievittää hyvän ohjelmointitaidon oppimisen tuskaa. Sisällön ovat määränneet ihmisten yleisimmät virheet ja vaikeudet.

• *Opetus-menetelmä*

Vaikka oppikirjoissa esitettävät ohjelmanpätkät ovat yleensä[1] varsin taidokkaasti tehtyjä, on harjoitustöistä selvästi nähtävissä, että esimerkeistä ei ole opittu. Tämän oppaan ohjelmointityyliä käsittelevässä osassa käytetään vahvempia opetusmenetelmiä: lukijalle ***näytetään tyypillistä huonoa koodia***, kritisoidaan sitä, esitetään parempi versio ja pyritään lopuksi yleistämään opittuja asioita säännöiksi.

• *Sisältö*

Oppaan luvut 3–9 käsittelevät hyvää ohjelmointityyliä. Luvuissa 6.2.–9 ja varsinkin luvussa 10 esitellään menetelmiä virheiden löytämiseksi ja välttämiseksi. Luvussa 11 annetaan joitakin neuvoja siitä miten tiedostoja tulisi käsitellä ohjelmointiprojektissa. Luku 12 kuvaa hyvän dokumentoinnin ja kommentoinnin perusteita. Opas antaa pinnallisen esittelyn, lähinnä mainoksenomaisen kehoituksen perehtyä syvemmin muutamiin työkaluihin: `lex`, `yacc`, `gdb`, `make` ja `rcs`.

Hyvä ohjelmointitaito on tietotekniikassa välttämätön perustaito. Mutta siihen kuuluu paljon muutakin. Esimerkiksi tärkeimpien algoritmien, tietorakenteiden, ohjelmistometodien ja käyttöjärjestelmien tuntemus on välttämätöntä. Näitä ei tässä oppaassa käsitellä, mutta toivottavasti tämän oppaan ansiosta näiden osa–alueiden kurssien harjoitustöissä menestytään paremmin!

• *Ympäristö*

Opas käsittelee ohjelmointia ANSI–C:llä POSIX–standardin mukaisessa ympäristössä. C:n perusteet ovat ehdottomia esitietoja. Monet hyvän ohjelmointityylin kriteerit pätevät myös muissa lausepohjaisissa ohjelmointikielissä.

• *Opas, ei säännöstö*

Mikäli jostakin tyyliseikasta taitavienkaan ohjelmoijien piirissä ei ole päästy yksimielisyyteen, pyritään esittämään puolueettomasti eri vaihtoehtoja ja herättämään keskustelua. Se, että jokin vaihtoehto on valittu tämän oppaan tyyliksi koodeja esitettäessä ei tarkoita sitä, että kaikki pitäisivät sitä parhaana.

• *Kiitokset*

Kiitokset kaikille opasta sen teko– ja hienosäätövaiheessa oikolukeneille ja arvokasta kritiikkiä antaneille TKK:n tietojenkäsittelyopin opetuslaboratorion "niksulan" hakkereille, erityisesti Tatu Ylöselle, Johannes Helanderille, Tero Kiviselle, Heikki Suonsivulle, Janne Frösenille, Tero Monoselle, Pekka Nikanderille ja Hannu Aronsonille sekä Lars Wirzeniukselle Helsingin yliopistosta. Oikoluvussa ja tekstin ymmärrettävyyden lisäämisessä Jyrki Kasvista, Sami Surakasta, Kalle Korpelasta ja Jari Natusesta on ollut korvaamatonta apua. Kiitokset myös TKK:n tietojenkäsittelyopin laboratorion vakinaiselle henkilökunnalle minulle osoitetusta kannustuksesta.

[1] Valitettavasti myös oppikirjoissa esiintyy huonoa koodia. Tämä on vakuuttava osoitus siitä, että hyvän ohjelmointityylin oppiminen ei ole helppoa!

2 Johdanto

• *Rajapinta*

Rajapinnalla (*interface*) tarkoitetaan minkä tahansa ohjelman osan (yleensä oleellisia ovat funktiot, makrot ja näiden joukot) kommunikointitapaa muun maailman kanssa. Esimerkiksi funktion `sqrt(x)` rajapinta määrittelee, että `sqrt()`:lle annetaan yksi `double`–tyyppinen argumentti ja se palauttaa yhden `double`n. Jos `x` on negatiivinen, se asettaa `errno`–muuttujan ja kutsuu `matherr()`:iä. Rajapinta ei määrittele funktion semanttista merkitystä. Se määrittelee vain millaista dataa sille on annettava, jotta se toimisi oikein ja mitä muutoksia se on tehnyt muun ohjelman tilaan[2], kun se päättyy (jos se päättyy).

• *Luettavuus*

Koodin *luettavuudella* (*code readability*) tarkoitetaan sitä, että kuka tahansa kohtalaisesti kieltä osaava koodia lukeva henkilö ymmärtää ohjelman toiminnan mahdollisimman pienellä vaivalla ja osaa mahdollisimman helposti löytää jotakin ohjelman toimintoa vastaavan kohdan ohjelmakoodista.

• *Modifioitavuus*

Koodin *modifioitavuudella*, muunneltavuudella, tarkoitetaan sitä, että ohjelman pieni muutos ei aiheuta yllättäviä ja vaikeasti selvitettäviä virhetoimintoja muualla.

• *Ylläpidettävyys*

Ohjelmakoodin *ylläpidettävyys* ($\approx$ luettavuus + modifioitavuus) on sen kaikkein tärkein ominaisuus. Näin siksi, että ylläpidettävää mutta toimimatonta koodia on mahdollista korjata. Toimiva mutta ylläpitokelvoton ohjelma on kirjoitettava alusta *kun* ohjelman määrittelyt muuttuvat. Tämä ei tarkoita sitä, että ylläpidettävä koodi olisi yleensä toimimatonta; kokemus on osoittanut, että ylläpidettävä koodi on yleensä virheettömämpää kuin ylläpitokelvoton koodi.

Nyt siis voidaan uskoa, että koodissa pitää pyrkiä ylläpidettävyyteen. Mutta kuinka kirjoitetaan ylläpidettävää koodia?

• *Modifioitavuuden parantaminen*

Modifioitavuutta pyritään parantamaan määrittelemällä rajapinnat selkeästi ja yksinkertaisesti. Lisäksi joskus liitetään koodiin tarkistuksia rajapintojen yli kulkevan datan oikeellisuudesta, esimerkiksi neliöjuuren laskeva funktio saattaa tarkistaa, että sen argumentti on ei–negatiivinen. Näin menetellään jos modifioitavuus katsotaan erityisen tärkeäksi, syöte tulee ohjelman käyttäjältä tai jos neliöjuuren laskemisessa ei ole erityisen kiire. Ohjelmointikielten tyyppijärjestelmillä pyritään suorittamaan testit jo käännösvaiheessa.

• *Luettavuuden parantaminen*

Luettavuuden parantaminen on monimutkaisempaa. Tärkeimmät, valitettavan ympäripyöreät, säännöt ovat:

1. Kirjoita *ymmärrettävää* koodia. Pyri yksinkertaisuuteen.
2. Kirjoita *havainnollista* koodia. Käyttämällä hyvää sisennystä, hyvää riveille jakoa jne.
3. Kirjoita *vähän muistamista*[3] *vaativaa* koodia. Käytä hyviä nimiä, kommentoimalla, ryhmittelemällä koodi oikein.

• *Kriittisyys*

Kaikkein tärkein tavoite on, että lukija oppii *suhtautumaan kriittisesti* muiden ja ennenkaikkea omaan koodiin. Tyytymättömyys omaan koodiin pakottaa ohjelmoijan hiomaan työtään ja täten sisäistämään oppaassa annetut hyvän ohjelmoinnin ohjeet. Juuri se, että oppaassa esitetään sekä hyvää että huonoa koodia toivottavasti auttaa lukijaa erottamaan ne toisistaan.

Lukujen lopussa on yleensä muutamia harjoitustehtäviä, jotka tulisi ainakin lukea! Vaikeat harjoitukset on merkitty tähdillä "$\star$". Kahden tähden harjoitus saattaa olla jopa satojen

[2] funktion paluuarvo, muut kuin funktion omat muuttujat ja tietorakenteet jne...

[3] ... ja yksinkertaistakin päättelyä ...

työtuntien suuruinen. Jotkut tehtävät, esimerkiksi 2.3, on tarkoitettu synnyttämään keskustelua eikä niihin välttämättä ole olemassa ainoaa oikeaa ratkaisua.

Harjoitus 2.1 Mikä on

1. funktion `acos`(3)
2. operaattorin `+`
3. funktion `random`(3)
4. funktion `sort`(3)

rajapinta ohjelmalle?

Harjoitus 2.2 Mikä on seuraavan koodinpätkän rajapinnan oltava, jotta se toimisi oikein? Miten muuttaisit allaolevaa koodia, jotta ohjelma, jossa tämä koodi on, olisi edes hieman modifioitavampi?

```
  ...
for (ctr = how_many_to_skip; ctr != 0; ctr--)
  next_char = getc(input_file);
```

Harjoitus 2.3 Laita tärkeysjärjestykseen allaolevat ohjelman hyvyyden kriteerit. Perustele!

- Toimivuus.
- Nopeus ja pieni muistinkäyttö.
- Ylläpidettävyys.
- Käyttäjäystävällisyys.
- Siirrettävyys.

3 Sisennys

Onko seuraava koodinpätkä sinusta luettavaa?

```
#include<stdio.h>
#include"priv.h"
int main(int argc, char*argv[]){int i;
for(i=1;i<=10;i++){
printf("Hello, world!(%d)\n",i);beep();}
}
```

Koodi on varsin sotkuista. Valitettavasti tällaista koodia esiintyy aina välillä opiskelijoiden harjoitustöissä, ja mikä pahinta, myös eräät oppikirjat käyttävät sotkuista sisennystä. Koodi näyttää heti paljon siistimmältä kun sitä väljennetään sopivasti ja käytetään sopivaa sisennystä (*indentation*), sillä sisennyksen avulla ohjelman rakenne tulee visuaalisesti selkeämmäksi.

```
#include <stdio.h>
#include "priv.h"

int main(int argc, char *argv[])
{
  int i;

  for (i = 1; i <= 10; i++) {
    printf("Hello, world!(%d)\n", i);
    beep();
  }
}
```

Yllä on käytetty kahden välilyönnin suuruista sisennystä, joka on useimpien ohjelmoijien mielestä luettavin. Myös muita sisennyksiä käytetään, esimerkiksi Kernighan ja Ritchie käyttävät neljän merkin syvyistä sisennystä. Olennaista kuitenkin on, että sisennystyyli on **aina sama**. • *Sisennys*

Välilyönnein voidaan luettavuutta parantaa edelleen. Tässä `for`:in perässä olevien sulkeiden ympärille ja niissä olevien lausekkeiden väliin on laitettu ylimääräiset välilyönnit. Niinikään `printf()`:in argumentteja erottavien pilkkujen jälkeen oleva välilyönti selkiyttää koodia. Lausekkeissa välilyöntejä voidaan käyttää selventämään lausekkeen rakennetta, esim. `c = 2*a + b`. • *Välilyönnit*

Myös `{`:n ja `}`:n paikasta voidaan kiistellä, mutta ilmeisesti yleisin tapa on laittaa ne kuten esimerkkikoodissa: `{` sitä edeltävän rakenteen jälkeen samalle riville ja `}` omalle rivilleen samalla sisennyksellä kuin `{`:iä edeltänyt rakenne. • *{ ja }*

Asiat on jaettava omille riveilleen. Yhdellä rivillä ei pidä yrittää sanoa liikaa. Yli 80 merkin rivejä ei tule kirjoittaa, sillä se on vieläkin yleisin päätteen ja tulostuksen leveys. Vain todella painavista syistä, esimerkiksi pitkien merkkijonovakioiden vuoksi, voi tästä ohjeesta luopua. Sitävastoin erittäin pitkät argumenttilistat ja lausekkeet voi varsin hyvin jakaa useammalle riville. `for`– ja `while`–silmukoiden sekä `if`–rakenteen toiminto–osat kannattaa sijoittaa omille riveilleen. Näin siinäkin tapauksessa, että toiminto on tyhjä! • *Jako riveille*

- *Tyhjät rivit*

Tyhjiä rivejä saatetaan käyttää mm. muuttujanmäärittelyosan erottamiseksi laskennan suorittavasta osasta, funktion eri osien erottamiseen jne. Funktioiden välillä on yleensä ainakin yksi tyhjä rivi, yleensä kaksi. Liiallinen määrä tyhjiä rivejä, etenkin funktioiden sisällä, on haitaksi — koodi näyttää olevan hajallaan.

Käytä sisennystä, välilyöntejä, hyvää riveille jakamista ja tyhjiä rivejä koodin rakenteen visualisoimiseksi

- *Tyylin yhdenmukaisuus*

Olennaista on myös se, että tyyli on aina sama — sekatyyli on hyvin ikävää luettavaa. Kovin persoonallisia visualisointimenetelmiä käytettäessä vaikutus saattaa olla harhaanjohtava. Tehdessäsi pieniä muutoksia toisten koodiin, käytä heidän tyyliään.

Harjoitus 3.1 Sisennä seuraava koodinpätkä. Selosta sen toiminta. **Naama peruslukemille, tämä ei ole hauskaa!**

```
/*
 * The next short recursive function does a MS-DOS--like match, '*'
 * matches any amount of any characters, including the empty string,
 * '?' matches exactly one character. Function returns 1, if match
 * found, 0 otherwise.
 */

int match(char*p,char*s){int help,chars_in_asterisk;switch(*p){case'?':
return(*s?match(p+1,s+1):0);case'*':chars_in_asterisk=0;help=0;while((s
[chars_in_asterisk]||!p[1])&&(help=match(p+1,s+chars_in_asterisk++))==0
);return help;case'\0':return !*s?1:0;default:return(*s==*p)?match(p+1,
s+1):0;}}
```

Harjoitus 3.2 Pohdi seuraavia asioita :

(1) Vertaile eri kielten luontaisia visualisointityylejä. Minne Pascalin `BEGIN` ja `END` sijoittuvat? Miksi jotkut kirjoittavat kaikki Pascalin varatut sanat isolla? Oletko samaa mieltä heidän kanssaan? Kuinka LISPiä tai Schemeä sisennetään. Minne sulkeet laitetaan? Entä FORTRAN?

(2) Kuinka eri editorit, erityisesti GNU Emacs, tukevat siistiä sisennystä?

(3) Tutki ohjelmia `cb(1)` ja `indent(1)`. Kokeile niiden toimintaa ja tutki kuinka hyvin ne onnistuvat tehtävässään.

Harjoitus 3.3 Vertaile seuraavia sisennystyylejä. Mitä hyvää ja huonoa niissä on? Onko jokin niistä selkeämpi kuin tekstissä esitetty? Lisäisikö jokin alla olevien tyylien piirteistä tekstissä olevan koodin selkeyttä? Perustele!

(1) (GNU–tyyli)

```
#include <stdio.h>
#include "priv.h"

int
main (int argc, char *argc[])
{
  int i;

  for (i = 1; i <= 10; i++)
    {
      printf ("Hello, world!(%d)\n", i);
      beep ();
    }
}
```

(2) (K&R–tyyli)

```
#include <stdio.h>
#include "priv.h"

int main(int argc, char *argv[])
{
    int i;

    for (i = 1; i <= 10; i++) {
        printf("Hello, world!(%d)\n", i);
        beep();
    }
}
```

(3) (Harvinaisempi paikka {}:lle...)

```
#include <stdio.h>
#include "priv.h"

int
main(int argc, char *argv[])
{
  int i;

  for (i=1; i<=10; i++)
  {
    printf("Hello, world!(%d)\n", i);
    beep();
  }
}
```

(4) (Liian harvaa...)

```
#include <stdio.h>
#include "priv.h"

int
main(int argc, char *argv[])
{
  int i;

  for ( i = 1; i <= 10; i++ ) {
    printf ( "Hello, world!(%d)\n", i ) ;
    beep ( ) ;
  }
}
```

4 Tunnukset

Seuraava funktio palauttaa merkkijonoon kirjoitetun kokonaisluvun arvon. Toisen argumentin osoittamaan paikkaan kirjoitetaan muunnettujen merkkien lukumäärä n, tai $-n$ jos merkkijono ei jatku numeromerkkien jälkeen.

```
int muunnos(char *string, int *length)
{
  int i = 0;

  if (string == NULL)
    fatal(74);
  *length = 0;
  while (string[*length] >= '0' && string[*length] < ':')
    i = 10 * i + string[(*length)++] - 48;
  if (string[*length] == 0)
    *length = -*length;
  return i;
}
```

• *Yksi kieli*

Ensimmäinen epämiellyttävä piirre on sekakielisyys. Sekakielisyys on ärsyttävää, sitä ei ymmärrä englantilainen eikä suomalainen. Sama sääntö pätee kommentteihin. Tässä oppaassa käytetään englantia, koska tällöin skandinaaviset aakkoset eivät tuota ongelmia.

• *Nimen kuvaavuus*

Funktion nimi `muunnos` on itsestäänselvä, eikö vain? Ei! Nimi ei kerro, mikä muunnos on kyseessä ja sopivassa kontekstissa, esimerkiksi Fourier–muunnosta laskevassa ohjelmassa se on harhaanjohtava. Parempi valinta olisi vaikkapa `string_to_int`.

• *Nimen ytimekkyys*

Entä onko argumentin `string` nimi hyvä? Kyllä, mutta se on tarpeettoman pitkä. Koska saman informaation näkee heti muualtakin, esim. `char *` –määrittelystä tai funktion tehtävän kuvauksesta, voi käyttää lyhyempääkin nimeä, vaikkapa vain `s`. Jos muuttujan nimen pituudella ei saavuteta mitään, voidaan varsin hyvin valita lyhyempi nimi.

Mitä laajemmalla alueella koodia muuttujaan viitataan, sitä kuvailevampi nimi on valittava. Varsinkin globaalien muuttujien tulisi olla erittäin hyvin nimettyjä.

Entä muuttuja `length`? Olisikohan `l` parempi? Vai kertoisiko `chars_converted` selkeämmin, että kyseinen luku (sen itseisarvo) on luettujen merkkien lukumäärä? Tämä on jo mielipidekysymys. Nimien valintaan ei voi antaa täsmällisiä ohjeita, vaan on turvauduttava omaan arvostelukykyynsä.

Muuttujaan `i` lasketaan lopputulosta, joten se voitaisiin nimetä uudelleen `res`:ksi, `result`:ksi, `val`:ksi, `value`:ksi ... Nimet kuten `i` ja `j` vihjaisevat, että kyseessä olisi kokonaisluku, jota käytetään indeksinä tai laskurina. Jotta päästäisiin eroon raskaan näköisestä `*length`:stä, korvataan se `i`:llä.

```
int string_to_int(char *s, int chars_converted)
{
  int result = 0, i = 0;

  if (s == NULL)
    fatal(74);
  while (s[i] >= '0' && s[i] < ':')
    result = 10 * result + s[i++] - 48;
  *chars_converted = (s[i] == 0) ? -i : i;
  return result;
}
```

Joskus, hyvin hyvin harvoin esiintyy todellista tarvetta erilaisten "flagien" käyttämiseen. Yleensä flagit kannattaa kuitenkin korvata muutoksella kontrollirakenteessa. Mutta silloin kun niitä tarvitaan, kannattaa nekin nimetä kuvaavasti, siis ei `flag1, flag2`, vaan `valid_input` tai `escape_read` jne.

Käytä kuvailevia ja ytimekkäitä tunnuksia

• Maagiset luvut makroiksi

Mutta emme ole vielä valmiit. Mitä ihmettä tarkoittavat `74`, `':'` ja `48`? Ilmeisesti `74` kertoo `fatal()`:lle mikä virheilmoitus pitää tulostaa. Mutta valitettavasti se ei kerro sitä suoraan ohjelman lukijalle. C:n makrot tarjoavat erään mekanismin tällaisten asiayhteydestään eksyneiden maagisten numeroiden (*magic numbers*) nimeämiseksi. Rivillä

```
#define ERROR_NULL_STRING 74
```

määritellään, että lukuun 74 voidaan viitata tunnuksella `ERROR_NULL_STRING`. Jos `fatal` on määritelty tiedostossa `errors.c`, voidaan `#define` ... laittaa tiedostoon `errors.h`, joka sitten luetaan `#include "errors.h"`:lla jokaiseen `fatal()`:ia käyttävään tiedostoon. Nyt `fatal()`:ia kutsuttaisiin `fatal(ERROR_NULL_STRING);`

Vielä parempi vaihtoehto on kirjoittaa `fatal()` uudestaan vapaamuotoisemmaksi siten, että se ottaisi argumenttinsa kuten `printf()`.

• Maagiset luvut luetelluiksi tyypeiksi

Makrojen lisäksi luetellut tyypit ovat hyvin käyttökelpoisia. Alla yleensä järkevän `boolean_t`–tyypin määritelmä. Etuna makroilla toteutettavaan versioon verrattuna on se, että debuggerit osaavat käsitellä lueteltujen tyyppien arvoja symbolisesti.

```
typedef enum { FALSE = 0, TRUE = 1 } boolean_t;
```

• Maagisten lukujen kommentointi

Joskus pelkkä kommentointi on riittävä, näin esimerkiksi silloin, kun vakio esiintyy vain yhdessä paikassa eikä sen kätevämpi muuttamismahdollisuus parantaisi ohjelman konfiguroitavuutta.

• Maagiset merkit

Mitä tarkoitetaan `while`:n ehdolla? Ascii–taulukoita tutkimalla saa selville, että ehdon toteuttavat merkit ovat numeromerkkejä. Jos ehdon kirjoittaisi muotoon `(s[i] >= '0' && s[i] <= '9')`, ei ascii–taulukoita tarvitsisi enää selata. Vielä parempi vaihtoehto on käyttää `ctype.h`:ssa määriteltyjä merkkiluokantarkistusmakroja, tässä tapauksessa makroa `isdigit()`. Entä `48`? Haa! Sehän on merkki `'0'`.

• *Erilaiset nollat*

Jotta merkit käsitettäisiin merkkeinä, kokonaisluvut kokonaislukuina ja osoitteet osoitteina, tulee merkkijonon lopettavaa nollamerkkiä verrata `'\0'`:aan, kokonaislukua `0`:aan ja osoitinta `NULL`:iin.

Vältä maagisia lukuja (ja merkkejä)

Vihdoinkin voimme kirjoittaa hyvän `string_to_int()`in:

```
#include <ctype.h>

int string_to_int(char *s, int chars_converted)
{
  int result = 0, i = 0;

  if (s == NULL)
    fatal("string_to_int : NULL argument.");
  while (isdigit(s[i]))
    result = 10 * result + s[i++] - '0';
  *chars_converted = (s[i] == '\0') ? -i : i;
  return result;
}
```

Jos halutaan, voidaan `*chars_converted` asettaa myös seuraavasti:

```
*chars_converted = s[i] ? i : -i;
```

• *yleisimmät tunnusten kirjoitustavat*

Tässä oppaassa käytetään C:ssä yleisesti käytössä olevaa tapaa kirjoittaa tunnukset kokonaan pienillä. Yhdyssanatunnusten osat erotetaan alaviivalla `_`. Toinen varsinkin muissa kielissä hyvin yleinen tapa on mm. SmallTalkin ja X11:n käytäntö kirjoittaa yhdyssanatunnukset yhteen niin, että yhdysosien ensimmäinen kirjain kirjoitetaan aina isolla, esimerkiksi `CharsConverted`. Makrot kirjoitetaan yleensä kokonaan isoilla kirjaimilla.

Harjoitus 4.1 Hyvin usein lyhyenkin nimen saattaa assosioida johonkin yleensä matematiikasta tuttuun tarkoitukseen. Oletko samaa mieltä allaolevan listan kanssa? Tuleeko mieleesi lisää vastaavia konventioita?

• *Konventiot*

- `i, j, k, l, m, n` ovat kaikki kokonaislukuja. `i` ja `j` ovat pääsääntöisesti taulukoiden (esim. matriisien) indeksointia varten. `m` ja `n` sisältävät lukumääriä.
- `x, y, z` ovat koordinaatteja, joko reaali– tai kokonaislukuja. `x` saattaa tarkoittaa jotain tuntematonta, mutta kyseisessä koodin kohdassa ratkaistavaa muuttujaa, vrt. matematiikan tehtävien `x`.
- `r, t, u, v, w` haiskahtavat reaaliluvuilta.
- Suuret yksikirjaimiset muuttujat ovat matemaattisessa koodissa matriiseja tai vektoreita.
- `s, str` ovat merkkijonoja. Samoin esim. `s1, s2`.
- `c, ch` ovat merkkejä.
- `f, fd` ja `fp` ovat tiedostoja, `fd` kokonaislukutyyppinen *file descriptor* ja `fp` tyyppiä `FILE *`.
- `p, q, ptr` jne. ovat osoittimia (*pointer*).
- `f, g` ovat funktio–osoittimia.
- `ctr, cnt, cntr` ja niiden muunnokset ovat laskureita, esim. jos halutaan laskea taulukon negatiivisten alkioiden lukumäärä, `i` olisi indeksi taulukkoon ja `ctr` toistaiseksi löydettyjen negatiivisten elementtien lukumäärä.
- `tmp, temp` ovat tilapäismuuttujia. Esimerkiksi `swap(i, j)` on `tmp = i; i = j; j = tmp;`
- `dummy, foo, bar, foobar` jne. ovat tarpeettomia, mutta teknisistä syistä mukana olevia muuttujia. Esimerkiksi `i = string_to_int(s, &dummy)` jos `dummy`n arvolla ei ole merkitystä.
- `argc, argv` ovat `main`in argumentit.
- Pääte `_t` kertoo kyseessä olevan tyypin nimen, esimerkiksi `complex_t`. Tämä käytäntö näyttää olevan yleistymässä.

Ylläoleva lista antaa viitettä siitä, millaisia nimiä kannattaa käyttää, jos pitkän nimen käyttäminen ei lisää koodin ymmärrettävyyttä. **Tällaisten typistettyjen nimien käyttö suurella alueella näkyvinä muuttujina**, esim. globaaleina **ei ole järkevää**. Miksei?

5 Yksinkertaista!

5.1 Sievennä monimutkaisia lausekkeita

Alla oleva koodi on muunnos kirjan **Kernighan & Plauger**, *"The Elements of Programming Style"* ensimmäisestä esimerkistä.

```
for (i = 1; i <= matrix_size; i++)
  for (j = 1; j <= matrix_size; j++)
    I[i-1][j-1] = (i/j) * (j/i);
```

Koska kokonaislukujen jakolasku suoritetaan tässä tapauksessa katkaisevana, on `i/j > 0` vain jos `i` > `j`. Siis jos `i=j`, on `(i/j) * (j/i)`:n arvo 1, muulloin se on 0, joten `I`:hin sijoitetaan yksikkömatriisi. Kerrassaan nerokasta, eikö vain!

• *Selkeys*

Kerrassaan vastenmielistä! Ohjelmat tulee tehdä helppolukuisiksi, Mensan testit ovat erikseen. Ylläoleva koodi on niin kieroutunutta, etteivät edes indeksit `i` ja `j` voi kulkea luonnollisesti nollasta `matrix_size-1`:n, vaan nollalla jakamisen estämiseksi indeksejä on jouduttu nostamaan yhdellä.

Sano sanottavasi selkeästi, älä viisastele

```
for (i = 0; i < matrix_size; i++)
  for (j = 0; j < matrix_size; j++)
    I[i][j] = (i == j);
```

Jos kirjoitetaan koodia C:tä tuntemattomille lukijoille tai kirjoittaja itse ei ole varma vertailulausekkeiden aritmeettisesta arvoista, kannattaisi luultavasti kirjoittaa

```
    I[i][j] = (i == j) ? 1 : 0;
```

Toinen esimerkki:

```
for (i = 0; !((i >= m) | (y <= 0)); i++)
  y += *(Deltax+i) * *(Dy+i) / r / (k << 1);
```

• *Muistiviittauksien syntaksi*

Vaikka `*(s+i)` ja `s[i]` ovat merkitykseltään samoja, kannattaa jälkimmäistä käyttää selvyyden vuoksi aina kun viitataan taulukkoon. Saman periaatteen mukaisesti struktuurin `p` kenttään `f` ei tule viitata `(*p).f` vaan `p->f` .

• *Matematiikan ja logiikan säännöt*

Käytä sulkeita laskujärjestyksen painottamiseksi. Valitse matemaattisen kaavan esitysmuodoksi selkein mahdollinen. Myös logiikan sääntöjä voidaan käyttää lausekkeiden siistimiseksi, erityisesti de Morganin säännöt ovat käyttökelpoisia.

• *Bittioperaatiot vain bittejä varten*

Älä käytä loogisissa lausekkeissa bittioperaatioita `&&`:n tai `||`:n asemesta. Useimmat kääntäjät osaavat optimoida kerto– ja jakolaskun kokonaislukuvakioiden kanssa bittioperaatioiksi automaattisesti, joten niitä ei pidä käyttää luonnollisimman esitystavan sijasta.

```
for (i = 0; i < m && y > 0; i++)
  y += Deltax[i] * Dy[i] / (2 * k * r);
```

Harjoitus 5.1 Siisti seuraavia lausekkeita, jos katsot niiden olevan siistimisen tarpeessa. Voit myös muuntaa niitä rakenteiksi.

- `x = ((a != b) != (c != d)) ? y : x;`
- Alla tiedetään, että $\forall i \in \{0 \ldots \texttt{MEASUREMENTS} - 1\} : a_i > 0$.
  ```
  #define MEASUREMENTS 5
    ...
  while (!(1 == a[0] == a[1] == a[2] == a[3] == a[4])) {
    smooth(a, MEASUREMENTS);
    smoothings_done++;
  }
  ```
- ```
  char *s = "3210";
  x = s[0] - *(s+1) + *(2+s) - 3[s];
  ```
- `return '-'-'-'-*"";`

Harjoitus 5.2 Perehdy C:n operaattorien presedenssi– ja assosiatiivisuussääntöihin. Perustele miksi ne ovat sellaiset kuin ne ovat. Miksi `&` ja `&&` sekä `|` ja `||` on eriytetty? (C:n edeltäjässä, B:ssä, `&&`:ä ja `||`:ä ei vielä ollut.)

5.2 Valitse sopiva rakenne

Allaoleva funktio suorittaa kahden merkkijonon vertailun aivan kuten `strcmp()`, mutta katkaisevasti. Esimerkiksi "foo" ja "foobar" ovat samanarvoiset ja `strccmp()` palauttaa arvon 0.

```
int strccmp(char *s1, char *s2)
{
  if (*s1 && *s2)
    do {
      if (*s1 == *s2)
        continue;
      return 2 * (*s1 > *s2) - 1;
    } while (*++s1 && *++s2);
  return 0;
}
```

Ensinnä havaitaan, että jommankumman merkkijonon loppumisen vertailu `*s1 && *s2` esiintyy kahdesti. Ensimmäisen kerran se pitää toteuttaa ennen, kuin päästään silmukkaan ja silmukassa ollaan kunnes ehto ei enää toteudu. Mutta saman asian ajaa suoraan `while`–silmukka, kunhan `s1`:n ja `s2`:n inkrementointi siirretään `while`n sisään. `continue` on täysin tarpeeton, ja siitä päästään eroon negatoimalla ehto. Myös palautettavan arvon lauseketta voi siistiä huomattavan paljon.
```

```
int strccmp(char *s1, char *s2)
{
  while (*s1 && *s2) {
    if (*s1 != *s2)
      return (*s1 < *s2) ? -1 : 1;
    s1++;
    s2++;
  }
  return 0;
}
```

• *Rakenteet*

Silmukoita varten C:ssä on `while`, `do ... while` ja `for` –rakenteet. Ehtoja varten on `if` ja `case`. Usein on näppärää käyttää C:n ehdollisia lausekkeita (`? :`), jos muuten jouduttaisiin käyttämään apumuuttujia tai jos toiminnot ovat lyhyitä, ja niiden toisistaan poikkeavat osat ovat funktionaalisia.

• *break, continue, goto ja flagit*

Liian innokas `break`in (poikkeuksena `case`) ja varsinkin `continue`n käyttö voi haitata koodin luettavuutta. Liiallista `goto`n käyttöä tulee välttää. Sen käyttö on suositeltavaa vain poikkeustapauksissa esimerkiksi toiminnan uudelleenkäynnistämiseksi tai virheenkäsittelyssä, jossa normaalisuoritusta vastaava koodi tulisi muuten epäselväksi. Käyttämällä flagejä, ohjelman kontrolliin vaikuttavia (yleensä kokonaislukutyyppisiä) muuttujia, voidaan lukija johtaa helposti harhaan. Flagit saattavat joskus olla tarpeellisia, mutta hillitön flagien ja `goto`n käyttö kertoo, että ohjelman tekijällä ei ole ollut kunnollista käsitystä ongelmasta ja koodista eikä hän ole suunnitellut ratkaisua ennen sen toteuttamista.

Valitse tarkoituksenmukainen rakenne

• *Pura liiat sisäkkäiset rakenteet*

Vaikka rakenteita käytettäisiinkin oikein, eivät ne kuitenkaan saa olla liian monimutkaisia. Ihmisten kykenemättömyys pitää mielessä paljon keskeneräisiä ajatuksia rajoittaa sisäkkäisten rakenteiden määrää. Yleensä kolme sisäkkäistä rakennetta ymmärtää vielä kohtuullisella vaivalla, mutta viisi sisäkkäistä rakennetta tekee koodista lukukelvotonta[4]. Tällaisissa tapauksissa on sisimpiä rakenteita siirrettävä funktioihin tai makroihin[5]. Myös rekursiivinen ajattelu saattaa yksinkertaistaa ongelmaa.

• *Rakenteiden kommentointi*

Kontrollirakenteita voi myös kommentoida ja ainakin kommentit `/*FALLTHROUGH*/` ja `/*NOTREACHED*/` on todettu hyödyllisiksi. Myös `lint` osaa hyödyntää näistä jälkimmäistä. Seuraavana on ote vielä työn alla olevan ohjelman argumenttien lukemiskohdasta, jossa optiota `-s` ei vielä ole toteutettu, mutta siihen on varauduttu. `fatal()` ei palaa, vaan lopettaa ohjelman suorituksen virheilmoituksen tulostamisen jälkeen `exit(1)`:llä.

[4] Kaikki rakenteet eivät ole samanarvoisia tässä mielessä. Esimerkiksi `case`n voi ymmärtää helpommin kuin `for`–silmukan.

[5] Kaikkien funktioiden ja makrojen tehtävien tulee tietysti olla yksinkertaisesti määriteltävissä, jotta tällä toimenpiteellä olisi mitään mieltä. Ks. seuraava luku.

```
char *passwd_file = "/etc/passwd";
int normal_passwd_format = 1; /* YP or normal format */
int c; /* Is int, but used like a char in switch */

while ((c = getopt(argc, argv, "ysf:")) != EOF)
  switch (c) {
  case 'y':
    normal_passwd_format = 0;
    break;
  case 'f':
    passwd_file = optarg;
    break;
  case 's':
    /* I'll hack it real soon now... */
    /*FALLTHROUGH*/
  default:
    fatal("Usage: %s -[ys] [-f filename]\n", argv[0]);
    /*NOTREACHED*/
    break;
  }
```

Harjoitus 5.3 Mitä kommentteja `lint` ymmärtää `/*NOTREACHED*/`:in lisäksi?

Harjoitus 5.4 Miksi optionlukuesimerkissä on `fatal()`–kutsunkin jälkeen järkevää laittaa `break`?

Harjoitus 5.5 Kirjoita ohjelma `detab`, joka korvaa `stdin`istä tulevien merkkijonojen kaikki tabulaattorimerkit `'\t'` vastaavalla määrällä välilyöntejä.

⋆ **Harjoitus 5.6** Tee ohjelma `margins`, joka lukee `stdin`istä tekstiä, välilyönnein erotettuja sanoja ja suorittaaa *word wrap*–tyylisen tasauksen niille sanoille, jotka menevät reunan yli. Vasemman ja oikean marginaalin etäisyys vasemmasta reunasta annetaan ohjelmalle komentoriviargumenttina.

⋆⋆ **Harjoitus 5.7** Taitavista matemaatikoistaan tunnetun Mathsvillen pormestari oli kutsunut Peter Productin ja Sam Sumin puheilleen. Pormestari sanoi ajattelevansa kahta lukua väliltä 2..100. Hän kertoi Peter Productille näiden kahden luvun tulon ja Sam Sumille niiden summan. Peter Productin ja Sam Sumin keskustelu kulki seuraavasti:

(1) En tiedä mitä lukuja pormestari ajatteli, sanoi Sam Sum.
(2) En minäkään, sanoi Peter Product.
(3) Sen minä tiesin.
(4) Hei, nyt tiedän mitä lukuja pormestari ajatteli!
(5) Nyt tiedän minäkin!

Tee ohjelma, joka repliikki kerrallaan karsii mielestään ne lukuparit, jotka eivät enää tule kyseeseen. Esimerkiksi toisesta repliikistä voidaan päätellä, että molemmat pormestarin ajattelemat luvut eivät ole alkulukuja. Keskity ohjelmassasi mahdollisimman suureen yksinkertaisuuteen, jotta et sekoaisi ajatuksissasi!

5.3 Muita keinoja yksinkertaistaa

• *Käytä valmiita rutiineja*

Toinen, yleensä vielä parempi mahdollisuus `strccmp()`:n toteuttamiseksi on valmiiden rutiinien käyttäminen. Muista, että myös `string_to_int()` yksinkertaistui tällä tavalla!

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))

int strccmp(char *s1, char *s2)
{
  return strncmp(s1, s2, MIN(strlen(s1), strlen(s2)));
}
```

$\boxed{\text{Käytä valmiita rutiineja}}$

• *Abstrahoi toistuvat rakenteet*

Yritä löytää helposti ymmärrettävä abstraktio useasti toistuville kontrollirakenteille. Luvussa 6 kuvattut funktio `xfopen()` ja makro `VERIFY()` ovat tällaisia, samoin esimerkiksi Common Lispin `map`–funktiot tai yleiskäyttöiset sekvenssifunktiot `every`, `some`, `notany` ja `notevery`.

• *Koodia dataksi*

Datataulukoilla säästää vaivaa ja lisää ohjelman konfiguroitavuutta. Esimerkiksi kuukauden päivän lukumäärää ei kannata kirjoittaa monimutkaiseen ehtoon vaan kokonaislukutaulukkoon. Lisäesimerkkejä on sivujen 31 ja 32 koodissa.

• *Funktionaalinen tyyli*

Funktionaalinen ohjelmointityyli on kaunista C:ssäkin.

```
c = get_char_from_tty(TRUE);
switch (c) {
case 'y': case 'Y':
  ...
```

Kannattaa poistaa täysin tarpeettomat muuttujat:

```
switch (get_char_from_tty(TRUE)) {
case 'y': case 'Y':
  ...
```

Harjoitus 5.8 Selaa UNIXin manuaalia (osia 2 ja 3) ja vastaa seuraaviin kysymyksiin

- Mitä matemaattisia funktioita sinulle tarjotaan?
- Kuinka suoritetaan säännöllisen lausekkeen (*regular expression*) etsiminen syötteestä? Kirjoita ohjelma, joka etsii komentoriviargumenttina annettuun säännölliseen lausekkeeseen sopivia sanoja tiedostosta `/usr/dict/words`.
- Mitä muistin ja merkkijonojen käsittelyfunktioita löydät?

6 Funktiot

6.1 Hyvän funktion tunnusmerkkejä

Älä kirjoita suuria funktioita

• *Kirjoita pieniä funktioita*

Ilman painavaa syytä, esimerkiksi suurta `switch`–rakennetta, tulisi funktion olla korkeintaan 20-50 riviä pitkä. Tätä ohjetta voi rikkoa, mutta yleensä vain todella taitavissa käsissä tämän säännön rikkominen ei johda kaaokseen. Aloittelevan ohjelmoijan (alle 50000 – 100000 riviä monipuolisia ohjelmointitehtäviä...) kannattaa ottaa tämä sääntö vakavasti. Lyhyiden funktioiden kirjoittamista ei pidä vältellä tehokkuussyistä, sillä nykyaikaiset kääntäjät osaavat tarvittaessa *inline*–laajentaa funktiokutsuja.

Alla muutamia esimerkkejä epäonnistuneista funktioista:

- `void handle_file(char *file_name, FILE **f, int action);`

  Jos `action`in arvo on

  0 avataan tiedosto `file_name` lukemista varten, jolloin tiedosto–osoitin talletetaan `*f`:n.

  1 suljetaan tiedosto `*f`.

  2 Kelataan `*f` alkuun.

  Kaikki toimenpiteet tarkistavat toiminnan onnistumisen.

- `void convert_celsius(float c, float *fahrenheit, float *kelvin);`

  Konvertoi annetun Celsiusasteen Fahrenheitiksi ja Kelviniksi ja tallentaa ne paikkoihin `*fahrenheit` ja `*kelvin`.

• *Yksinkertaiset rajapinnat*

Funktioiden rajapinnat kannattaa pitää yksinkertaisina. `handle_file()` on huono esimerkki rajapinnasta, sillä esimerkiksi `*f`:stä voidaan joko lukea tai siihen voidaan kirjoittaa `action`:ista riippuen.

Esimerkkifunktioiden semantiikka on varsin hankalasti määriteltävissä. Ensimmäisessä tapauksessa rutiini olisi pitänyt jakaa kolmeen osaan, yksi kutakin `action`in arvoa kohti. Jälkimmäisessä tapauksessa jako kahteen rutiinin olisi ollut paikallaan.

Hyvän funktion semantiikka voidaan kuvata täydellisesti muutamalla (1–3) yksinkertaisella virkkeellä

• *Semantiikan selkeys*

"Täydellinen" kuvaus on varsin hutera käsite. Se tarkoittaa käytännössä sitä, että kuvataan lähes kaikki rutiinissa oleva koodi olettaen rutiinin käyttämien alirutiinien jo olevan dokumentoituja. Esimerkiksi alla kuvatun rutiinin `xfopen()` kuvauksessa ei selitetä sen kutsumien alirutiinien `fopen()` (ja edelleen `open()`) ja `fatal()` toimintoja, mutta selitetään mitä niiden käytöllä on saatu aikaan.

- `MAX(a, b)`

  Makro palauttaa a:sta ja b:stä suuremman, jonka se myös evaluoi kahdesti.

- `FILE *xfopen(char *file_name, char *mode);`

  Avaa tiedoston `file_name` moodilla `mode` ja tarkistaa avauksen onnistumisen. `mode` kuten `fopen()`illa.

- `int read_token(FILE *fp, char *token_array[], int tokens);`

  Lukee merkkejä tiedostosta `fp` ja jos merkit muodostavat jonkun sanan `token_array[i]` , palautetaan `i` $\in \{0 \ldots$`tokens`$-1\}$. Muutoin palautetaan -1.

- `void *median(void *array[], int min_ind, int max_ind, int (*compare)(void *, void *));`

  Etsii taulukosta `array[min_ind..max_ind]` mediaanin, kun kahden elementin vertailu tapahtuu funktiolla `compare`.

• *Yleisyys*

Funktiot kannattaa kirjoittaa yleispäteviksi (kuten `median()` yllä). Yleispätevyydellä ei kuitenkaan tarkoiteta monen toiminnan yhdistämistä, vaan saman algoritmin soveltamista eri tyypeille, erikokoisille taulukoille jne.

• *Kommunikointi*

Funktion ja kutsujan paras kommunikaatiomekanismi on funktion argumentit ja paluuarvo. Joskus argumentit välitetään funktiolle osoitteina varsinaiseen arvoon, jolloin funktion oletetaan muuttavan k.o. arvoja (esim. `string_to_int()`:n toinen argumentti). Tähän turvaudutaan yleensä siinä tapauksessa, että funktion halutaan palauttavan useampi kuin yksi tieto.

• *Funktiojoukon yhdenmukaisuus*

Jos olet kirjoittanut useita samankaltaisia funktioita, kannattaa argumentit järjestää mahdollisimman samalla tavoin kaikissa funktioissa. Esimerkiksi X11:n funktioissa ensimmäinen argumentti on yleensä `Display`.

Harjoitus 6.1 Mitä yhdenmukaisuutta havaitset funktioissa `mem`*xxx*(3) ja `str`*xxx*(3)?

6.2 Makrot

Yleensä makroja kannattaa käyttää silloin, kun

• *Milloin makroja?*

- makrolla voidaan piilottaa jokin tyylillisesti epämiellyttävä piirre, esimerkiksi maaginen luku tai monimutkainen syntaktinen rakenne.
- makrolla peitetään käyttöjärjestelmästä tai kääntäjästä riippuvia palveluja tai jos makrolla voidaan lisätä ohjelman konfiguroitavuutta tai siirrettävyyttä. Esimerkiksi

```
#define LEXICON_CACHE_SIZE 64000 /* bytes */

#ifdef SYSV
#define index(s, c)      strchr(s, c)
extern char *strchr(char *s, int c);
#else
extern char *index(char *s, char c);
#endif /* SYSV */
```

- makroa vastaava koodi on hyvin pieni.
- makroa kutsutaan sivuvaikutuksettomilla argumenteilla.
- tarvitaan maksimaalista tehokkuutta eikä voida olettaa kääntäjän osaavan *inline*–laajentaa funktioita.

Makrot pitää kirjoittaa erittäin huolellisesti, koska C:ssä makrot laajennetaan tekstuaalisesti. Esimerkkejä huonosti kirjoitetuista makroista:

• *Tekstuaalinen laajentaminen*

```
#define MAX(a, b) a>b ? a : b
#define VERIFY(condition) \
  if (!condition) \
    goto error
```

Jos allaolevassa ohjelmassa käytetään näitä makroja, on tulos täysin intuition vastainen. Oletetaan, että `a` on 2 ja `b` on 3.

```
    ...
    c = 2*MAX(a, b) + 1;
    if (c < b)
      VERIFY(c < a);
    else
      printf("foo\n");
    return;

    error:
      fatal(...);
      /*NOTREACHED*/
  }
```

Erkki Luuseri saattaa jo kello 4 aamuyöllä tajuta, miksi ohjelma ei koskaan tulosta ”foo” vaan kutsuu `fatal()`:ia!

• *Sulkeita makroihin*

Lopulta hän tajuaa lisätä tarpeellisen määrän erilaisia sulkeita. Lisäksi kannattaa antaa virhetilanteessa hypättävän paikan nimi `VERIFY()`:ille, jolloin siitä tulee hieman yleiskäyttöisempi.

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define VERIFY(condition, error) \
  do { \
    if (!(condition)) \
      goto error; \
  } while (FALSE)
```

Harjoitus 6.2 Selvitä miksi ohjelma ei toiminut kuten voisi olettaa. Lavenna makrot koodiin ja lisää tarpeellinen määrä sulkeita nähdäksesi miksi makrot eivät toimineet. Mikä on `c`:n arvo ja mikä sen pitäisi olla? Mitä ohjelma tekee, jos vain jompikumpi makroista `MAX()` ja `VERIFY()` on määritelty hyvin?

Harjoitus 6.3 Miksi `VERIFY()`:iä ei määritelty seuraavasti?

```
#define VERIFY(condition, error) { \
  if (!(condition)) \
    goto error;\
}
```

Harjoitus 6.4 Makrojen nimet kirjoitetaan yleensä kokonaan isoilla kirjaimilla. Milloin tästä käytännöstä voi poiketa, miksi esimerkiksi `index()` on kuitenkin kirjoitettu pienillä kirjaimilla?

7 Käyttöjärjestelmän palvelut

7.1 system() vs. systeemikutsut

Seuraavan koodin tarkoituksena on poistaa tiedosto `filename`.

```
char command[...];
 ...
sprintf(command, "rm %s", filename);
system(command);
```

Mitä tapahtuu, kun C–ohjelmassa sanotaan `system("rm foo")`? C–ohjelma käynnistä `sh`:n, joka käynnistää toisen ohjelman `rm`, joka lukee argumenttinsa ja tässä tapauksessa suorittaa systeemikutsun `unlink("foo")`. Mikään ei estä C–ohjelmaa itse suorittamasta kyseistä systeemikutsua. On valitettavaa, että monille systeemikutsujen olemassaolo näyttää olevan täysin tuntematon asia.

• `system()`*:in heikkoudet*

`system()`:in käytössä on laskennan tarpeettoman raskauden lisäksi muitakin ongelmia. esimerkiksi se ei välttämättä käytä Bourne–shelliä, vaan käynnistää mielummin `SHELL` –ympäristömuuttujassa määritellyn komentotulkin. Myös käskyjä suoritettaessa tapahtuneiden virheiden havaitseminen on yleensä hankalaa. Älä käytä `system()`:iä, jos se on kohtuullisella vaivalla vältettävissä.

Vältä `system()`:iä, käytä systeemikutsuja

```
unlink(filename);
```

Mutta ratkaisu ei ole vielä täydellinen. Hyvä sääntö on, että kaikkien virhetilanteissa palaavien funktioiden virhetilanteet pitää havaita, varsinkin systeemifunktioiden virheet.

• *Tarkista*

Tarkista!

```
if (unlink(filename))
  warning("Could not delete file '%s'.\n", filename);
```

Ohjelmoija on tässä katsonut järkeväksi ilmoittaa virhetoiminnasta käyttäjälle, mutta toivoo, että virhe ei ole kovin hankala, vaan ohjelman suoritusta voidaan vielä ainakin yrittää jatkaa.

Harjoitus 7.1 Mikä vaikutus on kutsulla `system("cd $HOME");`?

Harjoitus 7.2 Eräässä ohjelmassa hakemiston sisältö luettiin kutsumalla `system("ls -1A > tmp")`, jonka jälkeen tiedoston `tmp` sisältö luettiin rivi kerrallaan. Lopuksi `tmp` poistettiin `system("rm tmp")`:lla. Mitä kaikkea on saattanut epäonnistua? Miten ohjelma voisi tietää epäonnistumiset? Kuinka hakemisto luetaan "oikein"? Vihje: `man 3 readdir`

7.2 Yleisimpiä systeemikutsut

• *Yleisimpiä systeemikutsuja*

Johdatuksena aloittelevalle UNIX–ohjelmoijalle, jolle manuaalin salat eivät vielä ole auenneet, on alla listattu yleisimpiä tarpeita ja niitä vastaavat man–entryt.

Tiedostojen lukeminen ja kirjoittaminen: File descriptoreilla `open`(2), `close`(2), `read`(2), `write`(2). Vastaavasti `FILE *`:lla `fopen`(3), `fclose`(3) jne. `FILE *`:ia vastaava file descriptor saadaan makrolla `fileno`(3), file descriptorista voidaan rakentaa `FILE *` funktiolla `fdopen`(3).

Tiedostojen luominen, tuhoaminen, siirtäminen, linkit: `creat`(2) (hakemistoille `mkdir`(2)), `unlink`(2) (`rmdir`(2)), `rename`(2), `symlink`(2).

Tiedostojen ja hakemistojen tutkiminen: `stat`(2), `lstat`(2), `readdir`(3).
Katso myös mm. `umask`(2), `chmod`(2), `chown`(2), `ioctl`(2), `fcntl`(2) jne. UNIX–prosessin on harvoin järkevää vaihtaa kotihakemistoaan, mieti miksi!

Prosessien luominen: `fork`(2), `execve`(2), `wait`(2), BSD:ssä kannattaa käyttää `vfork`(2), `wait3`(2).

Hyödyllisiä ovat myös `dup`(2), `pipe`(2). Käytöltään mukavampia voivat olla `popen`(3) ja `pclose`(3).

Prosessien välinen kommunikaatio: `socket`(2), `bind`(2), `listen`(2), `accept`(2), `connect`(2), `select`(2).

Signaalit: `signal`(3). Katso myös `longjmp`(3).

Pääteriippumaton päätteen hallinta: `curses`(3), joskus harvoin `termcap`(3).

Aika: `sleep`(2), `usleep`(2); `ctime`(3) ja `time(3)`.

Ympäristömuuttujien hallinta: `getenv`(3), `putenv`(3) tai `setenv`(3).

Lukitukset: `lockf`(2).

...ja paljon muuta. Oleellista on, että tunnetaan systeemikutsujen kattavuus ja että ohjelmoijalla on rohkeutta (ja kärsivällisyyttä) etsiä ongelmaann ratkaisua manuaaleista (käytä introja ja SEE ALSO–kappaleita), alan kirjallisuudesta ja tarvittaessa myös kokeneemmilta ohjelmoijilta.

Harjoitus 7.3 Lue UNIXin manuaalien osien 2 ja 3 introt, esim. `man 2 intro`

Harjoitus 7.4 Kirjoita funktiot `fatal()` ja `warning()`. Molemmat vastaavat `fprintf()`:ää `stderr`iin, mutta `fatal()` ei palaa. Katso `varargs(3)` ja `vsprintf(3)`[6].

⋆ **Harjoitus 7.5** Tee ohjelma, jolla voit muodostaa kehämäisen putken. Ts. jos ohjelmalle annetaan kolme argumenttia A, B ja C, ohjelma käynnistää kyseiset ohjelmat A, B ja C siten, että A:n `stdout` menee B:n `stdin`iin, B:n `stdout` menee C:n `stdin`iin ja C:n `stdout` menee takaisin A:n `stdin`iin. Ohjelman kutsu saattaisi olla vaikkapa

```
$ circulate 'mychessprogram -white' 'yourchessprogram -black'
```

⋆⋆ **Harjoitus 7.6** Tee UNIXin `talk`kia vastaava ohjelma. Käytä `curses`–kirjastoa näytön hallintaan ja `socket`teja kommunikointiin. Saat yksinkertaistaa varsin paljon UNIXin talkista.

[6] Joissakin ympäristöissä `_doprnt(3)`

8 Muistin käsittely

Tässä luvussa (ja vielä seuraavassakin) parannetaan eri tavoin seuraavaa esimerkkiä:

Väärin

```
boolean_t confirm(question)
char *question;
{
  char *answer = (char *) malloc(10 * sizeof(char));

  printf("%s (y/n + enter) :", question);
  gets(answer);
  return answer[0] == 'y' || answer[0] == 'Y';
}
```

8.1 `malloc()`, `calloc()`, `realloc()`, `free()`

Kuten aiemmin todettu, tulee virhetilanteissa palaavien funktioiden virheet tarkistaa. `malloc()` palauttaa `NULL`in, jos muisti on loppunut. Jos tarkistusta ei tehdä, `confirm()` yrittää kirjoittaa `NULL`in osoittamaan paikkaan, mikä ei järkevissä käyttöjärjestelmissä ole mahdollista.

• *Tarkista*

Funktio ei koskaan vapauta varaamaansa muistia ja koska `answer`ia ei sellaisenaan palauteta minnekään, ei ole myöskään oletettavissa, että joku toinen rutiini vapauttaisi muistin `confirm()`:in puolesta. Ohjelma siis "vuotaa" muistia ja saattaa siksi olla käyttökelvoton.

• *Vapauta*

Vapauta varaamasi muisti

Lisäksi ohjelma voi kaatua myös siinä tapauksessa, että sille syöttää liian pitkän merkkijonon. Periaatteessa `answer`ille varattavan tilan kasvattaminen suuremmaksi ei poista tätä ongelmaa. Eräs mahdollisuus olisi lukea merkkijono `fgets()`:illä, jolle voidaan antaa argumentiksi luettavan merkkijonon maksimipituus. Ratkaisu on edelleen erittäin käyttäjäepäystävällinen.

• *Älä viittaa varatun muistin ulkopuolelle*

```
boolean_t confirm(char *question)
{
  char *answer;
  boolean_t result;

  if ((answer = (char *) malloc(10)) == NULL)
    fatal("confirm: Out of memory");
  printf("%s (y/n + enter) :", question);
  fgets(answer, 10, stdin);
  result = answer[0] == 'y' || answer[0] == 'Y';
  free(answer);
  return result;
}
```

Missään tapauksessa ei saa viitata jo vapautettuun muistiin, siksi yllä tarvitaan `result`–muuttujaa. Vapautettuun tai varaamattomaan muistiin viittaavaa koodia voi olla hyvin

• *Älä viittaa vapautettuun muistiin*

vaikea havaita testaamalla. Siksi tähän seikkaan on kiinnitettävä erityisen suurta huomiota jo koodia kirjoitettaessa.

Älä koskaan viittaa varaamattomaan tai vapautettuun muistiin!

Koska C:ssä `sizeof(char)` on määritelmän mukaan 1, se voitiin jättää ylläolevasta koodista pois.

Harjoitus 8.1 Mitä hyötyä roskankeruusta (*garbage collection*) on? Miksi sitä ei ole C:ssä ja miksi sen hyvä toteutus C:hen on vaikeaa?

Harjoitus 8.2 Kuinka varataan ja vapautetaan kolmiulotteinen taulukko `field[0..XSIZE-1][0..YSIZE-1][0..ZSIZE-1]` `float`teja? Kuinka paljon taulukko kuluttaa muistia? Miksi C:ssä ei ole Pascalista tuttua mahdollisuutta indeksoida moniulotteisia taulukkoja tyyliin `field[x,y,z]` vaan pitää kirjoittaa `field[x][y][z]`?

$\star$ **Harjoitus 8.3** Tee testiohjelmia, joilla tutkit `malloc()`:in, `realloc()`:in ja `free()`:n tehokkuutta erikokoisilla alkioilla, satunnaisessa järjestyksessä käytettynä. Mikä on nopein, miksi?

8.2 Pinon käyttö

• *Pino*

Pinosta on erittäin kätevä varata muistia. C:n funktioiden lokaalien ei–staattisten muuttujien tarvitsema tila varataan pinosta.

```
boolean_t confirm(char *question)
{
  char answer[10];

  printf("%s (y/n + enter) :", question);
  fgets(answer, 10, stdin);
  return answer[0] == 'y' || answer[0] == 'Y';
}
```

• *Vain lokaaliin käyttöön*

Kun funktion suoritus päättyy, vapautuu `answer`ille pinosta varattu tila automaattisesti. Siksi on ehdottoman tärkeää, että **pinosta varattuun dataan ei saa viitata kutsun loputtua**. Ts. seuraava funktio on virheellinen:

Väärin

```
char *alloca_10_chars(void)
{
  char chars[10];

  return chars;
}
```

• alloca()

Joissakin järjestelmissä on hyvin käyttökelpoinen funktio `alloca()`, jolla voidaan varata muistia pinosta aivan kuten `malloc()`:illa, mutta sillä erolla, että `alloca()`:lla varattu muisti vapautuu automaattisesti. (Miksi?) Yleisestä käytöstään huolimatta ei `alloca()`:a voi siirrettävyyssyistä suositella, valitettavasti.

Harjoitus 8.4 Selvitä, mitä määre `static` tarkoittaa funktioiden sisällä? Mitä se tarkoittaa funktion ulkopuolella? Mistä paikallisen staattisen muuttujan tila varataan?

Harjoitus 8.5 Mitä muuta pinossa on kuin lokaaleja muuttujia? Millaisia virheitä saattaisi esimerkiksi `chars[-8]` tai `chars[15]`:n kirjoittaminen aiheuttaa?

9 Syötteen lukeminen ja jäsentäminen

Tässä luvussa on kaksi osaa:ensimmäisessä alaluvussa saatetaan loppuun edellisen luvun esimerkki ja kahdessa viimeisessä alaluvussa esitetään pieni nelilaskuri.

9.1 Syötteen varmistettu lukeminen

Käyttäjä tekee kaikkensa ohjelman kaatamiseksi

• *Toiminta-varmuus*

Eli kaiken lukemisen pitää olla mahdollisimman toimintavarmaa, mutta samalla käyttäjäystävällistä. Viedään edellisessä luvussa aloitettu esimerkkikoodi loppuun. Käytetään päätettä ns. CBREAK–moodissa, jolloin käyttäjän ei tarvitse painaa rivinvaihtoa. Pidetään myös huoli siitä, että käyttäjä painoi jotakin näppäimistä 'y', 'Y', 'n' tai 'N'.

```
#include <stdio.h>
#include <sys/ioctl.h>

/* Gets one character from the terminal with the terminal in CBREAK mode.
 * If echo_it, echo the character, otherwise hide it. NON-PORTABLE CODE!
 */
char get_char_from_tty(boolean_t echo_it)
{
  char c;
  struct sgttyb original_state, cbreak_state;

  /* Set the terminal in CBREAK mode. Read tty(4). */
  ioctl(0, TIOCGETP, &original_state);
  cbreak_state = original_state;
  cbreak_state.sg_flags |= CBREAK;
  ioctl(0, TIOCSETP, &cbreak_state);

  /* Get the character, echo if wanted. */
  c = getchar();
  if (echo_it)
    putchar(c);

  /* Return the terminal to the original state. */
  ioctl(0, TIOCSETP, &original_state);

  return c;
}
```

```
/* Ask the user the given question and return TRUE, if the user answered
 * 'y' or 'Y', FALSE if he answered 'n' or 'N'.
 */
boolean_t confirm(char *question)
{
  printf("%s (y/n) : ", question);
  while (TRUE)
    switch (get_char_from_tty(TRUE)) {
    case 'y':
    case 'Y':
      putchar('\n');
      return TRUE;
    case 'n':
    case 'N':
      putchar('\n');
      return FALSE;
    default:
      printf("\nPlease, answer 'y' or 'n' : ");
      break;
    }
  /*NOTREACHED*/
}
```

⋆ **Harjoitus 9.1** Tee seuraavat kaksi rutiinia:

- `char *read_line(int max_line_length);`

  Varaa `max_line_length+1`:n merkin verran tilaa, lukee siihen syötteen käyttäjältä CBREAK–moodissa ja palauttaa luetun stringin.

- `int read_int(int min_int, int max_int);`

  Lukee käyttäjältä kokonaisluvun väliltä `min_int`…`max_int`, jonka se palauttaa.

Molempien rutiinien tulee olla mahdollisimman luotettavia (*robust*eja). Rutiinien käyttäjäystävällisyyttä voit lisätä käyttämällä `curses`(3)–paketin palveluja.

9.2 Syötteen "ymmärtäminen", `lex` ja `yacc`

• *Analyysin vaiheet*

Yleensä hiemankin monimutkaisemman syötteen lukeminen kannattaa tehdä vaiheittain. Ohjelmointikielissä on yleensä ainakin kaksi tasoa. Ensimmäisenä on leksikaalinen analyysi eli syötteen (`"3+sin(8);"`) erottaminen tokeneiksi (`3, PLUS_OP, sin, OPEN_PAREN, 8, CLOSE_PAREN, SEMICOLON`). Toinen vaihe on gramaattinen analyysi eli tokenien jäsentäminen (*parsing*) esimerkiksi jäsennyspuuksi tai kuten myöhemmin esitettävissä koodinpätkissä suoritetaan laskenta.

```
  PLUS_OP
  /     \
 3      sin
          \
           8
```

C:ssä näitä vaiheita edeltää preprosessointivaihe, jossa tapahtuu mm. makrojen lavennus. Mikäli syöte on hyvin yksinkertainen, voidaan kaikki tehdä yhdessä. Yleisenä periaatteena myös muilla ohjelmoinnin aloilla on

Jaa monimutkainen, mahdollisesti jatkuva tai tapahtumaohjattu muunnostoiminta tasoihin

Tästä periaatteesta hyvinä esimerkkeinä ovat ISO:n OSI–malli, X11 ja yleinen kääntäjäteknologia.

• `lex` ja `yacc`

Ehdottomasti joustavin ja yleensä myös riittävän nopea tapa erottaa syötteestä sanoja ja muita *leksikaalisia yksiköitä*, on `lex`in tai jonkun vastaavan ohjelman, esimerkiksi `flex`in, tuottaman koodin kutsuminen. Vastaavasti `yacc` tai sitä vastaava `bison` tuottaa sille annetun kieliopin jäsentäjän. Näiden ohjelmien täydellinen tai edes pinnallinen läpikäyminen ei tässä ole mahdollista, mutta kevyt tutustuminen ja mainos on paikallaan. Niinpä tässä esitetään pieni laskin sekä ilman edellämainittuja työkaluja ja niitä käyttäen toteutettuna. Laskimen käyttöesimerkki alla:

*2*7+9-1;*
`22.000000`
sin(3.141);
`0.000593`
pow(sin(1),2)+pow(cos(1),2);
`1.000000`

Ensiksi yhtenä tiedostona oleva yli 300 riviä pitkä ilman `yacc`ia ja `lex`iä toteutettu versio. Koodi on kaikin puolin varsin opettavaista, siinä sovelletaan käytäntöön tässä oppaassa aiemmin esitettyjä ohjeita, ja vaikka ei haluaisikaan perehtyä `yacc`iin tai `lex`iin, kannattaa tämä versio lukea kaikesta huolimatta.

additup.c

```
/*
 * A simplified bc for doubles. Semicolons ‘;’ used as expression
 * separators. eg.
 *   sin(3.14159);
 *   7*4 + 3;
 *   pow(sin(1), 2) + pow(cos(1), 2); etc.
 *
 * 12.8.1992 Kenneth Oksanen <cessu@cs.hut.fi>
 *   ANSIfied. ‘shift()’ and ‘match()’ are now functions.
 *
 * 2.8.1991 Kenneth Oksanen <cessu@niksula.hut.fi>
 *   Clean ups, reorganizations.
 *
 * 9.6.1991 Kenneth Oksanen <cessu@niksula.hut.fi>
 *   Cleaned up the code a bit, added more comments.
 *
 * 25.5.1991 Kenneth Oksanen <cessu@niksula.hut.fi>
 *   First Version.
 */

#include <stdio.h>
#include <strings.h>
#include <ctype.h>
#include <math.h>


/*
 * These should be in some utility library and merely #included
 */

typedef enum { FALSE = 0, TRUE = 1 } boolean_t;

#define EXIT_NO_ERROR 0              Päästään eroon ainakin muutamista
#define EXIT_USER_ERROR 1            ärsyttävistä maagisista luvuista.
#define EXIT_PROGRAM_ERROR 2

/*
 * Info about predefined functions stored in a NULL-ending vector.
 */

struct {
  char *name;
  int arity;
  double (*function)(void);
} function_vector[] = {              Huomaa, miten taulukkoa
  { "sin",  1, sin },                voi käyttä hyväksi toistuvan
  { "cos",  1, cos },                ohjelmakoodin välttämiseksi
  { "atan", 1, atan },
  { "sqrt", 1, sqrt },
  { "exp",  1, exp },
  { "ln",   1, log },
  { "pow",  2, pow },
  { NULL,   0, NULL }
};
```



```
/*
 * Not elegant, but abbreviates things. Should be more or equal to
 * the amount of chars in longest name in function_vector[] .
 */
#define MAX_FUNCTION_NAME_LENGTH 10

typedef enum {
  END_OF_FILE,     OPEN_PAREN,     CLOSE_PAREN,     SEMICOLON,
  COMMA,           PLUS_OP,        MINUS_OP,        TIMES_OP,
  DIVIDES_OP,      FUNCTION_NAME,  CONSTANT,        END = 0
} token_type_t;                    Lisää hyviä nimiä.


typedef struct {
  token_type_t token_type;         Tällä structilla lekseri
  union {                          kommunikoi parserin kanssa.
    double numeric;
    int function_index;
  } value;
} token_t;


struct {
  int ch;                          Tämäkin säästää vaivaa myöhemmin.
  token_type_t token_type;
} one_char_tokens[] = {
  { '(', OPEN_PAREN },  { ')', CLOSE_PAREN },   { ';', SEMICOLON },
  { ',', COMMA      },  { '+', PLUS_OP     },   { '-', MINUS_OP  },
  { '*', TIMES_OP   },  { '/', DIVIDES_OP  },   { '\0', END      }
};


/* The lexer. Gets the next token, analyzes it and stores info about it
 * in the given address.
 */

void get_next_token(token_t *token)      Selvitä, miksi välitetään osoite!
{
  int ch, i;

#define VALUED_RETURN(type, field, analyzed_value, pushback, ch) \
  do { \
    if (pushback) \
      ungetc((ch), stdin); \         Ilman näitä selviäisi myös.
    (token)->token_type = (type); \
    (token)->value.field = (analyzed_value); \
    return; \
  } while (FALSE)

#define RETURN(type, pushback) VALUED_RETURN(type, numeric, 0, pushback, EOF)

  /*
   * Read away all whitespaces. If isspace() weren't a macro or if it were
   * guaranteed to evaluate its argument only once, this would be
   *
```

```
   *   while (isspace(ch = getchar()))
   *     ;
   *
   * Notice that isspace() works correctly also on EOF.
   */
  do {
    ch = getchar();
  } while (isspace(ch));

  if (ch == EOF)
    RETURN(END_OF_FILE, FALSE);

  /*
   * ch was not EOF, now test if it is one of the one character long tokens
   * defined in one_char_tokens[].
   */
  for (i = 0; one_char_tokens[i].ch; i++)
    if (one_char_tokens[i].ch == ch)
      RETURN(one_char_tokens[i].token_type, FALSE);

  /*
   * If ch is a digit, the token to be read is a number.
   */
  if (isdigit(ch)) {
    double value = 0, unit_decimal_value;

    do { /* Get the integer part ... */
      value = 10 * value + ch - '0';
      ch = getchar();
    } while (isdigit(ch));
    if (ch != '.')
      VALUED_RETURN(CONSTANT, numeric, value, TRUE, ch);
    unit_decimal_value = 1; /* ... and the possible decimal part. */
    ch = getchar();
    while (isdigit(ch)) {
      unit_decimal_value /= 10;
      value += unit_decimal_value * (ch - '0');
      ch = getchar();
    }
    VALUED_RETURN(CONSTANT, numeric, value, TRUE, ch);
  }

  /*
   * ch starts a function name, read it ...
   */
  if (isalpha(ch)) {
    char s[MAX_FUNCTION_NAME_LENGTH+1];

    /*
     * A function name is [a-zA-Z][a-zA-Z0-9]* as a regexp. Read such
     * a regexp as long as it is possible.
     */
    for (i = 0; i < MAX_FUNCTION_NAME_LENGTH && isalnum(ch); i++) {
      s[i] = ch;
      ch = getchar();
    }
```

Lekserin hankalimpia paikkoja. Kannattaa lukea!

```
    s[i] = '\0';

    /*
     * Search for the name in function_vector[] and return the index.
     */
    for (i = 0; function_vector[i].name != NULL; i++)
      if (!strcmp(function_vector[i].name, s))
        VALUED_RETURN(FUNCTION_NAME, function_index, i, TRUE, ch);

    /*
     * If name didn't match any of the predefined functions,
     * report an error.
     */
    fprintf(stderr, "unrecognized function name '%s'\n", s);
    exit(EXIT_USER_ERROR);
  }

  /* The input could not be recognized.
   */
  fprintf(stderr, "garbled input\n");
  exit(EXIT_USER_ERROR);

#undef RETURN
#undef VALUED_RETURN
  /*NOTREACHED*/
}
```

Miksi tervettä sanoa undef*?*

```
/*
 * The lexer ends and parser starts here. The parser needs to know one
 * lexical item ahead in order to know what to do. yacc's REDUCE
 * corresponds to return from a parse_and_eval_...-function, shift
 * means to continue by asking the lexer to read and analyze the next
 * lexical item.
 */

static token_t lookahead;           /* Initialized in main() */


static void shift(void)
{
  get_next_token(&lookahead);       /* Read the next token. */
}


static void match(token_type_t t, char *message)
{
  if (lookahead.token_type != t) {
    fprintf(stderr, "expected %s\n", message);
    exit(EXIT_USER_ERROR);
  }
  shift();
}


static double parse_and_eval_expr(void);
```

```
/*
 * The highest precedence level corresponds to this function
 */

double parse_and_eval_factor(void)
{
  double tmp;
  int i;

  switch (lookahead.token_type) {
  case CONSTANT:
    tmp = lookahead.value.numeric;
    shift();
    return tmp;
  case OPEN_PAREN:
    shift();
    tmp = parse_and_eval_expr();
    match(CLOSE_PAREN, "')'");
    return tmp;
  case MINUS_OP:
    shift();
    return -parse_and_eval_factor();
  case FUNCTION_NAME:
    i = lookahead.value.function_index;
    shift();
    match(OPEN_PAREN, "'('");
    tmp = parse_and_eval_expr();
    switch (function_vector[i].arity) {
    case 1: /* Unary function */
      match(CLOSE_PAREN, "')'");
      return function_vector[i].function(tmp);
    case 2: /* binary function */
      match(COMMA, "','");
      tmp = function_vector[i].function(tmp, parse_and_eval_expr());
      match(CLOSE_PAREN, "')'");
      return tmp;
    default:
      fprintf(stderr, "Function arities above 2 not implemented.\n");
      exit(EXIT_PROGRAM_ERROR);
    }
    /*NOTREACHED*/
    break;
  default:
    fprintf(stderr, "syntax error\n");
    exit(EXIT_USER_ERROR);
  }
  /*NOTREACHED*/
}


double parse_and_eval_term(void)
{
  double value;
  token_type_t operator;
```



```
  value = parse_and_eval_factor();
  while ((operator = lookahead.token_type) == TIMES_OP ||
         operator == DIVIDES_OP) {
    shift();
    if (operator == TIMES_OP)
      value *= parse_and_eval_factor();
    else
      value /= parse_and_eval_factor();
  }
  return value;
}


double parse_and_eval_expr(void)
{
  double value;
  token_type_t operator;

  value = parse_and_eval_term();
  while ((operator = lookahead.token_type) == PLUS_OP ||
         operator == MINUS_OP) {
    shift();
    if (operator == PLUS_OP)
      value += parse_and_eval_term();
    else
      value -= parse_and_eval_term();
  }
  return value;
}
```

Aja ohjelma esim. syötteellä `sin(3-7.1);` *mielessäsi ymmärtääksesi sen toiminnan!*

```
int main(int argc, char *argv[])
{
  shift();

  while (lookahead.token_type != END_OF_FILE) {
    printf("%lf\n", parse_and_eval_expr());
    if (lookahead.token_type == SEMICOLON)
      shift();
    else if (lookahead.token_type != END_OF_FILE) {
      fprintf(stderr, "expected ';' or '^D' (end of file)\n");
      exit(EXIT_USER_ERROR);
    }
  }
  exit(EXIT_NO_ERROR);
  /*NOTREACHED*/
}
```

Seuraavaksi jossakin määrin vastaavat toiminnot tarjoava lekseri `lex`illä kirjoitettuna, tiedostona `lexer.l`. `lex`istä ja `yacc`ista saa tietoa useimpien UNIX–järjestelmien mukana tulevista oppaista ja monista kirjoista, esim. **Kernighan, Pike**, *"The UNIX Programming Environment"*, **Aho, Sethi, Ullman**, *"Compilers, Principles, Techniques and Tools"* jne.

lexer.l

```
%{
/*
 * Sumitup is like additup, a simple expression evaluator for floating
 * point numbers.
 *
 * 25.5.1991 Kenneth Oksanen <cessu@niksula.hut.fi>
 *   First version.
 */

#include <math.h>
#include <stdio.h>
#include "parse.tab.h"
%}

%%

[ \t\n]+             { ; /* Pass them */ }
\+                   { return(PLUS_OP); }
-                    { return(MINUS_OP); }
\*                   { return(TIMES_OP); }
\/                   { return(DIVIDES_OP); }
\,                   { return(COMMA); }
\(                   { return(OPEN_PAREN); }
\)                   { return(CLOSE_PAREN); }
;                    { return(SEMICOLON); }
[0-9]+(\.[0-9]*)?    { yylval.numerical = atof(yytext); return(CONSTANT); }
sin                  { yylval.function = sin; return(UNARY_FUNCTION); }
cos                  { yylval.function = cos; return(UNARY_FUNCTION); }
atan                 { yylval.function = atan; return(UNARY_FUNCTION); }
sqrt                 { yylval.function = sqrt; return(UNARY_FUNCTION); }
exp                  { yylval.function = exp; return(UNARY_FUNCTION); }
ln                   { yylval.function = log; return(UNARY_FUNCTION); }
pow                  { yylval.function = pow; return(BINARY_FUNCTION); }

%%

int yywrap(void)
{
  return 1;
}
```

`yacc`in tuottama C–koodi sisältää kutsuja ja muita viittauksia `lex`in tuottamaan koodiin, olennaisin `get_next_token()`:ia vastaava `yylex()`. Vastaavasti `parse.y`:n määrittelyt, esimerkiksi `PLUS_OP`, kerrotaan lekserille tiedoston `parse.tab.h` kautta. Alla `yacc`ille annettava syöte, tiedosto `parse.y`.

parse.y

```
%{
/*
 * Yacc file for sumitup.
 *
 * 25.5.1991 Kenneth Oksanen <cessu@niksula.hut.fi>
 *   First version.
 */

#include <stdio.h>

int yyerror(char *msg)
{
  fprintf(stderr, "%s\n", msg);
}

%}                                    Selaa parse.tab.c:tä nähdäksesi tuloksen!

%start expressions

%union {
  double numerical;
  double (*function)(void);
}

%token COMMA OPEN_PAREN CLOSE_PAREN
%token <numerical> CONSTANT
%token <function> UNARY_FUNCTION BINARY_FUNCTION
%type <numerical> expression
                                      Varmista, että olet ymmärtänyt,
%left SEMICOLON                       mitä presedenssi ja assosiatiivisuus
%left PLUS_OP MINUS_OP                tarkoittavat.
%left TIMES_OP DIVIDES_OP
%left UMINUS                          Katso UNIXin ohjelmoijan oppaasta!

%%

expressions: expression SEMICOLON { printf("%lf\n", $1); } expressions
        |                             Rekursiivinen määritelmä vastaa silmukkaa!
        ;

expression: OPEN_PAREN expression CLOSE_PAREN              { $$ = $2; }
        | UNARY_FUNCTION OPEN_PAREN expression CLOSE_PAREN { $$ = $1($3); }
        | BINARY_FUNCTION OPEN_PAREN expression COMMA expression CLOSE_PAREN
                                                          { $$ = $1($3,$5); }
        | expression PLUS_OP expression                    { $$ = $1 + $3; }
        | expression MINUS_OP expression                   { $$ = $1 - $3; }
        | expression TIMES_OP expression                   { $$ = $1 * $3; }
        | expression DIVIDES_OP expression                 { $$ = $1 / $3; }
        | MINUS_OP expression %prec UMINUS                 { $$ = - $2; }
        | CONSTANT                                         { $$ = $1; }
        ;

%%
```

```
extern FILE *yyin;

main(int argc, char *argv[])
{
  yyin = stdin;
  return yyparse();
}
```

`yacc` tuottaa funktion nimeltä `yyparse()`, jota kutsutaan käyttäjän kirjoittamasta muusta koodista, tässä tapauksessa pääohjelmasta, joka on sijoitettu `parse.y`:n loppuun.

• *Makefile*

Johdantona seuraavaan lukuun esitetään vielä ylläolevien kolmen tiedoston kääntämiseen tarvittava `Makefile`.

Makefile

```
# A very simple makefile, could/should be better!

sumitup: parse.tab.c lex.yy.c
	cc -O parse.tab.c lex.yy.c -o sumitup -ll -lm

parse.tab.h parse.tab.c: parse.y
	yacc -d parse.y          Tähän Makefileeseen viitataan
	                         seuraavassa luvussa.
lex.yy.c: lexer.l parse.tab.h
	lex lexer.l

clean:
	rm -f core sumitup lex.yy.c parse.tab.c *~ #*
```

• *Työkalut säästävät vaivaa ja parantavat konfiguroitavuutta*

Kokonaisuudessaan `yacc`ia ja `lex`iä käyttäen yli 300 riviä C:tä puristui noin 110 riviin. Tilansäästöä suurempi hyöty on jälkimmäisen version huomattavasti helpompi konfiguroitavuus ja parempi luettavuus.

Käytä työkaluja

Yleensä monimutkaistenkin työkalujen oppimisen vaiva korvautuu moninkertaisesti säästettynä hikenä, kiroiluna ja kyyneleinä.

Harjoitus 9.2 Laajenna `lex`illä kirjoitettua lekseriä siten, että se hallitsee myös mantissa–eksponentti –muodossa annetut liukuluvut, siis esimerkiksi `3.7E-7`

Harjoitus 9.3 Laajenna ensimmäistä versiota niin, että se hallitsee huomattavasti enemmän funktioita, esimerkiksi trigonometrisia ja hyperbolisia funktioita ja niiden käänteisfunktioita. Lisää mahdollisuus käyttää kolmiargumenttisia funktioita

⋆ **Harjoitus 9.4** Paranna ohjelman virheensietoa, esimerkiksi niin, että virheen sattuessa tulostetaan virheilmoitus, mutta ollaan valmiita jatkamaan seuraavasta ';':n erottamasta syötteestä.

⋆ **Harjoitus 9.5** Laajenna ohjelmia niin, että se hallitsee reaalilukujen lisäksi myös kompleksiluvut. Kannattaa pohtia tarkasti, missä kompleksiluvun ”ymmärtäminen” tapahtuu, lekserissä vai parserissa.

⋆⋆ **Harjoitus 9.6** Laajenna jälkimmäistä versiota niin, että voit käyttää muuttujia, suorittaa ehto– ja toistorakenteita ja määritellä funktioita. . . HUPS! Meillähän on ohjelmointikielen tulkki!

10 Ohjelmointivirheet

10.1 C:n ansoja

• *Syntaksi*

C:n syntaksin vuoksi seuraavat virheet ovat hyvin yleisiä:

- Sijoitusoperaattori `=` ja vertailuoperaattori `==` ovat sotkeentuneet. Tätä virhettä kääntäjä ei välttämättä huomaa.
- Bittioperaattorien `^, |, &, <<, >>` käyttö loogisten ja matemaattisten operaattorien sijasta aiheuttaa helposti virheitä, koska niillä on eri presedenssi. `|` ja `&` evaluoivat aina molemmat operandinsa, `||` ja `&&` vain tarvittaessa (*short circuit evaluation*).
- Älä kirjoita `a=*b`, sillä vanhat kääntäjät saattavat ymmärtää sen samaksi kuin `a*=b`. Laita välilyönnit `=`:n ympärille `a = *b`.
- Kerto– ja jakolaskuoperaattorien presedenssit ovat samat ja ne assosioidaan vasemmalta oikealle.
- `^` ei ole potenssilasku, vaan exclusive–or.

• *Laskujärjestys*

Laskujärjestys on C:ssä usein määrittelemätön. Esimerkiksi funktiokutsun argumenttien laskujärjestystä ei ole määritelty. Epäselvää on myös, missä välissä selvitetään mitä funktiota kutsutaan[7].

Lausekkeen sivuvaikutukset luvataan tehdyiksi lähimpien ympäröivien `||`, `&&`, `?`, `:` tai `,` –operaattorien sisällä. On siis epäselvää, mikä lausekkeen `a++ + ++a` arvo on. Samoin on epäselvää, miten `a` muuttuu lausekkeessa `a = a++`. Sivuvaikutukset luvataan suoritetuiksi lausekkeen myös lopun jälkeen (`;`, `}` tai rakenne).

• *Makrot*

Luvussa 6.2 näytettiin makrojen tekstuaalisen lavennuksen aiheuttavan monia vaikeasti selvitettäviä ongelmia. Siksi makrojen käytössä tulee olla hyvin huolellinen. Monet argumentteja saavat makrot kannattaisi kirjoittaa funktioiksi. Ongelmatilanteissa kannattaa vilkaista preprosessoitua C–koodia, jonka kääntäjä yleensä antaa `-E`–optiolla.

• *Bitit ja tavut*

Bittien ja tavujen kanssa on oltava hyvin varovainen, sillä eri tietokoneet tallettavat samoja tyyppejä eri tavoin. Tavujärjestys vaihtelee (*bigendian, little–endian*) ja jotkut tietokonearkkitehtuurit osaavat viitata kokonaislukuihin vain jos ne on tallennettu neljällä jaolliseen muistiosoitteeseen (*alignment*). Siksi kannattaa välttää bitti– ja tavutasolla tapahtuvaa laskentaa. Datan esitykselle kannattaa etsiä korkeampi abstraktio (taulukot, struktuurit jne.) tai esittä se kokonaan toisella tavalla (ascii–tiedostoja mielummin kuin binääritiedostoja jne.)

• *Tyypit*

Vaikka C on heikohkosti tyypitetty kieli, ei tyyppien kanssa kannata leikkiä aivan miten tahansa. Erityyppisiin alkioihin osoittavien osoitintyyppien sotkemisesta seuraa helposti virhe, Esimerkiksi tyyppejä `char *`, `int *` tai `long *` ei saa sekoittaa. Tyypitä (*cast*) `NULL` oikean tyyppiseksi, jos se annetaan argumenttina.

• *Muisti*

Muistin käyttöön liittyvät ongelmat ovat äärimmäisen hankalia selvittää. Virhe saattaa piileskellä ohjelmassa kauan kunnes virheen olemassaoloa osataan epäillä. Virheen paikantaminen on äärimmäisen hankalaa. Alla virheellinen ohjelma ja sen tulostukset eri ympäristöissä:

[7] Tällä on merkitystä, jos kutsutaan muuttujassa olevaa funktiota argumentilla, joka muuttaa kyseisen funktiomuuttujan arvoa.

```
#include <stdio.h>

main(int argc, char *argv[])
{
  int a[1], m = 1;

  printf("%d\n", m += m-- - --m + a[m -= --m]);
}
```

Kone (prosessori)	Kääntäjä ja optiot	Tulostaa:
SUN 4	`cc`	0
(SPARC)	`gcc`	2
	`gcc -O`	-1
SONY NWS-3400	`cc`	2
(MIPS R3000)	`gcc`	0
	`gcc -O`	-1
HP 9000/720 (PA)	`cc`	1
SONY NWS-1510	`cc`	2
(Motorola 68030)	`gcc`	0
DecStation 5000/200	`cc`	-1
(MIPS R3000)	`gcc`	2
	`gcc -O`	-1
PC 486	`cc`	0
(Intel 80486)	`cc -O`	-1
	`gcc`	0
	`gcc -O`	-1
VaxStation 3000	`cc`	2
	`cc -O`	0
IBM RS/6000	`cc`	0
(RIOS)	`cc -O`	-559038737
	`gcc`	2
	`gcc -O`	-1

Harjoitus 10.1 Evaluoi seuraavia virheellisiä lausekkeita eri koneissa, eri kääntäjillä ja optimoinneilla käännettynä ja erilaisiin ohjelmanpätkiin upotettuina. Millaisia tuloksia saat?

- `*((int *) "abcd")`
- `*((char *) &m)`, kun `m` on tyyppiä `int` ja arvoltaan 2.
- `m = m++`
- `m = m-- - --m`
- `m = m + m>>2`

Harjoitus 10.2 Mitä standardi sanoo `short`:in, `int`:in ja `long`:in pituudesta?

10.2 `assert()`

• *Invariantit ja assertiot*

Invariantti ja *assertio* tarkoittavat tietyssä ohjelman kohdassa tai kohdissa aina totta ehtolauseketta. Käytännössä invariantilla tarkoitetaan teoreettista työkalua, hyvin tiukkaa ehtoa, jonka käyttämisellä pyritään esimerkiksi todistamaan ohjelman oikeellisuus. Assertio taas on käytännön työkalu, luonteeltaan usein karkeampi tarkistus, jolla pyritään kohtuullisella vaivalla löytämään mahdollisimman paljon virheitä mahdollisimman varhain.

```
#include <assert.h>

void print_strings(int strings, char *string_array[])
{
  int i;

  assert(string_array != NULL);
  assert(strings >= 0);
  for (i = 0; i < strings; i++) {
    assert(string_array[i] != NULL);
    printf("%s\n", string_array[i]);
  }
}
```

• `assert()` *vs.* `fatal()`

`assert()`:illa ei ole tarkoitus tarkistaa esimerkiksi systeemikutsujen onnistumista, sillä sen antama virheilmoitus on yleensä käyttäjälle melko hyödytön eikä epäonnistuneesta `assert()`:ista voi enää jatkaa. Sensijaan ohjelman kehityksen apuvälineenä ohjelmointivirheiden havaitsemiseksi `assert()` on parhaimmillaan. Jos makro `NDEBUG` on määriteltynä ei assertioita käännöksessä oteta huomioon.

10.3 `gdb`

• *Tarkoitus*

`gdb` on ns. *debuggeri*. `gdb`:llä voidaan muun muassa asettaa *breakpoint*teja, joissa ohjelman suoritus keskeytyy. Ohjelman suoritusta voidaan askeltaa (*trace*) ja ohjelman tilaa voidaaan tutkia ja manipuloida.

`gdb` esitellään esimerkin avulla. Alla on oppaan alusta tuttu funktio `strccmp()`, mutta tällä kertaa siinä on virhe.

```
int strccmp(char *s1, char *s2)
{
  while (*s1 && *s2) {
    if (*s1 != *s2)
      return (*s1 < *s2) ? -1 : 1;
    s1++;
  }
  return 0;
}
```

gdb

```
cessu@joker <opas> 2:39am: gcc -g bug.c -o bug
cessu@joker <opas> 2:39am: gdb bug
GDB 3.5, Copyright (C) 1989 Free Software Foundation, Inc.
There is ABSOLUTELY NO WARRANTY for GDB; type "info warranty" for details.
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "info copying" to see the conditions.
Reading symbol data from /m/joker/1/staff/cessu/c/olab/opas/bug...done.
Type "help" for a list of commands.
(gdb) break strccmp                   Asetetaan breakpoint
Reading in symbols for bug.c...done.
Breakpoint 1 at 0x5a: file bug.c, line 6.
(gdb) run aa aa1                      Ajetaan ohjelma
Starting program: /m/joker/1/staff/cessu/c/olab/opas/bug aa aa1

Bpt 1, strccmp (s1=(char *) 0x7fffdaf3 "aa", s2=(char *) 0x7fffdaf6 "aa1")
(bug.c line 7)
7         while (*s1 && *s2) {         Tultiin breakpointtiin, tämä on seuraava rivi
(gdb) print s1
$1 = (char *) 0x7fffdaf3 "aa"          Katsotaan mitä s1 ja s2 ovat
(gdb) print s2
$2 = (char *) 0x7fffdaf6 "aa1"
(gdb) step                            Askelletaan ohjelmaa
8           if (*s1 != *s2)
(gdb)
10          s1++;
(gdb)
7         while (*s1 && *s2) {
(gdb)
8           if (*s1 != *s2)            Hmmm, epäilys alkaa herätä...
(gdb)
10          s1++;
(gdb) print s1
$4 = (char *) 0x7fffdaf4 "a"           Tämä on normaalia, mutta
(gdb) print s2
$5 = (char *) 0x7fffdaf6 "aa1"         tämä on väärin. HAA! s2++ puuttui!
(gdb)
```

`gdb` on huomattavan laaja ja sen saa toimimaan mukavasti yhteistyössä GNU `emacs`in kanssa.

11 Projekti

11.1 Koodin jakaminen tiedostoihin

• *koodi– ja header–tiedostot*

Suurta C–ohjelmaa ei kannata kirjoittaa yhteen tiedostoon. Yleensä kooditiedostoa (`.c`) vastaa yksi header–tiedosto (`.h`), jonka sisältämät määrittelyt voidaan lukea `#include"..."`:lla muista tiedostoista. Poikkeuksena on `main()`:in sisältävä päätiedosto, jolla on harvoin header–tiedostoa. Ohjelman kaikille osille yhtenäisiä vakioita ja muita määrittelyjä määritellään yleensä tiedostossa `defs.h` tai `config.h`.

Jokaisen header–tiedoston tulee kertoa mitä vastaava kooditiedosto tarjoaa muille ohjelman osille. Tänne kuuluvat makrot, funktioiden prototyypit, tyyppinmäärittelyt jne.

• *Jako tiedostoihin*

Tiedoston koko voi vaihdella paljon, mutta yleensä koodin lukeminen tulee ikäväksi kun rivien lukumäärä kasvaa yli 500:n. Tiedosto kannattaa jakaa osiin, jos siinä olevat funktiot kuuluvat loogisesti eri kokonaisuuksiin. Tiedostoihin jakoa kannattaa miettiä ennen ainoankaan koodirivin kirjoittamista!

• *Sisällön järjestys*

Myös tiedostojen sisällä tulee vallita kuri. Tiukasti yhteen kuuluvat funktiot ryhmitellään lähelle toisiaan. Muuttujat ja tietorakenteet kannattaa esitellä mahdollisimman lokaalisesti ja lähellä niitä käyttävää koodia.

• *Koodi vs. kirja*

Koodi kirjoitetaan luettavaksi, aivan kuten kirjatkin. Suuret teokset laitetaan eri kansiin, sillä yhtä jättiläismäistä kirjaa on hankalampi käsitellä kuin montaa pientä. Tekstissä asiat kerrotaan johdonmukaisessa järjestyksessä (kaunokirjallisuudessa kronologinen järjestys on yleisin, ohjelmoinnissa järjestellään yleensä aihepiireittäin). Liian raskaita lauseita ja virkkeitä kirjoittava tai muuten epäselvää tekstiä kirjoittava kirjailija ei menesty – ei myöskään huonoa koodia tuottava ohjelmoija.

11.2 Tiedostot valmiissa ohjelmassa

• *Tiedostot*

Ohjelmoijien maailmassa, varsinkin UNIX–ympäristöissä, on kehittynyt melko selkeä käytäntö projektin tiedostoista ja niiden nimeämisestä.

- `README` sisältää periaatteessa mitä tahansa, jonka ohjelman tekijä haluaa kertoa muille. Yleensä siinä on myös lyhyt selostus ohjelman käyttötarkoituksesta, toiminnasta, käytöstä, historiasta jne.
- `COPYRIGHT` kertoo ohjelman copyright–ehdot.
- `INSTALL` sisältää ohjelman installointiohjeet, ellei pelkkä `make` riitä.
- `CHANGES` tai `ChangeLog` sisältää listan jonkin hetken jälkeen tehdyistä muutoksista.
- `MANIFEST` on luettelo projektiin tai toimitukseen kuuluvista tiedostoista.
- `FILES` on vaihtoehto edelliselle, mutta tässä tiedostojen sisältö selostetaan.
- `TODO` Tähän on hyödyllistä kerätään lista tulevissa versioissa toteutettavista ominaisuuksista.
- `BUGS` Lista virheitä käyttäjien varoittamiseksi.
- `Makefile` on aina hyvä olla olemassa.
- `*.h` ja `*.c` varsinaiset kooditiedostot.
- `*.[1-9]*` eli manuaalisivut.

Jos kooditiedostoja on kymmeniä, kannattaa harkita kooditiedostojen, dokumentaation, testien ja demojen sekä mahdollisesti myös binäärien sijoittamista eri hakemistoihin `src/`, `doc/`, `bin/`, `examples/` jne.

• *Alihakemistot*

Harjoitus 11.1 Mitä tarkoitetaan *information hiding*:illa?

Harjoitus 11.2 Tutki `gcc`:n tai jonkin muun suuren ja hyvin kirjoitetun ohjelman tiedostojen sisältöä.

11.3 Työkaluajattelu

UNIXin työkaluajattelua kannattaa käyttää hyväksi. Harvoin kannattaa kirjoittaa ohjelmaa, joka esimerkiksi optiolla `-s` järjestää tulostuksen aakkosjärjestykseen. Kannattaa mielummin käyttää UNIXin `sort`:ia. Vastaavasti esimerkiksi `grep`in, `tr`:n, `sed`in, `awk`in tai minkä tahansa muun työkalun toiminnallisuutta on tyhmää kirjoittaa uudelleen. Usein on järkevää kirjoittaa `sh`–, `awk`–, tai `perl`–ohjelma (*script*), joilla pieniä C–ohjelmia voi yhdistellä.

• *Työkaluajattelu*

11.4 `make`

`make`–ohjelman avulla helpotetaan ohjelman kääntämistä. `make` lukee kuvaustiedoston (`makefile` tai `Makefile`), jossa määriteltyjen riippuvuuksien perusteella se päättää mitkä tiedostot pitää kääntää uudestaan. Jos esimerkiksi tiedosto `foo.c` on muuttunut viimeisen käännöksen jälkeen, `make` osaa päätellä, että `foo.c` pitää kääntää uudelleen `foo.o`:ksi. Tällainen riippuvuus ilmoitetaan säännöllä

• *Tehtävä*

```
foo.o: foo.c
```

Riippuvuutta kuvaavan säännön vasemmalla ja oikealla puolella voi olla useita tiedostoja. Ellei esimerkiksi komentorivillä toisin mainita, `make` pyrkii toteuttamaan `Makefile`n ensimmäisen säännön.

• *Säännöt*

Seuraavalle riville kirjoitetaan käännöksessä tarvittava käsky, tässä tapauksessa ylläolevaa riviä seuraisi välittömästi rivi

```
        gcc foo.c -c
```

Monien ominaisuuksien joukosta esitettäköön vielä `make`n muuttujat, joiden avulla `Makefile`iden parametrointi käy tuskattomammin. Esimerkiksi `sumitup`in tekemisessä käytettyä `Makefile`ä voisi kehittää seuraavasti:

• *Muuttujat*

```
...
CC=gcc
CFLAGS=-O
LIBS=-lm -ll
...
sumitup: parse.tab.c lex.yy.c
        $(CC) $(CFLAGS) parse.tab.c lex.yy.c -o sumitup $(LIBS)
...
```

Mitä enemmän tiedostoja on tai mitä monimutkaisempia järjestelmiä käännetään, sitä enemmän hyötyä `make`n käyttämisestä on. Paras `make`n dokumentaatio on GNU–projektin `gmake`n mukana oleva yli satasivuinen, mutta hyvin helppolukuinen käyttöohje.

Harjoitus 11.3 Mitä tarkoitetaan implisiittisillä säännöillä ja automaattisilla muuttujilla (jälkimmäiset vain `gmake`:ssa)?

Harjoitus 11.4 Mitä tekee seuraava `Makefile`?

Makefile

```
OBJECTS=executor.o sym.o stack.o
HEADERS=defs.h sym.h stack.h

CC=gcc
CFLAGS=-g -O
LD=gcc
LDFLAGS=-g -O

all: $(OBJECTS)
        $(LD) $(LDFLAGS) $(OBJECTS) -o executor

executor.o: executor.c lex.yy.c
        $(CC) $(CFLAGS) executor.c -c

sym.o: sym.c
        $(CC) $(CFLAGS) sym.c -c

stack.o: stack.c
        $(CC) $(CFLAGS) stack.c -c

lex.yy.c: lex.l
        lex lex.l

$(OBJECTS): $(HEADERS)

clean:
        rm -f $(OBJECTS) core *~ lex.yy.c
```

11.5 rcs

• *Tarkoitus*

`rcs` on lyhenne sanoista *Revision Control System*. `rcs`:n avulla ohjelmoija vapautuu ikävästä versionhallinnan ja varmuuskopioinnin vaivasta. `rcs`:llä voi palata mihin tahansa aiempaan versioon jos jotain meni pieleen. Myös ryhmän yhteistyö helpottuu.

• *Käytön oppiminen*

`rcs` on yllättävän helppokäyttöinen. Kannattaa lukea allaoleva pieni käyttöesimerkki, sen jälkeen manuaalisivut `ci`(1), `co`(1) ja `rcs`(1).

RCS-typescript

```
cessu@sauna <sumitup> 3:21am: L            ls -lFg
total 4
-rw-r--r--  1 cessu    user          348 Jun 10 03:19 Makefile
-rw-r--r--  1 cessu    user          935 Jun 10 03:18 lexer.l
-rw-r--r--  1 cessu    user         1143 Jun 10 03:19 parse.y
```

RCS:n käyttöä ei vielä ole aloitettu. Tältä maailma näyttää.

```
cessu@sauna <sumitup> 3:21am: mkdir RCS
cessu@sauna <sumitup> 3:21am: head -17 parse.y
%{
/*
 * Yacc file for sumitup.
 *
 * HISTORY
 *
 *   $Log$                                 $Log$-merkistä RCS tietää minne laittaa
 */                                        tietoa versioista.

#include <stdio.h>

int yyerror(char *msg)
{
  fprintf(stderr, "%s\n", msg);
  exit 1;
}
cessu@sauna <sumitup> 3:21am: ci -u *
RCS/Makefile,v  <--  Makefile
initial revision: 1.1
enter description, terminated with ^D or '.':
NOTE: This is NOT the log message!
>> First revision
>> ^D
done
```

Sama muille tiedostoille.

```
cessu@sauna <sumitup> 3:22am: L
total 1
drwxr-xr-x  2 cessu    lang          512 Jun 10 03:21 RCS/
```

Versio on nyt talletettu. Seuraavaksi halutaan takaisin tiedosto `parse.y`*, jotta siihen voitaisiin tehdä parannuksia. Jos tiedostoja halutaan muuttaa, ne lukitaan optiolla* `-l`*. Jos tiedostoja halutaan vain lukea, niitä ei kannata lukita.*

```
cessu@sauna <sumitup> 3:22am: co -l RCS/parse.y
RCS/parse.y,v  -->  parse.y
revision 1.1 (locked)
done
cessu@sauna <sumitup> 3:22am: L
total 4
drwxr-xr-x  2 cessu    lang          512 Jun 10 03:22 RCS/
-rw-r--r--  1 cessu    user         1222 Jun 10 03:22 parse.y
```

Tässä välissä käyttäjä editoi tiedostoa `parse.y`

```
cessu@sauna <sumitup> 3:22am: L
total 6
drwxr-xr-x  2 cessu    lang          512 Jun 10 03:22 RCS/
-rw-r--r--  1 cessu    user         1212 Jun 10 03:22 parse.y
-rw-r--r--  1 cessu    user         1222 Jun 10 03:22 parse.y~
cessu@sauna <sumitup> 3:23am: rm *~
cessu@sauna <sumitup> 3:23am: ci parse.y        Talletetaan muutokset
RCS/parse.y,v  <--  parse.y
new revision: 1.2; previous revision: 1.1
enter log message:
(terminate with ^D or single '.')
>> Changed yyerror() so that it returns,
>> thus hopefully improving error recovery
>> ^D
done
cessu@sauna <sumitup> 3:25am: L
total 1
drwxr-xr-x  2 cessu    lang          512 Jun 10 03:24 RCS/
```

Harjoitus 11.5 Kuinka lukituksia käytetään ryhmätyöskentelyssä? Mitä hyötyä niistä on? Kuinka lukitsematon tiedosto voidaan lukita ja päinvastoin? Kuinka hyödyttömiä versioita poistetaan?

Harjoitus 11.6 Tutki `RCS`:n `,v`–tiedostoja. Miten eri versiot talletetaan? Kuinka `rcsdiff` toimii?

12 Selväkieliset selostukset

Kaikessa on pyrittävä selkeyteen ja ymmärrettävyyteen. Hankala ja liian monimutkainen selostus on aivan yhtä hyödytön kuin lukukelvoton koodi.

12.1 Koodin kommentointi

Alla katkelma taulukon `a[]` lajittelevasta ohjelmasta:

```
for (i = 0; i < size-1; i++) /* Loop through the array */
  for (j = i+1; i < size; j++)/* Loop starting from where the outer is */
    if (a[i] > a[j]) {       /* If the two a[]'s in wrong order, */
      tmp = a[i];            /* Exchange them with tmp. */
      a[i] = a[j];
      a[j] = tmp;
    }
```

Ylläolevat kommentit kertovat paljon sellaista, mitä lukija näkee suoraan ohjelmasta sen enempiä miettimättä. Jokainen näkee kaksi silmukkaa, joista sisempi käy läpi aina taulukon loppuosan. Näillä kommenteilla ei ole mitään arvoa; ne on vain jo kirjoitetun asian toistamista. Kaiken lisäksi ne ovat suorastaan harhaanjohtavia, sillä `tmp`:n kanssa ei vaihdeta mitään.

• *Kommentin informaatioarvo*

```
/* Sort the array a[0..size-1] to an increasing order using Bubblesort
 */
for (i = 0; i < size-1; i++)
  for (j = i+1; i < size; j++)
    if (a[i] > a[j]) {
      tmp = a[i];                  /* Exchange a[i] with a[j] */
      a[i] = a[j];
      a[j] = tmp;
    }
```

Alussa kerrotaan suoraan mitä koodilla on tarkoitus saada aikaan, mitä tietorakennetta se muokkaa ja minkä algoritmin implementaatio se on.

Vain poikkeustapauksissa on syytä kommentoida koodia itseään. Yleensä kommentin tehtävänä on kertoa asioita, joita ei koodista suoraan näe ja ennenkaikkea se, mitä koodilla pyritään saamaan aikaan!

12.2 Ohjelman dokumentointi

Dokumentaatio vaihtelee paljon. Jotkut laitostuneet ohjelmointiympäristöt tuottavat jokaisesta pienestä funktiosta monta sivua sovitun standardin mukaista dokumenttia. Tähän ei yleensä ole syytä pyrkiä, vaikka paperilla olevien dokumenttien ja koodissa olevien kommenttien tuleekin olla sopivasti päällekkäiset.

• *Sisältö*

Dokumentteissa kannattaa ylläpitää mm. seuraavia asioita:

- ohjelman yleiskuvaus yleensä usealla eri tarkkuudella esitettynä
- hakemistorakenne, tiedostot
- käytetyt aliohjelmakirjastot, käyttöjärjestelmän ominaisuudet, sovelluskehittimet jne.
- algoritmien tarkkoja kuvauksia, mahdollisesti niiden matemaattinen perusta ja aika– ja tilakompleksisuusarvioita
- viittauksia muihin dokumentteihin, vastaaviin töihin ja kirjallisuuteen.
- kirjanpito järjestelmän eri versioista, löydetyistä virheistä jne.

12.3 Käyttöohje

• *Sisältö*

Dokumentaation olennaisimpia osia on kuitenkin käyttöohje. Käyttöohjeessa kerrotaan kädestä pitäen kuinka ohjelma toimii. Riippuen ohjelman käytön monimutkaisuudesta, tulee käyttöohjeen tarjota seuraavia palveluja:

- ohjelman tarkoituksen ja tehtävän selostus
- käytetyn erikoissanaston selitys
- tiivis johdatus ohjelman käyttöön, eli "pikaopas", jonka jälkeen ohjelmaa tulisi osata käyttää välttävästi
- täsmällinen ja järjestelmällinen, mahdollisesti hakuteosmainen selostus kaikista toiminnoista
- hakemisto.

Käyttöohjeen ehdoton vaatimus on, että se on helppolukuista ja ymmärrettävää tekstiä. Kuvia ja esimerkkiajoja kannattaa käyttää.

12.4 UNIXin manuaalisivu

Usein pelkkä UNIXin manuaalisivu riittää käyttöoppaaksi. Manuaalisivut tulee tuottaa joko `nroff`illa tai `troff`illa. Manuaalisivujen yleisimmät kentät ovat:

• *Sisältö*

`NAME`	Ohjelman[8] nimi ja yhden virkkeen kuvaus tai lyhennenimen selostus.
`SYNOPSIS`	Ohjelman käynnistäminen, aliohjelman tai makron käyttäjälle näkyvä määritelmä.
`DESCRIPTION`	Vapaamuotoinen kuvaus ohjelmasta.
`OPTIONS`	Ohjelman optiot.
`USAGE`	Ohjelman käyttöohje.
`RETURN VALUES` `DIAGNOSTICS`	Paluuarvot.
`ERRORS`	Ohjelman, tässä varsinkin aliohjelman, toiminta virhetilanteissa.
`ENVIRONMENT`	Ympäristömuuttujat, johon ohjelma viittaa.
`FILES`	Ohjelman käyttämät tiedostot.
`SEE ALSO`	Viittaukset muihin vastaaviin ohjelmiin. Tätä kenttää kannattaa käyttää!
`BUGS`	Tunnetut virheet ja puutteet.
`AUTHORS`	Ohjelman tekijät.

Monia muitakin kentännimiä esiintyy.

Manuaalisivut on luokiteltu.

• *Luokittelu*

1. ohjelmat
2. systeemikutsut
3. aliohjelmakirjastot
4. laiteohjaimet ja protokollat
5. tiedostoformaatit
6. pelit
7. sekalaista
8. ylläpito.

Nykyisin on yleistä, että käytössä on muitakin luokkia esimerkiksi ikkunointijärjestelmää varten.

Harjoitus 12.1 Kirjoita manuaalisivu `sumitup`:sta.

13 Yhteenveto

• *Työtä, työtä*

Hyväksi ohjelmoijaksi ei ole oikotietä. Hyväksi ohjelmoijaksi tuleminen on henkisesti raskas urakka. Vaikka muiden ohjelmoijien kritiikki joskus masentaa, on se kuitenkin arvokasta palautetta. Toisten ihmisten töitä kannattaa lukea ja verrata niitä omiinsa. Omiin töihinsä pitää aina olla kriittinen ja etsiä jatkuvasti parempia vaihtoehtoja ratkaista annettu ongelma! Ohjelmoijan pitää olla suunnitelmallinen ja punnita eri vaihtoehtoja ennen työhön ryhtymistä — ohjelmointityöstä usein yli puolet ajasta kuluu ongelmien hahmottamiseen ja ratkaisumenetelmien kehittämiseen, työhön joka sujuu parhaiten silmät kiinni tai paperi ja kynä kädessä.

• *Muut kielet*

Ohjelmointitaitoa ei missään nimessä mitata osattujen ohjelmointikielten lukumäärällä, mutta silti kannattaa perehtyä ainakin kolmeen riittävän erilaiseen ohjelmointikieleen. Itse suosittelen C:tä, Common Lispiä, Prologia ja jotain modernia objektiorientoitunutta ohjelmointikieltä, kuten Sather, Modula-2, C++ jne. *Ravi Sethi*n kirjoittama kirja "*Programming Languages, Concepts and Contstructs*" on hyvin suositeltava kirja kaikille, jotka haluavat perehtyä yleisimpiin ohjelmointikielten perusajatuksiin.

• *Muut ohjelmointitaidon osa–alueet*

Myös kohtalainen tiedollinen osaaminen on välttämätöntä: Tärkeimmät algoritmit ja tietorakenteet on syytä opetella todella hyvin. Käyttöjärjestelmien tunteminen säästää monen turhan kokeilun vaivan. Valmiisiin ohjelmointityökaluihin perehtyminen säästää usein paljon vaivaa. Ohjelmistometodiikkojen soveltamistaito auttaa määrittelemään ja paloittelemaan suuria ongelmia jne.

• *Tietojenkäsittely vs. ohjelmointi*

Kannattaa myös pitää mielessä, että tietojenkäsittely on tieteenä ja ammattina paljon muuta kuin ohjelmointia, siitäkin huolimatta, että hyvä käytännön ohjelmointitaito on korvaamaton.

⋆ **Harjoitus 13.1** Kirjoita jokin vähintään parin sadan rivin ohjelma uudelleen parin kuukauden väliajoin. Talleta kaikki versiot ja vertaile niitä keskenään ainakin vuoden ajalta.

⋆⋆ **Harjoitus 13.2** Kerää eri kielten suositeltavia tyylejä. Vertaile niitä. Miten kirjoittaisit Erathosteneen seulan eri ohjelmointikielillä? Entä polynomien jakolaskun? Mieti yleisiä suuntaviivoja miten ohjelmointikieli tulisi valita ongelman mukaan.

A Kirjallisuus

Seuraavat kirjat ovat hyviä C:n oppi– ja käsikirjoja:

- **Brian W. Kernighan, Dennis M. Ritchie**, *"The C Programming Language 2"*
- **ANSI X3J11**, *"Programming Language C"*
- **T. Plum**, *"Learning To Program In C"*
- **Kelley, Pohl**, *"A Book On C"*
- **T. Plum**, *"The Art of C Programming"*
- **K. Tizzard**, *"C For Professional Programmers"*

Muita C–ohjelmointiin liittyviä kirjoja:

- **Andrew Koenig**, *"C Traps and Pitfalls"*
- **Kernighan, Pike**, *"The UNIX Programming Environment"*
- **S. R. Bourne**, *"The UNIX System"*
- **J. Kay, B. Kummerfield**, *"C Programming In a UNIX Environment"*

C:n ohjelmointistandardeja:

- **Cannon et. al.**, *"Recommended C Style and Coding Standards"* Anonymous ftp monesta eri paikasta
- **GNU-projekti**, *"The GNU Coding Standard"* Anonymous ftp gnu.prep.ai.edu

C:n siirrettävyys:

- **A. Dolenc, A. Lemmke, D. Keppel**, *"Notes On Writing Portable Programs In C"* Anonymous ftp sauna.cs.hut.fi
- **M. Horton**, *"Portable C Software"*

Muuta suositeltavaa luettavaa on mm. **Kernighan, Plauger**, *"The Elements of Programming Style"* .

B Tiivistelmä

Alla on listattu tärkeimmät ohjeet, lähinnä kertauksen vuoksi. Nämä ohjeet ovat hyvin typistettyjä eikä poikkeuksia ja perusteluja ole otettu mukaan.

- Ohjelmointityylin on oltava konsistentti.
- Opettele koodin visualisointi: sisennykset, `{` ja `}`, välilyönnit, riveille jako ja tyhjät rivit.
- Kirjoita nimet ja kommentit samalla kielellä.
- Valitse kuvailevat, mutta ytimekkäät nimet.
- Lyhyitä konventionimiä tulee käyttää vain hyvin paikallisesti.
- Kommentoi maagiset luvut tai peitä ne makroilla tai luetelluilla tyypeillä.
- Käytä merkkien käsittelyyn `ctype.h`(3):ta, merkkijonojen käsittelyyn `string.h`(3):ta ja muistin käsittelyyn `memory.h`(3):ta.
- Käytä eri nollia tarpeen mukaan: `0` on luku, `'\0'` on merkki ja `NULL` on osoitin.
- Yksinkertaista ja siisti lausekkeita ja rakenteita.
- Pura liian syvät sisäkkäiset rakenteet eri funktioihin ja makroihin.
- Käytä valmiita rutiineja.
- Älä sotke bitti–, loogisia ja matemaattisia operaattoreita keskenään.
- Älä optimoi koodia — valitse parempi algoritmi.
- Kirjoita lyhyitä, yleiskäyttöisiä ja selkeästi määriteltyjä funktioita. Funktion maksimipituus on 50 riviä.
- Muista makrojen vaarat!
- Vältä `system()`:iä.
- Tarkista kutsujen paluuarvo!
- Vapauta varaamasi muisti.
- Älä viittaa vapautettuun muistiin tai varatun muistin ulkopuolelle.
- Tee ohjelmista käyttäjäystävällisiä ja toimintavarmoja.
- Käytä `assert()`:ia ohjelmointivirheiden löytämiseen ja muita mekanismeja ulkoisista syistä aiheutuvien virhetilanteiden käsittelemiseen.
- Älä sotke eri tyyppejä toisiinsa.

- Jaa ohjelma tiedostoihin. Maksimaalinen tiedoston pituus on noin 500 riviä.
- Käytä ohjelmankehitystyökaluja: `lex`, `yacc`; `gdb`.
- Käytä ohjelmistonhallintatyökaluja: `make`, `rcs`.
- Käytä hyväksesi UNIXin työkaluajattelua: `grep`, `sort`, `tr`, `sed`, `sh`, `awk`, `perl` jne.
- Älä kommentoi koodia — kommentoi koodin tehtävä, merkitys todellisuudessa, sen käyttämä tietorakenne ja algoritmi.
- Kirjoita selkeä manuaalisivu ja/tai käyttöohje.

```
/*␣Create␣a␣new␣string,␣which␣is␣the␣concatenation␣of␣given␣strings.
␣*␣Does␣NOT␣deallocate␣the␣given␣strings.
␣*/
char␣*concat(char␣*s1,␣char␣*s2)
{
␣␣char␣*res;

␣␣assert(*s1␣&&␣*s2);
␣␣res␣=␣malloc(strlen(s1)␣+␣strlen(s2)␣+␣1);␣/*␣1␣for␣the␣last␣'\0'␣*/
␣␣if␣(res␣==␣NULL)␣{
␣␣␣␣fatal("Out␣of␣memory␣in␣concat(\"%s\",␣\"%s\").\n",␣s1,␣s2);
␣␣␣␣exit(EXIT_SYSTEM_ERROR);
␣␣}
␣␣strcpy(res,␣s1);
␣␣strcat(res,␣s2);
␣␣return␣res;
}
```