Budapesti Műszaki Egyetem
Villamosmérnöki és Informatikai Kar

Deklaratív Programozás

Oktatási segédlet

Bevezetés a funkcionális programozásba

Negyedik, javított kiadás

Hanák D. Péter

Irányítástechnika és InformatikaTanszék

Budapest, 2001

Tartalomjegyzék

1. fejezet

Bevezetés

Ez a jegyzet oktatási segédletként a *Deklaratív programozás* (korábban *Programozási paradigmák)* c. tárgy funkcionális programozással foglalkozó részéhez készült.

1.1. A programozási paradigmákról

A programozási paradigma[1] viszonylag újkeletű szakkifejezés *(terminus technicus).* A paradigma az *Idegen szavak és kifejezések szótára*[2] szerint görög-latin eredetű, és két jelentése is van:

- bizonyításra vagy összehasonlításra alkalmazott példa;
- (nyelvtani) ragozási minta.

Az *Akadémiai Kislexikon*[3] a fentieken túl még egy, a mi szempontunkból fontos jelentését említi:

- valamely tudományterület sarkalatos megállapítása.

Programozási paradigmának nevezzük:

1. azt a módot, ahogyan a programozási alapfogalmakat felhasználják valamely programozási nyelv létrehozására; ill.
2. azt a programozási stílust, amelyet valamely programozási nyelv sugall.

A programozási paradigmáknak két alaptípusa van: *imperatív* és *deklaratív.*

1.2. A monolitikus és a strukturált programozásról

Minden programnak, legyen szó bármilyen stílusról, van valamilyen szerkezete. E tekintetben különbséget kell tennünk a *monolitikus* és a *strukturált (moduláris)* programozás között. A monolitikus program

- lineáris szerkezetű, azaz a programszövegben egymás után álló programelemeket rendszerint az adott sorrendben kell végrehajtani vagy kiértékelni, és nincsenek benne bonyolultabb adatszerkezetek;
- egyetlen fordítási egység, azaz változás esetén a teljes programszöveget újra kell fordítani.

[1] 1999-ig a most Deklaratív programozásnak nevezett tantárgynak Programozási paradigmák volt a neve.
[2] Akadémiai Kiadó, Budapest 1989
[3] Akadémiai Kiadó, Budapest 1990

Nyilvánvaló, hogy ilyen stílusban nem lehet nagyméretű programokat készíteni. A hatvanas évek közepén mozgalom indult a strukturált programozási elvek elfogadtatására, a megfelelő programozási nyelvek és fordítóprogramok kidolgozására és elterjesztésére, a szükséges elméleti és módszertani háttér kimunkálására. A strukturált, más néven moduláris programot elsősorban

- eljárások, függvények, biztonságos vezérlési szerkezetek,
- elemi és összetett adattípusok, absztrakt adattípusok, osztályok, objektumok, valamint
- önálló fordítási egységek, kapcsolatleírások, generikus programrészek megjelenése, használata jellemzi.

A több évtizedes tapasztalatok és kutatások megváltoztatták a programozás mibenlétéről kialakult képet: egyre nagyobb jelentőséget tulajdonítunk a *követelmények elemzésének,* a (formális) *specifikációnak,* a módszeres és szabványos *tervezésnek* és *dokumentálásnak,* a *programhelyességnek,* a *karbantartásnak,* a *módosíthatóságnak,* a *változáskövetésnek,* a *hordozhatóságnak,* a *minőségnek,* és egyre kisebbet magának a kódolásnak. E tekintetben itt elsősorban a *specifikáció* és a *megvalósítás,* a ***mit*** és a ***hogyan*** szétválasztásának fontosságát emeljük ki.

1.3. Az imperatív programozási paradigma

Az imperatív[4] (más néven procedurális) programozási paradigma a legelterjedtebb, a legrégibb; erősen kötődik a Neumann-féle számítógép-architektúrához. Két fő jellemzője a *parancs* és az *állapot.*

A program *állapottere* az a sokdimenziós tér, amelyet a program változóinak értelmezési tartománya határoz meg; a program pillanatnyi állapotát változóinak pillanatnyi tartalma írja le. A program állapotát *értékadással* – azaz a változók frissítésével – változtathatjuk meg. Állapotváltozás nélkül nem lehet, de legalábbis körülményes modellezni az időt, a valós világ jelenségeit.

Az imperatív paradigmán belül sajátos stílus jellemzi többek között

- a szekvenciális,
- a valós (azonos, ill. kötött) idejű,
- a párhuzamos és elosztott, valamint
- az objektum-orientált programozást.

A szekvenciális programozás mindennek az alapja, hiszen pl. bármely párhuzamos program szekvenciális programrészekből áll. A parancsokból, mint jól tudjuk, *vezérlési szerkezetek* – felsorolás, választás, ismétlés – felhasználásával összetett parancsokat, *absztrakcióval* pedig eljárásokat hozhatunk létre; ezért szoktunk az imperatív programozásról mint procedurális programozásról beszélni.

A valós idejű programozás erősen kötődik a párhuzamos és elosztott programozáshoz, ui. valós idejű rendszerekben egyes programrészeket rendszerint egyidejűleg, egymással párhuzamosan kell végrehajtani. Párhuzamos végrehajtásra ugyanakkor más esetekben, pl. numerikus számítások elvégzéséhez, aritmetikai kifejezések kiértékelésekor is szükség lehet.

A ma oly divatos objektum-orientált programozás is az imperatív programozási paradigma egyik válfaja, szoros rokonságban az *absztrakt adattípusokra* épülő programozással.

Imperatív stílusú programozás esetén a programozónak tudatosan törekednie kell a *mit* és a *hogyan* módszeres szétválasztására. A ma legelterjedtebb programozási nyelvek közül a FORTRAN, a COBOL és az eredeti Pascal alig, a (Turbo, Borland) Pascal és a C inkább, az Ada, a Modula, a C++ és a Java még inkább támogatja e szétválasztást. Azonban sikerüljön bármilyen jól a szétválasztás, a feladatot megoldó algoritmusok megírása a programozó dolga marad.

[4]Latin szó, jelentése: parancsoló (vö. imperativus, imperátor).

1.4. A deklaratív programozási paradigma

Az imperatív stílussal ellentétben a deklaratív[5] stílusban programozónak – elvileg – csak azt kell megmondania, hogy *mit* akarunk, az algoritmust az értelmező- vagy fordítóprogram állítja elő. A deklaratív programozás két válfaját szokás megkülönböztetni: a *logikai* és a *funkcionális* programozást.[6]

1.4.1. A logikai programozási paradigma

A programozási paradigmák közül, amint a neve is mutatja, a legerősebben kötődik a matematikai logikához. Jellemzői:

- a tények,
- a szabályok, és
- a következtetőrendszer.

A legelterjedtebb logikai programozási nyelv a Prolog. Professzionális, gyakorlati feladatok megoldására alkalmas megvalósításai a deklaratív nyelvi elemek mellett imperatív elemeket is tartalmaznak. Természetesen más logikai programozási nyelvek is vannak, pl. az OPS5, a CLP, a Mercury. (Az utóbbi a Prologtól átvett logikai programozási elemeket a típusfogalommal és a funkcionális programozást támogató nyelvi elemekkel egészíti ki.)

1.4.2. A funkcionális programozási paradigma

A funkcionális programozás két fő jellemzője

- az érték (a függvényérték is érték!) és
- a függvényalkalmazás.

A funkcionális programozás nevét a függvények kitüntetett szerepének köszönheti. A tisztán funkcionális programozási nyelvek a matematikában megszokott függvényfogalmat valósítják meg: *a függvény egyértelmű leképzés a függvény argumentuma és eredménye között,* a függvény alkalmazásának nincs semmilyen más hatása. Tisztán funkcionális programozás esetén tehát nincs állapot, nincs (mellék)hatás, nincs értékadás.

Funkcionális program pl. az `e e1` kifejezés, ahol az `e`-nek függvényértéket eredményező kifejezésnek kell lennie, az `e1` pedig tetszőleges kifejezés lehet. A matematikában megszokott módon azt mondjuk, hogy az `e` függvényt (vagy kissé körülményesebben: az `e` függvényértéket adó kifejezést) *alkalmazzuk* az `e1` argumentumra. Függvények alkalmazásáról lévén szó, funkcionális helyett szinonimaként gyakran *applikatív*[7] programozásról beszélünk.

Az applikatív programozás elmélete a λ-kalkulus (lambda-kalkulus), az a függvényelmélet, amelyet Alonzo Church az 1930-as években dolgozott ki, majd Moses Schönfinkel és Haskell Curry fejlesztett tovább. A λ-kalkuluson alapuló első funkcionális programozási nyelvet, a LISP-et (LISt Programming) John McCarthy dolgozta ki az 1950-es évek közepén, az 1960-as évek elején. A sokféle változat közül a professzionális célokra alkalmazható Common LISP a legismertebb. A LISP-dialektusok és modernebb utódjuk, a Scheme is típus nélküli nyelvek.

Az első típusos funkcionális nyelv az ML (Meta Language) egyik korai változata volt a 70-es évek közepén, amelyben R. Milner megvalósította típuselméleti eredményeit. Eredetileg *logikai állítások igazolására, tételbizonyításra* tervezték, erre utal a nem túl ötletes *Meta Language* elnevezés

[5] Ugyancsak latin szó, jelentése: kijelentő, kinyilatkoztató (vö. deklaráció).

[6] Vannak, akik a deklaratív programozást a logikaival azonosítják, és a funkcionális programozást nem tekintik deklaratívnak.

[7] Szintén latin szó, jelentése: alkalmazó (vö. applikáció).

is. A HOPE-pal és más funkcionális nyelvekkel szerzett tapasztalatok alapján dolgozták ki a Standard ML (SML) nyelvet a 80-as évek közepétől kezdve. Számos megvalósítása készült el különféle számítógépekre, és természetesen megjelentek különféle dialektusai is, pl. a Caml. A SML-családba tartozó nyelvek, kevés kivétellel, ún. *mohó kiértékelést,* azaz érték szerinti paraméterátadást alkalmaznak. Ez azt jelenti, hogy amikor egy függvénykifejezést alkalmazunk egy argumentumra, akkor az SML-értelmező először az argumentumot értékeli ki, és csak ezután lát hozzá a függvénykifejezés kiértékeléséhez.

A Miranda, az 1990-ben megjelent Haskell, és a még újabb Clean nyelv ezzel szemben *lusta kiértékelést* használ. A lusta kiértékelés az 1960-as években az Algol nyelvben alkalmazott *név szerinti paraméterátadás* modern leszármazottja; nem tévesztendő össze a Pascalban, a C-ben és más nyelvekben használt *cím szerinti paraméterátadással.* Megjegyzendő, hogy Lazy SML néven az SML-nek is van lusta kiértékelésű változata.

Az SML – akárcsak a körülményes szintaxisú, típus nélküli Common LISP – gyakorlati programozási feladatok megoldására készült, ezért nemcsak a tisztán funkcionális, hanem az imperatív stílusú programozáshoz szükséges nyelvi elemek is megtalálhatók benne: frissíthető változók, tömbök, mellékhatással járó függvények stb., továbbá a nagybani programozást segítő fejlett modulrendszere van.

1.4.2.1. A *read-eval-print* ciklus

Az SML-t, más deklaratív nyelvekhez hasonlóan, rendszerint *értelmezőprogrammal* (interpreterrel) valósítják meg: az értelmezőprogram a kifejezéseket *beolvassa* és *kiértékeli,* majd *kiírja* az eredményt, és azután ismét a beolvasással folytatja (ezt nevezik *read-eval-print* ciklusnak).

1.4.2.2. Hivatkozási átlátszóság

Az ún. *hivatkozási átlátszóság* (referential transparency) megléte vagy hiánya fontos jellemzője a programozási nyelveknek. Ha egy programnyelv, mint pl. az SML, rendelkezik ezzel a tulajdonsággal, akkor ez azt jelenti, hogy *egyenlők helyettesíthetők egyenlőkkel,* pl. egy kifejezés az értékével, az $E_1 + E_2$ kifejezés az $E_2 + E_1$ kifejezéssel (ahol a + jel a kommutatív aritmetikai összeadást jelenti).

A hivatkozási átlátszóság megléte esetén egy *kifejezés* értelme, *jelentése* egyszerűen a kiértékelésének az eredménye, és ezért egyes részkifejezéseit egymástól függetlenül lehet kiértékelni. Ezzel szemben egy *parancs* végrehajtása azt jelenti, hogy a program *állapota* megváltozik, vagyis a parancs megértéséhez meg kell érteni a parancs *hatását* a program teljes *állapotterére.*

1.4.2.3. A funkcionális program

A funkcionális program: mennyiségek közötti kapcsolatokat leíró egyenletek halmaza.

Pl. a `square(x) = x * x` megfelelő alakú egyenlet, ún. *kiszámítási szabály.* Ezzel szemben az `sqrt(x) * sqrt(x) = x` alakú egyenlet *csak deklarálja* a kívánt tulajdonságokat, kiszámításra nem, csupán *ellenőrzésre* alkalmas.

1.5. Egyszerű példák SML-ben

1.5.1. Négyzetre emelés

```
- fun square (x) = x * x;
   > val square = fn : int -> int
```

A `square` függvény típusa: `int -> int`. A `square : int -> int` kifejezést a `square` függvény *szignatúrájának* nevezzük. Alapértelmezés szerint az aritmetikai műveletekben az operandusoknak `int` a típusa.

1.5.2. Pénzváltás

Adott különböző címletű pénzérmék érték szerint csökkenő sorrendű listája, pl.

```
[20, 10, 5, 2, 1]
```

Most olyan SML-programot írunk, amely tetszőleges összeget apróra vált, és az eredményt ugyancsak listaként adja vissza! Feltesszük, hogy a megadott összeg mindig felváltható, azaz az érmék között van 1-es értékű. Az SML-program tanulmányozása előtt azt javasoljuk az olvasónak, hogy oldja meg a feladatot valamilyen általa ismert programozási nyelven.

A feladatot érdemes rekurzió alkalmazásával megoldani. Két esetet kell megkülönböztetnünk:

1. a felváltandó összeg 0,
2. a felváltandó összeg nem 0.[8]

Az 1. (triviális) esetben semmit nem kell tennünk, a feladat meg van oldva. A 2. esetben megpróbáljuk visszavezetni a feladatot egy már ismert részfeladatra.

```
- fun change(0, coins) = []
    | change(sum, coin :: coins) =
        if sum >= coin
        then coin :: change(sum - coin, coin :: coins)
        else change(sum, coins);
> val change = fn : int * int list -> int list
```

Nézzük a jelöléseket! A példában `fun`, `if`, `then`, `else`, `val` és `fn` a nyelv *kulcsszavai,* betűvel írt *terminális szimbólumai.* A `[]`, más néven `nil` az *üres lista* (üres sorozat) jele. A `|` (vonás) *változatokat* választ el egymástól. A `::` (négyespont) a bal oldalán álló elemet fűzi a jobb oldalán álló listához. A `fun` kulcsszó után a definiálandó függvény *neve* (most: `change`) áll, a nevet egy vagy több (most két) *paraméter* követi. Később látni fogjuk, hogy az SML-ben minden függvényt egyparaméteresnek tekintünk. A paraméterek megengedett értékei alapján *mintaillesztéssel* választunk az esetek közül. Ha a paraméter *konkrét érték* (pl. most 0), akkor az SML-értelmező az adott ágat csak akkor hajtja végre, ha a függvényt pontosan ilyen értékű argumentummal hívjuk meg. Ha a paraméter *név* (más szóval azonosító, most pl. `coins` vagy `sum`), akkor az tetszőleges *mintára* illeszkedik. A (`coin :: coins`) olyan *összetett minta,* amely legalább egyelemű (most: egész számokból álló) listára illeszkedik: `coin`-nak egy elemre, `coins`-nak egy – esetleg üres – listára kell illeszkednie. (Jelölésrendszerünket később egyszerűsíteni fogjuk.)

A program helyessége is könnyen belátható. Ha a felváltandó összeg 0, az eredmény az üres lista. Ha a felváltandó összeg nem 0, két további esetet kell megkülönböztetnünk az `if-then-else` *feltételes operátor* alkalmazásával. (Használhatnánk-e itt mintaillesztést? Használhatnánk-e feltételes kifejezést mintaillesztés helyett a 0 és a nem 0 esetek megkülönböztetésére?) Ha a felváltandó összeg (`sum`) nem kisebb a soron következő címletnél (`coin`), akkor ez jó érték, és be kell rakni annak az eredménylistának az elejére, amelyet úgy kapunk, hogy a maradék összeget (`sum - coin`) is megpróbáljuk felváltani ugyanilyen és nála kisebb értékű érmékkel (`coin::coins`). De ha a felváltandó összeg kisebb a soron következő címletnél, akkor ez nem jó érték, és a váltást a soron következő címlettel kell megpróbálni.

Jegyezzük meg, hogy a fenti függvénydefinícióban a változatok sorrendje nem közömbös, mivel a `sum` azonosító *minden* egész típusú mintára, így a 0 állandóra is illeszkedik. A kifejezések kiértékelési sorrendje – balról jobbra, fentről lefelé – garantálja, hogy ha az aktuális paraméter illeszkedik a 0 mintára, akkor a `sum` mintát tartalmazó változatra ne kerüljön sor.

[8]Feltesszük, hogy felváltandó összegnek csak nemnegatív számot adunk meg. Később látni fogjuk, hogy mit tehetünk a hibás bemeneti adatok kiszűréséért.

1.5.3. Legnagyobb közös osztó

Nézzük a jól ismert *euklideszi algoritmus* egy megvalósítását a legnagyobb közös osztó kiszámítására! Matematikai definíciója (feltesszük, hogy $0 \leq m \leq n$):

$$\begin{aligned}&\mathrm{gcd}(0, n) = n\\&\mathrm{gcd}(m, n) = \mathrm{gcd}(n \bmod m, m),\ \text{ha } m > 0\end{aligned}$$

Egy lehetséges kódolása Pascalban:

```
function gcd(m, n: integer): integer;
  var prevm: integer;
begin
  while m <> 0 do
  begin prevm := m; m := n mod m; n := prevm end;
  gcd := n
end (* gcd *);
```

és egy változata SML-ben:

```
fun gcd (m, n) = if m=0 then n else gcd(n mod m, m);
```

Az utóbbihoz hasonló programot Pascalban is lehet írni, csakhogy a Pascalban kevésbé hatékony a rekurzió megvalósítása, és több töltelékszöveget kell írnunk:

```
function gcd (m,n: integer): integer;
begin
  if m = 0
  then gcd := n
  else gcd := gcd (n mod m, m)
end;
```

SML-ben az eredeti matematikai definícióra nagyon hasonlító programot is írhatunk:

```
fun gcd(0, n) = n
  | gcd(m, n) = gcd(n mod m, m);
```

1.6. SML-értelmezők és fordítók

Az SML-nyelvnek két szintje van. A nyelv magját az *alapnyelv* (Core Language) képezi, a nagyobb programok írását a *modulnyelv* (Module Language) támogatja. A nyelvbe beépített elemeket gazdag és egyre bővülő *alapkönyvtár* (Basis Library) egészíti ki. A SML-nyelv és az alapkönyvtár definícióját legutóbb 1996-ban vizsgálták fölül.

Az első ML-értelmezőt 1977-ben írták az edinborough-i egyetemen. Az évek során számos értelmező és fordítóprogram készült el, egy részük licencköteles, más részük szabadon használható. Az utóbbiak körül kettőt ajánlunk az olvasó figyelmébe (mindkettő már az 1997-es, módosított definíciót követi):

- Az *mosml (Moscow SML)* legújabb, 2.x változata az *alapnyelvet* és a *modulnyelvet* teljes egészében megvalósítja, sőt az utóbbit új elemekkel egészíti ki. *Alapkönyvtára* is folyamatosan bővül, a már elkészült modulok kielégítik az 1997-es definíciót. Többek között linux, unix, Win95, Win98 és WinNT alatt használható, *kis erőforrásigényű* programrendszer.
- Az *smlnj (SML of New Jersey)* a teljes SML-nyelvet, azaz a szabványos *alapnyelv* mellett az ugyancsak szabványos *modulnyelvet,* valamint az 1997-es definíciónak megfelelő *alapkönyvtárat* valósítja meg. Többek között linux, unix, Win95, Win98 és WinNT alatt használható, *erőforrásigényes* programrendszer.

Az SML-nyelvvel most ismerkedők igényeinek a kevésbé erőforrásigényes *mosml* is mindenben megfelel.

Mindkét SML-értelmezőnek van egy nagy hátránya: a kezelői felületük írógépszerű, alig van mód az elütések javítására, és nincs lehetőség a korábban leírt sorok előhívására. Ezen az *emacs* szövegszerkesztő SML-progamozást támogató környezete segít, nevezetesen az SML-üzemmód. (A Prolog-értelmezők kényelmes használatához ugyancsak az *emacs*-ra, mégpedig az *emacs* Prolog-üzemmódjára van szükség.)

A funkcionális nyelvek általában interaktívak, megvalósításukra értelmezőprogramot (interpretert) írnak. Az SML-értelmezők és a programozók többek között a *készenléti jel,* a *folytatójel,* a *kiértékelőjel* és a *válaszjel* révén társalognak egymással. Az alábbi táblázatban a bal oldali oszlopban az *mosml,* ill. az *smlnj* által használt jeleket adjuk meg:

-	a sor elején álló *készenléti jel* (prompt): az SML új kifejezés begépelésére vár,
;	a bevitelt záró *kiértékelőjel:* hatására megkezdődik a kiértékelés,
=	a sor elején álló *folytatójel:* az SML a megkezdett kifejezés folytatására vagy lezárására (a kiértékelőjelre) vár (az *smlnj*-ben; az *mosml*-ben a folytatósort nem jelzi külön jel),
>	a sor elején álló *válaszjel:* az SML válaszát jelöli (az *mosml*-ben; az *smlnj*-ben a válaszsort nem jelzi külön jel).

Az *mosml* és az *smlnj* válaszai és főleg hibaüzenetei különbözhetnek egymástól. Ebben a jegyzetben rendszerint az *mosml* 2.0 verziójának válaszát és hibaüzeneteit adjuk meg, és utalunk rá, ha ettől valamilyen ok miatt eltérünk.

Az SML-ből kilépni a készenléti jelre adott többféle válasszal lehet:

- a `quit()` függvényhívással,
- az `exit` *`arg`* függvényhívással, ahol *`arg`* helyébe a kilépési kódot jelentő egész számot kell írni,
- MS-DOS alatt a ***ctrl-z,*** majd az ***enter*** leütésével,
- unix (linux) alatt a ***ctrl-d*** leütésével.

Az SML-értelmező kalkulátorként is használható, pl.

```
- 2+2;
> val it = 4 : int
- 3.2 - 2.3;
> val it = 0.9 : real
- Math.sqrt 2.0;
> val it = 1.41421356237 : real
```

`Math.sqrt` a `Math` könyvtárbeli `sqrt` függvényt jelöli. Egyes SML-könyvtárak tartalmát a D függelékben ismertetjük.

1.7. Információforrások

A funkcionális programozásnak magyar nyelvű irodalma alig van, az SML-nek – e jegyzeten és korábbi kiadásain kívül – egyáltalán nincs.

Az angol nyelvű könyvek közül különösen a következőket javasoljuk:

1. Richard Bosworth: *A Practical Course in Functional Programming Using Standard ML.* McGraw-Hill 1995. ISBN: 0-07-707625-7.
2. Lawrence C Paulson: *ML for the Working Programmer* (2nd Edition, ML97). Cambridge University Press 1996. ISBN: 0-521-56543-X (paperback), 0-521-57050-6 (hardback).
   `<http://www.cl.cam.ac.uk/users/lcp/MLbook/>`
3. Jeffrey D. Ullman: *Elements of ML Programming* (2nd Edition, ML97). MIT Press 1997.
   `<http://www-db.stanford.edu/~ullman/emlp.html>`
4. M.Felleisen, D.P.Friedman: *The Little MLer.* MIT Press, 1998
5. Chris Okasaki: *Purely Functional Data Structures.* Cambridge University Press, 1998. ISBN: 0-521-63124-6
6. Michael R. Hansen, Hans Rischel: *Introduction to Programming using SML.* Addison-Wesley, 1999. ISBN: 0-201-39820-6.
   `<http://www.it.dtu.dk/introSML>`

Az on-line információforrások közül javasoljuk a következőket:

1. *COMP.LANG.ML Frequently Asked Questions and Answers.*
   `<http://www.cis.ohio-state.edu/hypertext/faq/usenet/meta-lang-faq/faq.html>`
2. Andrew Cumming: *A Gentle Introduction to ML.*
   `<http://www.dcs.napier.ac.uk/course-notes/sml/manual.html>`
3. Stephen Gilmore: *Programming in Standard ML ´97: An On-line Tutorial.*
   `<http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/97/ECS-LFCS-97-364/>`

Az *mosml* és az *smlnj* megvalósítások honlapjának címe:

1. Moscow ML: `<http://www.dina.kvl.dk/~sestoft/mosml.html>`
2. Standard ML of New Jersey: `<http://cm.bell-labs.com/cm/cs/what/smlnj>`

1.8. Köszönetnyilvánítás

Köszönet illeti

- az *ML for the Working Programmer* c. könyv szerzőjét, *Lawrence C. Paulsont:* a könyvből sok érdekeset tanultam az SML-ről, többek között a jegyzetben bemutatott példák és megoldások jó része is ebből a könyvből származik;
- a *Moscow ML értelmező/fordító program* szerzőit, *Peter Sestoftot* és *Szergej Romanyenkót* az oktatási célra (is) kitűnő SML-megvalósításért.

1.9. Hibajelentés

A szerző köszönettel fogad a `hanak@inf.bme.hu` címre érkező bármilyen (sajtóhibákra, tartalomra vonatkozó) észrevételt a jegyzettel kapcsolatban.

2. fejezet

Nevek, függvények, egyszerű típusok

2.1. Értékdeklaráció

Deklaráció: valamilyen értéknek (pl. egésznek, valósnak, karakternek, füzérnek, függvénynek), típusnak, szignatúrának, struktúrának, funktornak stb. *nevet* adunk, ***kötést*** hozunk létre.[1] Az SML-ben a kötés ***statikus:*** fordítási időben jön létre a név és az érték között. (A futási időben létrejövő ***dinamikus*** kötés az objektum-orientált programozási nyelvek jellemzője.)

2.1.1. Névadás állandónak

Egy állandó lehet

- tartós állandó (pl. π, `pi`),
- átmeneti állandó (pl. valamilyen részeredmény).

SML-példák (az SML-értelmező válaszát nem minden esetben adjuk meg):

```
- val seconds = 60;
> val seconds = 60 : int
- val minutes = 60;
- val hours = 24;
- seconds * minutes * hours;
> val it = 86400 : int
```

A `60`, a `24` és a `86400` tovább nem egyszerűsíthető, ún. ***kanonikus*** kifejezések. Az SML-értelmező válasza minden esetben kanonikus kifejezés.

Van egy kitüntetett szerepű azonosító, az `it`, amely mindig a legfelső szintű kifejezés értékét veszi fel. A fenti kifejezéssorozat kiértékelése után pl. az `it` értéke 86400.

```
- it;
> val it = 86400 : int
- it div 24;
> val it = 3600 : int
```

`it` értékét elrakhatjuk későbbre, pl.

```
- val secsInHour = it;
> val secsInHour = 3600 : int
```

[1] Az SML-ben rendszerint nem teszünk olyan éles különbséget definíció és deklaráció között, mint a C-ben.

A nevekben a kis- és nagybetűk, a decimális számjegyek, az aláhúzás-jel (_) és a percjel (´, más néven felülvessző, aposztróf) használhatók. Jegyezzük meg, hogy az SML *különbséget tesz* a kis- és nagybetűk között!

2.1.2. Névadás függvénynek

Legyen[2]

```
- val pi = 3.14159;
- val r = 2.0;
```

akkor

```
- val area = pi * r * r;
> val area = 12.56636 : real
```

vagy függvényként

```
- fun area (r) = pi * r * r;
> val area = fn : real -> real
```

ahol `r` a (formális) paraméter, `pi * r * r` pedig a függvény törzse.

Az SML-ben a függvény maga is: **`érték!`** A `fun` definíció tulajdonképpen rövidítés, az értékdefiníció egy változata. Az `area` függvényt így is definiálhatjuk:

```
- val area = fn r => pi * r * r;
> val area = fn : real -> real
```

Talán meglepő, de az `fn r => pi * r * r` maga is kanonikus kifejezés, hiszen tovább nem egyszerűsíthető![3].

Egy függvény argumentumának típusa – mint halmaz – tartalmazza az *értelmezési tartományát* (domain), eredményének típusa – mint halmaz – pedig tartalmazza az *értékkészletét* (range).[4] Gyakran előfordul ugyanis, hogy

- az argumentum típusa által megengedett értékek egy részére a függvény nincs értelmezve (pl. az egész számokra értelmezett `div` függvény, ha 0 az osztója), vagy
- az eredmény típusa által megengedett értékek közül nem mindet állítja elő a függvény (pl. az egész típusú eredményt adó `sqrt`, amely csak nemnegatív eredményt állíthat elő).

A függvényt *leképezésnek*, transzformációnak (angolul mappingnek) is nevezzük. A függvény *típusa* adja meg, hogy milyen típusú értéket milyen típusú értékké képez le. Pl. az `area` függvény típusa: `real -> real`. Vegyük észre, hogy a függvény *eredményének* típusa nem azonos a *függvény* típusával!

Amikor az SML-ben egy függvényt egy argumentumra alkalmazunk, az argumentumot, ha kanonikus kifejezés, nem kell zárójelbe tenni. Helyesek tehát az alábbi példák:

```
- area(2.0);
- area 1.0;
- fun area r = pi * r * r
```

Az *állandókat* tekinthetjük függvényeknek is, mégpedig argumentum nélküli függvényeknek. A jól ismert unáris (egyoperandusú, monadikus) és bináris (kétoperandusú, diadikus) operátorok

[2] A `pi` állandó a `Math` könyvtárban is megvan.

[3] Az `fn` jelölésről részletesen egy későbbi fejezetben szólunk. Az `fn` jelet sokszor *lambdának* ejtjük, ami az eredetére (ti. a λ-kalkulusra) utal. λ-kalkulusbeli jelöléssel a fenti függvény: $\lambda r \bullet \pi \cdot r \cdot r$. A kétargumentumú szorzásfüggvény definíciója a λ-kalkulusban: $\lambda x \bullet \lambda y \bullet x \cdot y$, SML-jelöléssel: `fn x => fn y => x*y`.

[4] A típus és a halmaz rokonértelmű fogalmak: a típus határozza meg azoknak az értékeknek a halmazát, amelyeket az adott típusba tartozó azonosítók, nevek felvehetnek.

(műveleti jelek) szintén függvények. Az *unáris operátor* olyan függvény jele, amelynek *egyetlen* argumentuma (operandusa) van; az operátor az operandus előtt, ún. *prefix* helyzetben van. A *bináris operátor* olyan függvény jele, amelynek *két* argumentuma (operandusa) van; az operátor a két operandus között, ún. *infix* helyzetben van. Természetesen vannak kettőnél több operandusú műveletek is, például az `if-then-else`.

2.1.3. Nevek újradefiniálása

Tetszőleges értéknek adhatunk *nevet* az SML-ben. A név egy érték, esetleg egy másik név *szinonímája.* Név, szinonima helyett gyakran *azonosítóról,* ritkábban (matematikai értelemben vett) *változóról* beszélünk. Az utóbbi elnevezés egyes programozók számára félrevezető lehet, ugyanis az SML-beli "változók" másképpen viselkednek, mint az imperatív nyelvekből jól ismert társaik: *nem frissíthetők,* azaz nem kaphatnak új értéket a megszokott *értékadással.* Nem frissíthető változók esetén *érték-szemantikáról,* frissíthető változók esetén *hivatkozás-szemantikáról* beszélünk.

Ha az SML-ben egy azonosítót újradefiniálunk, az *nincs hatással* az azonosító korábbi alkalmazásaira: az értékdeklaráció *statikus* (és *nem dinamikus)* kötést hoz létre. Az alábbi példában hiába definiáljuk újra `pi`-t, az `area` függvény definíciójába a `pi` *korábbi értéke* (3.14259) van beépítve, és *nem a hivatkozás* a *pi* néven tárolt értékre.

```
- val pi = 0.0;
- area 1.0;
> val it = 3.14159 : real
```

Jegyezzük meg, hogy ha egy programban egy függvényt újradefiniálunk, az egész programot újra le kell fordítanunk, különben a változtatás *hatástalan maradhat.*

2.1.3.1. Nevek képzése

A nevek (azonosítók) tetszőleges hosszúságúak lehetnek. Az SML-ben *alfanumerikus* (azaz kis- és nagybetűkből, számjegyekből, aláhúzás-jelből, valamint percjelből) és *írásjelekből képzett* (azaz egyéb jelekből álló, angolul *symbolic)* neveket különböztetünk meg.

Az írásjelekből képzett nevekben 20-féle jel fordulhat elő:

```
!  % & $ # + - * / : < = > ?  @ \ ~ ´ ^ |
```

Egyes jelsorozatoknak különleges jelentésük van, ezeket *fenntartott azonosítóknak* vagy *szintaktikai jeleknek* nevezzük. Példák:[5]

```
- | = => -> #
abs val fun fn
int real list
+ - * / ~
```

Lássunk egy példát írásjelekből képzett nevek deklarálására!

```
- val +-+-+ = 1415;
> val +-+-+ = 1415 : int
```

Az SML-ben csak egyes belső függvények (`abs`, `+`, `*` stb.) neve többszörös terhelésű, a programozó nem definiálhat többszörösen terhelt neveket. Ez azért van így, mert az SML tervezői az automatikus *típuslevezetés (type inference)* megvalósítását fontosabbnak tartották a vele ütköző *többszörös terhelésnél (overloading).* A nevek többszörös terhelésének csökkent a jelentősége a moduláris programozás elterjedésével, hiszen a modulnevek (az SML-ben a struktúra- és a funktornevek) *szelektorként,* a nevek előtagjaként használhatók. Erről bővebben a moduláris programozásról szóló fejezetekben lesz majd szó.

[5] `int` típusnév, `list` típusoperátor, `real` pedig egyidejűleg típusnév is és függvénynév is. Jegyezzük meg, hogy ugyanaz a név *egyidejűleg* jelölhet értéket, típust, modult (struktúrát, ill. funktort), valamint rekordmezőt.

2.2. Egész, valós, füzér, karakter és más egyszerű típusok

Az SML-ben a más programozási nyelvekben megszokott egyszerű típusok neve: `int`, `real`, `char`, `string` és `bool`. Vannak további egyszerű típusok is, pl. `word`, `word8`, `order`, `unit` és `substring`.

A gyakran használt függvények be vannak építve a nyelvbe, ezeket *belső függvényeknek* (built-in functions) nevezzük. További gyakran használt függvények definícióját elindításakor olvassa be az SML-értelmező: ezek az *előre definiált függvények* (predefined functions) a *kezdeti környezet* (initial environment) részét képezik. Sok hasznos függvény található az *Alapkönyvtárban* (SML Basis Library).

Egyébként a `string` és `substring` típus átmenetet képez az egyszerű és az összetett típusok között, ugyanis vannak olyan belső műveletek, amelyek ilyen típusú értékekre mint *elemi értékekre* alkalmazhatók (pl. `=`, `<>`, `<`, `<=`, `>=`, `>`, `size`, `^`), de vannak olyanok is, amelyekkel ilyen értékek *összetevőin*, ill. *részein* végezhetünk műveletet (pl. `String.sub`, `String.substring`, `Substring.string`).

2.2.1. Egészek és valósak

A négy numerikus típus az `int`, a `word`, a `word8` és a `real`. `int` az előjeles, `word` és `word8` az előjel nélküli egész, `real` pedig a valós (tulajdonképpen racionális) számok típusa. A numerikus típusok gépi ábrázolása függ a megvalósítástól. A legnagyobb `int` típusú szám neve `Int.maxInt`, a legkisebbé `Int.minInt`. A `word` típusú értékek bitszámát `Word.wordSize` adja meg, a `word8` típusúak minden megvalósításban 8-bitesek. E négy típus közös, csak hivatkozásra használt megnevezéseit az alábbi táblázat mutatja.

megnevezés	*hivatkozott egyszerű típusok*
realint	`int, real`
wordint	`int, word, word8`
num	`int, real, word, word8`

2.2.1.1. A `real`, `floor`, `ceil`, `abs`, `round` és `trunc` függvény

A belső `real` függvény egész (`int`) értéket alakít át valóssá (`real`).[6] Az ugyancsak belső `floor` és `ceil` függvények valós szám egész részét adják eredményül: `f = floor r` az a legnagyobb egész, amelyre `real f <= r`, `c = ceil r` pedig az a legkisebb egész, amelyre `real c >= r`. Más szóval `floor` a $-\infty$, `ceil` pedig a $+\infty$ felé kerekít.[7] E három függvény típusa:

```
real  : int -> real
floor : real -> int
ceil  : real -> int
```

A két utóbbi függvény szemantikáját pontosabban az alábbi, *tulajdonságot definiáló egyenlőtlenséggel* írhatjuk le:

```
real(floor r) <= r < real(floor r + 1)
real(ceil r)  >= r > real(ceil r - 1)
```

Az SML-értelmező a `real` név begépelésére az alábbi választ adja:

```
> val it = fn : int -> real
```

[6]Emlékezzünk vissza: ugyanaz a név többféle dolgot jelenthet! A `real` név itt egyrészt egy típust, másrészt egy függvényt azonosít.

[7]A `floor` és `ceil`, valamint az alább ismertetett `round` és `trunc` függvényeket a `General` és a `Real` könyvtár definiálja. A `General` könyvtárban definiált `real` függvény azonos a `Real` könyvtárban definiált `fromInt` függvénnyel. Az ugyancsak alább ismertetett `abs` függvény egészekre alkalmazható változatát az `Int`, valósakra alkalmazható változatát a `Real` könyvtár definiálja.

Az `fn` szintaktikai jel jelzi, hogy `real` *függvényértéket* jelöl, az `int -> real` *típuskifejezés* pedig a függvény típusát adja meg.

Egy szám abszolút értékét a belső `abs` függvény állítja elő. Az `abs` azonosító többszörös terhelésű, mind `int`, mind `real` típusú értékre alkalmazható:[8]

```
abs : realint -> realint
```

Az SML-értelmező az `abs` név begépelésére az alábbi választ adja:

```
> val it = fn : int -> int
```

Ez azért van így, mert alapértelmezés szerint a többszörösen terhelhető nevek `int` típusú értéket várnak. `abs` szemantikája az alábbi *kiszámítási szabályként* is használható *egyenlettel* adható meg:[9]

```
abs x = max(x, ~x)
```

Kerekítésre a `round` és a `trunc` függvényt használhatjuk:

```
round : real -> int
trunc : real -> int
```

Szemantikájukat *kiszámítási szabályként* is használható *egyenlettel* írjuk le; `round` a 'legközelebbi" egész szám, `trunc` pedig a 0 felé kerekít:

```
round r = floor(r + 0.5)
abs(real(trunc r)) <= abs r
```

2.2.1.2. Alapműveletek előjeles egész számokkal

Az előjeles egész számokra alkalmazható alapműveleteket a következő táblázat foglalja össze.

jele	*jelentése*	*jellege*	*pozíciója*
`~`	negatív előjel	monadikus	prefix
`+`	összeadás	diadikus	infix
`-`	kivonás	diadikus	infix
`*`	szorzás	diadikus	infix
`div`	osztás, $-\infty$ felé tart	diadikus	infix
`mod`	`div` maradéka	diadikus	infix
`quot`	osztás, 0 felé tart	diadikus	infix
`rem`	`qout` maradéka	diadikus	infix

A kétoperandusú (diadikus, bináris) egészműveletek típusa: `int * int -> int`, az egyoperandusú (monadikus, unáris) egészműveleté: `int -> int`.

A következő táblázat `div` és `quot`, ill. `mod` és `rem` eredményének különbségére mutat példát.

operandusok		`div`	`mod`	`quot`	`rem`
5	3	1	2	1	2
5	~3	~2	~1	~1	2
~5	~3	1	~2	1	~2
~5	3	~2	1	~1	~2

[8] Jusson eszünkbe, hogy az alább használt *`realint`* megnevezés csak leírásokban fordulhat elő, az SML-értelmezők nem ismerik!

[9] `max` néven az `Int` és a `Real` könyvtárban is van egy-egy függvény, de az `abs` azonosítóval ellentétben `max` nem terhelhető többszörösen. Ha tehát a fenti egyenletet kiszámítási szabályként akarnánk használni, a `fun abs x = Int.max(x,~x)` és a `fun abs x = Real.max(x,~x)` definíciók közül a megfelelőt kellene alkalmaznunk. Jegyezzük meg, hogy az SML-ben a negatív előjel a `~` (tilde) és nem a `-` (mínusz).

- Ha a két operandus *azonos* előjelű, `div` és `mod`, ill. `quot` és `rem` eredménye azonos.
- Ha a két operandus *különböző* előjelű, `div` és `mod`, ill. `quot` és `rem` eredménye különböző.
- `a mod b` előjele `b` előjelével azonos.
- `a rem b` előjele `a` előjelével azonos.

A műveletek precedenciájáról, a `quot` és a `rem` műveletről további részletek a 2.2.1.4 szakaszban olvashatók.

2.2.1.3. Alapműveletek valós számokkal

Valós számok *fixpontos* és *lebegőpontos* alakban is megadhatók, pl.

```
0.01
~1.2E12
7E~5
```

Az `E` után tízes számrendszerben kell megadni a kitevőt pozitív vagy negatív egészként. A valós számokra alkalmazható alapműveleteket a következő táblázat foglalja össze.

jele	*jelentése*	*jellege*	*pozíciója*
`~`	negatív előjel	monadikus	prefix
`+`	összeadás	diadikus	infix
`-`	kivonás	diadikus	infix
`*`	szorzás	diadikus	infix
`/`	osztás	diadikus	infix

A kétoperandusú (diadikus, bináris) valósműveletek típusa: `real * real -> real`, az egyoperandusú (monadikus, unáris) valósműveleté: `real -> real`. A `~`, `+`, `-` és `*` műveleti jelek *többszörösen terhelt,* írásjelekből képzett nevek.

A `Math` könyvtár definiálja a gyakran használt aritmetikai állandókat és függvényeket, közülük a legfontosabbakat a következő táblázat mutatja.

név	*jelentés*	*típus*	*megjegyzés*
`pi`	a π állandó	`real`	1
`e`	az e állandó	`real`	2
`sqrt`	sqrt x = x négyzetgyöke	`real -> real`	
`sin`	sin x = x szinusza	`real -> real`	x radiánban
`cos`	cos x = x koszinusza	`real -> real`	x radiánban
`tan`	tan x = x tangense	`real -> real`	x radiánban
`exp`	exp x = e az x-ediken	`real -> real`	
`pow`	pow(x, y) = x az y-odikon	`real * real -> real`	3
`ln`	ln x = x természetes alapú logaritmusa	`real -> real`	$x > 0.0$
`log10`	log10 x = x 10-es alapú logaritmusa	`real -> real`	$x > 0.0$

Megjegyzések a táblázathoz:

1. Az egységnyi átmérőjű kör kerülete
2. A természetes alapú logaritmus alapja
3. `pow` alkalmazásának összetett a feltétele: $y \geq 0.0 \wedge (\text{integral } y \vee x \geq 0.0) \vee y < 0.0 \wedge ((\text{integral } y \wedge x \neq 0.0) \vee x > 0.0)$, ahol integral x SML-beli megvalósítása pl. a `real(round x) = x` feltétel lehet. integral x olyan valós számra teljesül, amelynek „nincs” törtrésze.

Bármely függvényalkalmazás (más szóval az összes monadikus műveleti jel, azaz az összes *prefix* operátor) erősebben köt a diadikus műveleti jeleknél (más szóval az infix operátoroknál). Ezért pl. $\exp\ a + b = (\exp\ a) + b \neq \exp\ (a + b)$.

2.2.1.4. Precedencia

Az alábbi táblázat az *infix* pozícióban használható aritmetikai és relációs operátorok típusát és precedenciáját mutatja az SML-ben.

A precedenciát 0 és 9 közötti egész számokkal adjuk meg (a 9-es szint a legmagasabb). A nagyobb precedenciájú operátor erősebben köt.

operátor	*típus*	*megjegyzés*
	7-es szintű precedencia	
`*`	*`num * num -> num`*	1
`/`	`real * real -> real`	1
`div, mod`	*`wordint * wordint -> wordint`*	1
`quot, rem`	`int * int -> int`	1, 2
	6-os szintű precedencia	
`+, -`	*`num * num -> num`*	1
	4-es szintű precedencia	
`=, <>`	`''a * ''a -> bool`	1, 4, 5
`<, <=, >=, >`	*`numtxt * numtxt`* `-> bool`	1, 3, 4

Megjegyzések a táblázathoz:

1. `quot` és `rem` *prefix* változatát az `Int` könyvtár definiálja, a többi operátor *infix* pozíciójú és a kezdeti környezet része.

2. `quot` és `rem` használatához be kell tölteni az `Int` könyvtárat: `load "Int"`. Betöltés után vagy a *hosszú nevükkel* hivatozhatunk rájuk (`Int.quot`, `Int.rem`), vagy láthatóvá tehetjük az `Int` könyvtárban definiált összes nevet (`open Int`), vagy pedig értékdeklarációval új nevet adhatunk a két fügvénynek (`val quot = Int.quot; val rem = Int.rem`). Ha a két függvényt *infix* pozícióban akarjuk használni, alkalmazni kell rájuk az `infix` deklarációt (ajánlott precedenciaszintjük a 7-es): `infix 7 quot rem`.

3. A *`numtxt`* megnevezés a *`num`*-csoportba tartozó típusokon kívül még a `char` és a `string` típust jelenti (részletek a fejezet hátralévő részében).

4. `bool` típusú érték az igazságérték (részletek a fejezet hátralévő részében).

5. Az `''a` ún. típusváltozó, és olyan értekek típusát jelöli, amelyekre az *egyenlőségvizsgálat* elvégezhető (a részleteket egy későbbi fejezet ismerteti). Jegyezzük meg, hogy valós számok egyenlőségét nem lehet tesztelni az SML-ben, és más nyelv használata esetén sem célszerű, mert az ábrázolás pontatlansága, a kerekítési hibák miatt még azonosnak vélt értékekre is hamis eredményt adhat a vizsgálat. Két valós szám egyenlősége helyett azt kell megvizsgálni, hogy a különbségük kisebb-e egy elegendően kicsi valós számnál. (Az egyenlőségvizsgálat kérdésében nem egységesek az SML-megvalósítások: pl. az *mosml* megengedi, az *smlnj* nem engedi meg, hogy az `=` és a `<>` operátort valós számokra alkalmazzuk.)

6. Vannak más típusokra alkalmazható *infix* operátorok is. Ezek precedenciájáról később lesz szó.

2.2.1.5. Alapműveletek előjel nélküli egészekkel

Az előjel nélküli egészek típusa az SML-ben: `word`, ill. `word8`. `word` bitszáma függ a megvalósítástól, `word8` minden megvalósításban 8 bites. Jelölésükre lássunk néhány példát!

```
0w241
0wxf1
0wxF1
```

`0w` (a nulla után kis w áll!) jelöli, hogy `word` vagy `word8` típusú értékről van szó. A `0w` után decimális vagy hexadecimális egésznek kell jönnie. A hexadecimális számot kis `x` betűvel kell kezdeni, jegyei 0 és 9 között számjegyek, továbbá A és F (vagy a és f) közötti betűk lehetnek. Alapértelmezés szerint az így megadott állandók `word` típusúak. `word8` típusú érték előállításához *típusmegkötést* kell alkalmaznunk (l. a 2.2.1.6 szakaszt). Vegyük észre, hogy nemcsak nevek, hanem *állandók* is lehetnek többszörösen terhelve: az SML-ben ilyenek a `word` és `word8` típusú állandók, a C-ben és a Pascalban ilyen az egész értékű `int` vagy `real` típusú számok jelölése.

`word`, `word8`, `int` és `char` típusú értékek konverziójára a `Word` és `Word8` könyvtárbeli függvények használhatók.

2.2.1.6. Típusmegkötés

Az SML a legtöbb kifejezés típusát le tudja vezetni a kifejezésben előforduló értékek (nevek, állandók) típusából. A *típuslevezetés* (type inference) csak néhány *többszörösen terhelhető nevű* belső függvény esetében nem lehetséges. Ilyenkor, ha az alapértelmezéstől el akarunk térni, *típusmegkötést* kell használni. (Mint tudjuk, alapértelmezés szerint a többszörösen terhelhető nevek `int` típusú értéket várnak.) Pl.

```
- fun sq x = x * x;
> val sq = fn : int -> int
```

Egy kifejezés vagy részkifejezés típusát úgy köthetjük meg, hogy utána kettősponttal elválasztva megadjuk a típusnevet: *kifejezés : típusnév*. Példák:[10]

```
- val mask = 0wx7F : Word8.word;
- fun sq´r (x : real) = x * x;
- fun sq´r x : real = x * x;
- fun sq´r x = (x * x) : real;
- fun sq´r x = x * (x : real);
- fun sq´r x = x * x : real;
- val sq´r : real -> real = fn x => x*x;
- val sq´r = (fn x => x*x) : real -> real;
```

Zárójelezéssel szabhatjuk meg, hogy a típusmegkötés mire vonatkozzon. a `:` *operátor* precedenciája kisebb, mint a függvényalkalmazásé vagy az aritmetikai műveleteké.

Az `sq´r` névben az `´r` csak emlékeztet arra, hogy az `sq` (*square*, négyzet) függvény e változatát valós számokra lehet alkalmazni. A konvenció használata nem kötelező, csak célszerű. Az egész számokra alkalmazható változat neve e konvenció szerint `sq´i` lehetett volna.

2.2.2. Füzérek

A `string` típusú füzért a szokásos módon, idézőjelek között álló, esetleg egyetlen karaktert sem tartalmazó karaktersorozattal jelöljük az SML-ben, például

```
- "abraka";
```

[10] Az *mosml*-ben a `Word8.word` típusnév csak akkor használható, ha a `Word8` könyvtár be van töltve. `Word8` betöltése nélkül is használható az *mosml*-ben a `word8` típusnév, használatát azonban nem javasoljuk, mert az ilyen program csak módosítás után futtatható az *smlnj* alatt.

```
> val it = "abraka" : string
- "z" (* egyetlen karakter *);
> val it = "z" : string
- " " (* szóköz *);
> val it = " " : string
- "" (* üres füzér *);
> val it = "" : string
```

2.2.2.1. Escape-szekvenciák

Különleges karakterek beírására (a C nyelvből is jól ismert) *escape-szekvenciák* használhatók, ezeket az alábbi táblázatban soroljuk föl. Egyes dolgok megértéséhez tudni kell, hogyan jelöli a karaktereket az SML, és mit csinál az `ord` függvény. Ezekről a kérdésekről a 2.2.3 szakaszban lesz szó.

jelölés	*escape-szekvencia megnevezése*	*decimális ASCII-kódja*	*megjegyzés*
`\a`	csengő (alert)	7	
`\b`	visszalépés (backspace)	8	
`\t`	vízszintes (horizontális) tabulátor	9	
`\n`	újsor-jel (newline)	10	
`\v`	függőleges (vertikális) tabulátor	11	
`\f`	lapdobás (form feed)	12	
`\r`	sorelejére-jel (return)	13	
`\^c`	vezérlőkarakter	`ord c - 64`	1
`\ddd`	tetszőleges karakter	`ddd`	2
`\"`	idézőjel (double quote)	34	
`\\`	hátratört-vonal (backslash)	92	
`\w...w\`	figyelmen kívül hagyandó		3

Megjegyzések a táblázathoz:

1. A `c` helyébe olyan karakter jele írható, amelyre `#"@"<= c < #"_"`.
2. `ddd` háromjegyű decimális számot jelöl. Mindhárom számjegyet ki kell írni, a vezető nullákat is.
3. Az olyan sorozatot, amelyben `w` helyén egy vagy több szóköz jellegű formázó karakter (szóköz, tabulátor, lapdobás, újsor stb.) áll, az SML nem veszi figyelembe.[11]

2.2.2.2. Gyakori műveletek füzérekkel

Két füzért kapcsol egybe a `string * string -> string` típusú `^` (felfelé mutató nyíl, kalap, circumflex) *infix* operátor, füzér hosszát adja eredményül a `string -> int` típusú `size` függvény:

```
- "alma" ^ "fa";
> val it = "almafa" : string
```

[11] Ha egyetlen sorba akarunk kiíratni egy olyan füzért, amely a begépeléskor több sorban fér csak el, *folytatósort* (continuation line) használunk: a formázó karakterek elé és mögé egy-egy hátratört-vonalat (\) rakunk, pl.

```
- "abrakad\
\abra";
> "abrakadabra" : string
```

A folytatósort nyitó és záró hátratört-vonalak között formázó karakter nem adható meg escape-szekvenciaként! A következő példában az SML a `\t`-ből a `t`-t betű szerint, a `\a` párt pedig csengőjelnek veszi:

```
- "abrakad\
\t \abra";
```

```
- size it;
> val it = 6 : int
- size "almafa";
> val it = 6 : int
- size "";
> val it = 0 : int
```

`string` típusú értékek összehasonlítására használhatók a szokásos relációs operátorok (`<`, `<=`, `=`, `>=`, `>`, `<>`).

2.2.3. Karakterek

Az SML-ben a karaktertípust a `char` típusnévvel jelöljük.

Az SML-ben a karakterjelölés elég hosszadalmas: egyetlen karakterből álló füzérállandó, amely előtt `#` áll. Ez azért van így, mert eredetileg nem volt karakterjelölés az SML-ben, karakter helyett az egyetlen karakterből álló füzért használták. Példák:

jelölés	*magyarázat*	*ASCII-kód*
`#"a"`	a kis *a* betű	97
`#"Z"`	a nagy Z betű	90
`#"0"`	a 0 számjegy	48
`#"^G"`	a `^G`-vel jelölt vezérlőkarakter	7
`#"\007"`	a 007 kódú karakter	7
`#"\a"`	a csengő jele	7

2.2.3.1. Gyakori műveletek karakterekkel

Karakter ASCII-kódját állítja elő az `ord`, adott ASCII-kódú karaktert ad vissza a `chr` függvény (`ord` típusa `char -> int`, `chr` típusa `int -> char`), mindkettő belső függvény. `ord` 0 és 255 közötti értéket állít elő. `chr` csak 0 és 255 közötti értékekre alkalmazható, más argumentumra az SML-értelmező hibát jelez.[12] Példa:

```
- fun digit i = chr(i + ord #"0");
> val digit = fn : int -> char
```

`char` típusú értékek összehasonlítására használhatók a szokásos relációs operátorok (`<`, `<=`, `=`, `>=`, `>`, `<>`).

2.2.4. Igazságértékek, logikai kifejezések, feltételes kifejezések

Az SML igazságérték-típusa (`bool`) ugyanaz, mint pl. a Pascal `boolean` típusa. Csupán kétféle `bool` típusú állandó van, jelölésük: `true` és `false`.

Igazságértéket adnak eredményül a relációs operátorok:

- előjeles és előjel nélküli egészekre, valósakra, karakterekre, füzérekre: `<=`, `<`, `>`, `>=`,
- az ún. *egyenlőségi típusokra* (equality types): `=`, `<>`.[13]

A `bool` típusú értéket eredményül adó kifejezést gyakran nevezik *predikátumnak*.

[12] Vegyük észre, hogy a `chr` függvény esetén az *értelmezési tartomány* csak részhalmaza az argumentum típusának, az `ord` függvény esetén pedig az *értékkészlet* szintén csak részhalmaza az eredmény típusának.

Függvénynek az olyan leképzést nevezzük, amely az értelmezési tartomány minden elemének az értékkészlet pontosan egy (egymástól nem feltétlenül különböző) elemét felelteti meg . Ha ez nem teljesül, azaz ha az értelmezési tartomány egy-egy elemének több elem is megfeleltethető az értékkészletben, akkor a leképzést *relációnak* hívjuk.

[13] Mint említettük, a valós számok közötti egyenlőség- és egyenlőtlenségvizsgálat könnyen hibás eredményhez vezet, ezért az *smlnj* a `real` típust nem tekinti egyenlőségi típusnak.

2.2.4.1. Feltételes operátor

A *feltételes kifejezés* az SML-ben

```
if E then E1 else E2
```

alakú, ahol

- az `E` logikai kifejezés `bool` típusú értéket ad eredményül,
- az `E1` és `E2` kifejezések tetszőleges, de egyforma típusú értéket adnak eredményül, és
- az `else` ág sohasem maradhat el.

Példa:

```
- fun sign n =
        if n > 0 then 1
        else if n = 0 then 0
        else ~1;
```

A feltételes kifejezés `if-then-else` operátora, mint látjuk, *három* operandusú. Az operandusokat az SML-értelmező *lustán* értékeli ki. Ez annyit jelent, hogy az `E1`, ill. az `E2` kifejezés kiértékelésére csak akkor kerül sor, ha az `E` kiértékelésének `true`, ill. `false` az eredménye.

2.2.4.2. Logikai operátorok

Az SML-ben logikai kifejezésekben három logikai operátort alkalmazhatunk, ezek a kétoperandusú `andalso` és `orelse`, valamint az egyoperandusú `not`. `andalso` és `orelse` *lusta* kiértékelésű: ha a bal oparandus kiértékelése elég az eredmény meghatározásához, az SML-értelmező a jobb operandust egyáltalán nem értékeli ki.[14]

A lusta kiértékelésű operátorok hasznosak pl. tömbök indexhatárának vagy listák végének a kezelésére (hogy az utolsó utáni elem feldolgozására már ne kerüljön sor), a 0-val való osztás elkerülésére (amikor egy változó értéke 0 is lehet) stb.

Jegyezzük meg, hogy `orelse` precedenciája kisebb `andalso` precedenciájánál, és mindkettőé kisebb bármely más *infix* operátor precedenciájánál. Ezzel szemben a `not` precedenciája a lehető legnagyobb, a többi prefix helyzetű egyoperandusú operátorhoz (más szóval: függvényjelhez) hasonlóan.

`andalso` és `orelse` felhasználásával definiálhatjuk például a logikai *konjunkciót* és *alternációt* (diszjunkciót) megvalósító függvényeket `&&&`, ill. `|||` néven:

```
- fun &&& (b, j) = b andalso j;
- fun ||| (b, j) = b orelse j;
```

A lényeges különbség `andalso` és `&&&`, ill. `orelse` és `|||` között nem az, hogy `andalso` és `orelse` *infix,* `&&&` és `|||` pedig *prefix* helyzetben használandók, hanem az, hogy az előbbiek *lusta,* az utóbbiak pedig *mohó* kiértékelésűek! Hiába lenne eldönthető a teljes kifejezés eredménye `&&&`, ill. `|||` első argumentuma alapján, az SML-értelmező kiértékelésük előtt a második argumentumukat is kiértékeli (további részletek a 4.2 szakaszban).

Természetesen e két függvényt *infix operátorként* is használhatjuk, ha *infix* voltukat deklaráljuk, például így (l. még a 4.1 szakaszt):

```
- infix 2 &&&;
- infix 1 |||;
```

[14] Más nyelvek a lusta kiértékelésű `andalso` és `orelse` operátort *shortcut* operátornak nevezik, és például *cand* és *cor* néven emlegetik. A C-ben e két operátor jele: `&&` és `||`.

Ha `E`, `E1` és `E2` egyaránt `bool` típusú kifejezések, `if E then E1 else E2` helyett írhatjuk, hogy `E andalso E1 orelse not E andalso E2`[15], mert `andalso` és `orelse` is lusta kiértékelésűek, de helyettük nem használhatnánk a most definiált `&&&` és `|||` operátorokat.

2.2.4.3. Tesztelő függvények

Predikátumot használunk annak eldöntésére, hogy bizonyos értékek adott feltételt kielégítenek-e. Az ilyen célra használt predikátumot gyakran tesztelő függvénynek nevezzük. A `Char` könyvtárban például sok olyan jól használható függvény van, amelyek karakterek osztályozását teszik lehetővé. Ilyen függvényeket persze saját magunk is definiálhatunk. Nézzünk néhány példát!

```
- fun isLower s = #"a" <= s andalso s <= #"z";
- fun isUpper s = #"A" <= s andalso s <= #"Z";
- fun isLetter s = isLower s orelse isUpper s;
```

A `Char` könyvtár leghasznosabb tesztelő függvényeit az alábbi táblázatban soroljuk föl. Ezek a függvények mind `char -> bool` típusúak.

A függvények szemantikáját a `contains : string -> char -> bool` függvény segítségével adjuk meg.[16] `contains s c` akkor igaz, ha a `c` karakter benne van az `s` füzérben. (A `contains` függvényt ugyancsak a `Char` könyvtár definiálja.)

függvénynév	*arg.*	*jelentés*
`isLower`	`c`	`contains "abcdefghijklmnopqrstuvwxyz" c`
`isUpper`	`c`	`contains "ABCDEFGHIJKLMNOPQRSTUVWXYZ" c`
`isDigit`	`c`	`contains "0123456789" c`
`isAlpha`	`c`	`isUpper c orelse isLower c`
`isHexDigit`	`c`	`isDigit c orelse contains "abcdefABCDEF" c`
`isAlphaNum`	`c`	`isAlpha c orelse isDigit c`
`isPrint`	`c`	`c` látható karakter vagy szóköz (`#" "`)
`isSpace`	`c`	`contains " \t\r\n\v\f" c`
`isPunct`	`c`	`isPrint c andalso not(isSpace c orelse is alphaNum c)`
`isGraph`	`c`	`not(isSpace c) andalso isPrint c`
`isAscii`	`c`	`0 <= ord c <= 127`
`isCntrl`	`c`	`not(is Print c)`

[15] Az utóbbi, kirakva a zárójeleket, így értendő: `(E andalso E1) orelse ((not E) andalso E2)`.

[16] A `string -> char -> bool` típuskifejezés jelentését a részlegesen alkalmazható függvényekről szóló fejezetben magyarázzuk meg.

3. fejezet

Párok, ennesek, rekordok

3.1. Pár, ennes

A `(firstname, lastname)` egy pár, a `(day, month, year)` egy hármas stb. Az *ennes* (angolul: *tuple*) elemeit vessző választja el, az elemek típusa tetszőleges, sorrendjük fontos.[1] Egy ennes elemei különböző típusúak lehetnek. Példák:

```
- ("Laca", 18);
> val it = ("Laca", 18) : string * int
- (18, "Laca");
> val it = (18, "Laca") : int * string
```

Mivel az elemek sorrendje fontos, a fenti két pár különböző típusú.

A *rekord* olyan ennes, amelyben az egyes elemeket megcímkézzük: a sorrend többé nem fontos, az elemekre nem a helyük szerint, hanem a címkéjükkel hivatkozunk. A rekordról a 3.2 szakaszban lesz szó.

Típuskifejezés. `string * int` és `int * string` speciális kifejezések, ún. *típuskifejezések.* A típuskifejezés elemei *típusállandók* (`int`, `real`, `string` stb.), *típusváltozók* (`´a`, `´b` stb.), *típusoperátorok* (`*`, `->` stb.) és más típuskifejezések lehetnek. A típusoperátoroknak is van precedenciájuk: a *keresztszorzat* (Descartes-szorzat, jele a `*`) precedenciája nagyobb a *leképzés* (jele a `->`) precendenciájánál. Típuskifejezésben is használható *zárójel* a műveletek sorrendjének meghatározására.[2]

Látni fogjuk, hogy az eddig megismerteken kívül vannak más típusállandók és típusoperátorok is, sőt a programozó maga is definiálhat típusállandókat, típusváltozókat és típusoperátorokat.

3.1.1. Példa: vektorok

A bemutatott példákban a vektor `real * real` típusú.

Az (x, y) vektor hossza $\sqrt{x^2 + y^2}$, ellentettje $(-x, -y)$.

```
- val zerovec = (0.0, 0.0);
- val a = (1.5, 6.8);
- val b = (3.6, 0.9);
- fun lengthvec (x, y) = Math.sqrt (x * x + y * y);
> val lengthvec = fn : real * real -> real
```

[1]Később, a 3.1.4 szakaszban látni fogjuk, hogy kitüntetett szerepe van az egyetlen elemet sem tartalmazó ennesnek, a `()` jellel jelölt *nullasnak.* Az egyelemű ennes a jól ismert zárójeles kifejezés; ha nem okoz félreértést, a zárójel elhagyható. Kételemű ennes a *pár,* háromelemű a *hármas* stb.

[2]A típuskifejezés fogalma kevésbé új, mint első hallásra gondolnánk: pl. a `TYPE mmm = ARRAY [...] OF ...` Pascal-deklaráció jobb oldala típuskifejezés, bár a Pascalban ritkán használják ezt a fogalmat.

```
- lengthvec a;
> val it = 6.96347614342 : real
- lengthvec (1.0, 1.0);
> val it = 1.41421356237 : real
- fun negvec (x, y) = (~x, ~y) : real * real;
> val negvec = fn : real * real -> real * real
- negvec b;
> val it = (~3.60000, ~0.90000) : real * real
```

Típusdeklarációval új nevet is adhatunk egy típusnak:

```
- type vec = real * real;
> type vec = real * real
```

A `type` kulcsszóval bevezetett típusdeklaráció *gyenge absztrakció*, hiszen csak *új nevet* ad egy már *létező* adattípusnak, nem hoz létre új adattípust. A `vec` név a `real * real` típuskifejezés *szinonímája.*[3]

3.1.2. Függvény több argumentummal és eredménnyel

Nézzük a következő definíciót:

```
- fun average (x, y) = (x + y) / 2.0;
```

Szemlélet kérdése, hogy az `average` függvénynek két argumentuma van-e, vagy csak egy, nevezetesen egy pár.

Az SML szemléletmódja szerint minden függvénynek *egyetlen* argumentuma és *egyetlen* eredménye van, egy-egy *ennes*. Ez azért jó, mert egyszerű.

```
- fun addvec ((x1, y1), (x2, y2)): vec =(x1 + x2, y1 + y2);
```

Az *mosml* válasza:

```
- val addvec = fn : (real * real) * (real * real) -> real * real
```

A válaszban a `vec` típusnév helyett a vele egyenértékű `real * real` típuskifejezés áll! Az *mosml* tervezői ezzel is tudatosítani akarják, hogy a `vec` név csak *szinoníma,* nem új típus. Az *smlnj* válasza kicsit más:

```
- val addvec = fn : (real * real) * (real * real) -> vec
```

Mivel a programozó `addvec` eredményét `vec` típusúnak deklarálta, válaszában az *smlnj* fordító is a `vec` szinonímát írja ki az eredmény típusaként.

A `real * real` típuskifejezés másik két előfordulását azonban egyetlen SML-értelmező sem helyettesítheti a `vec` névvel, mert nem tudhatja, hogy az adott `real * real` típuskifejezésnek van-e köze a `vec` típusnévhez.

3.1.2.1. Gyakorló feladatok

1. Hány argumentuma van az `addvec` függvénynek? 1, 2 vagy éppen 4?
2. Definiáljon `subvec` néven olyan függvényt, amely két vektor, `v1` és `v2` különbségét állítja elő! A függvény típusa legyen `subvec : vec * vec -> vec`.
3. Definiáljon `scalevec` néven olyan függvényt, amely az `r` valós számmal megszorozza a `v` vektort ! A függvény típusa: `scalevec : real * vec -> vec`, deklaratív specifikációja: `scalevec(r, v)` = az `r` skalár és a `v` vektor szorzata.

[3] Új adattípus létrehozására, azaz *erős absztrakcióra* kétféle lehetőség is van az SML-ben. Az egyik a sokféleképpen használható `datatype` deklaráció, amellyel az SML modulszerkezetébe (`structure` és `signature`) rejtve valósíthatunk meg erős absztrakciót. A másik lehetőség a már idejétmúltnak tekinthető `abstype` deklaráció. Minderről később lesz szó a jegyzetben.

3.1.3. Ennes elemeinek kiválasztása mintaillesztéssel

Egy ennes elemeit elegánsan *mintaillesztéssel* azonosíthatjuk. Példa:

```
- val (xc, yc) = scalevec (4.0, a);
> val xc = 6.0 : real
> val yc = 27.2 : real
```

Az `(xc, yc)` pár *illeszkedik* `scalevec` eredményére: `xc` a pár bal, `yc` pedig a jobb oldali tagjára. Az értékdeklarációban alkalmazott minta éppolyan összetett lehet, mint az argumentum-minta a függvénydeklarációban.

3.1.4. A nullas

A *nullas* (angolul *0-tuple)* olyan ennes, amelynek egyetlen eleme sincs. Az SML-ben a nullast a `()` jellel jelöljük. Az angol szakirodalom néha *hermit*-nek, remetének is nevezi, ugyanis ez az egyetlen eleme a `unit` (az ALGOL68 és a C nyelv szóhasználata szerint: *void)* típusnak.

A `unit` típus a típusműveletek *egységeleme.* Olyan esetben használjuk, amikor a függvénynek nincs argumentuma, vagy amikor függvényt a *mellékhatása* miatt alkalmazzuk, mert nincs felhasználandó eredménye.

3.1.4.1. A `print`, a `use` és a `load` üggvény

Ebben a szakaszban három olyan függvényt mutatunk be, amelynek `unit` típusú az eredménye.

```
- print;
> val it = fn : string -> unit
```

A `print`-et akkor használjuk, amikor valamit ki kell íratni a képernyőre. `print` nem igazi függvény, inkább imperatív stílusú eljárás, amely maradandó változást okoz a környezetben (ui. megváltozik a képernyő tartalma). Nézzünk további példákat:

```
- use;
> val it = fn : string -> unit
```

A `use` *forrásprogramok* betöltésére szolgál az interaktív SML-környezetben, pl. `use "x.sml"`. Jegyezzük meg, hogy a típusjelzést (célszerűen a`.sml`-t) mindig ki kell írni.

```
- load;
> val it = fn : string -> unit
```

A `load`-dal a már lefordított *tárgyprogramokat* tölthetjük be az interaktív *mosml*-környezetben (az *smlnj* másképpen kezeli, ezért ott másképpen kell betölteni a modulokat). Pl. `load "Math"` a `Math.uo` nevű könyvtári modult, `load "x"` az `x.sml` program lefordított változatát, `x.uo`-t tölti be. Jegyezzük meg, hogy `load` a `.uo` névkiterjesztést tételezi fel, akár kiírjuk, akár nem.

3.1.4.2. Mi a különbség?

Nézze meg figyelmesen az alábbi példákat, és magyarázza el, hogy mi a különbség közöttük!

```
- fun harom() = 3;
- harom;
> val it = fn : unit -> int
- harom();
> val it = 3 : int
- val harom = 3;
- harom;
> val it = 3 : int
```

Mi most az *mosml*-értelmező válasza a `harom()` kifejezésre?

```
- harom();
> ???
```

3.2. Rekord

A rekord olyan *felcímkézett* ennes, amelyben – a címkék használata miatt – az elemek sorrendje közömbös. A rekord elemeit *kapcsos zárójelek* között kell felsorolni. Ugyanaz az eredménye például az alábbi két deklarációnak:

```
- val empl = {name = "Jones", age = 25, salary = 15300};
> val empl =
    {age = 25, name = "Jones", salary = 15300}
    : {age : int, name : string, salary : int}
- val empl = {name = "Jones", salary = 15300, age = 25};
> val empl =
    {age = 25, name = "Jones", salary = 15300}
    : {age : int, name : string, salary : int}
```

Az SML-értelmezők válaszukban a rekord elemeit rendszerint a címkék *ábécé-sorrendjében* írják ki.

A deklaráció után az elemekre a címkéjükkel hivatkozhatunk a program szövegében (a címkék természetesen lokálisak az adott rekordot deklaráló programegységben), például:

```
- #name empl;
> val it = "Jones" : string
- #age empl;
> val it = 25 : int
```

Az ennes is rekord, olyan rekord, amelyben a címkék "láthatatlan" természetes számok, például

```
- val negyes = {1="a", 2="b", 3="c", 4="d"};
> val negyes = ("a", "b", "c", "d") : string * string * string * string
- val negyes = {3="c", 4="d", 2="b", 1="a"};
> val negyes = ("a", "b", "c", "d") : string * string * string * string
```

A két változat egyenértékű, amint az az SML-értelmező válaszából látszik. Ugyanez a megszokott rövid alakban, a címkék elhagyásával:

```
- val negyes = ("a", "b", "c", "d");
> val negyes = ("a", "b", "c", "d") : string * string * string * string
```

Akárhogyan is definiáltuk a `negyes`-t, az elemeire, ha szükséges, a címkékkel hivatkozhatunk:

```
- #1 negyes;
> val it = "a" : string
- #4 negyes;
> val it = "d" : string
```

Egy újabb példa:

```
- #3 (#"a", #"b", 3, false);
> val it = 3 : int
```

Címkék helyett, ahol csak lehet, inkább az elegáns mintaillesztést használjuk:

```
- val (a, b, c, d) = (#"a", #"b", 3, false);
> val c = 3 : int
```

3.2.1. Rekordminta

Rekordelemre *címke* = *név* szerkezetű mintát lehet illeszteni, ahol az = *név* rész el is hagyható. Például:

```
- val {name = ename, salary = esalary, age = eage} = empl;
> val eage = 25 : int
> val ename = "Jones" : string
> val esalary = 15300 : int
```

Az SML szintaxisa megengedi, hogy a számunkra érdektelen mezőket a rekordmintákból elhagyjuk, és az összes elhagyott minta helyett `...`-ot írjunk. A `...`-ot tartalmazó *rekordspecifikációt részlegesnek* (parciálisnak) nevezzük, például:

```
- val {name = ename, salary = esalary, ...} = empl;
> val ename = "Jones" : string
> val esalary = 15300 : int
```

Mintának magukat a mezőneveket is használhatjuk:

```
- val {name, age, salary} = empl;
> val name = "Jones" : string
> val age = 15300 : int
> val salary = 25 : int
```

Az érdektelen mezők most is elhagyhatók, ha részleges rekordspecifikációt alkalmazunk.

```
- val {name, ...} = empl;
> val name = "Jones" : string
```

Jegyezzük meg, hogy függvény *argumentumaként* csak *teljesen specifikált rekord* adható meg, mert csak teljesen specifikált rekordnak van egyértelműen meghatározott típusa.

4. fejezet

Kifejezések

4.1. Újra az infix operátorról

Az *infix* operátor olyan függvény, amelynek a nevét (jelét) a két argumentuma közé írjuk.

Az SML-ben a programozó is definiálhat *infix* operátort. Az *infix* operátor előnye, hogy használatát az általános iskolától kezdve megszoktuk, és ezért az *infix* operátort tartalmazó kifejezést könnyebben olvassuk. Az operátorok precedenciájáról már korábban szóltunk. A programozó az `infix` deklarációban meghatározhatja az adott operátor precedenciáját is (l. a következő példát).

A logikai operátorokról szóló 2.2.4.2 szakaszban definiáltuk a logikai konjunkciót és alternációt megvalósító függvényeket `&&&`, ill. `|||` néven. Sokkal kényelmesebb a használatuk *infix* helyzetben:

```
- fun &&& (b, j) = b andalso j;
- infix 2 &&&;
- fun ||| (b, j) = b orelse j;
- infix 1 |||;
```

E definíció- és deklarációsorozattal az a baj, hogy az SML-értelmező hibát jelez, ha a definíciókat újból beolvassa, mert az *infix* helyzetűnek deklarált `&&&` és `|||` nevek a függvénydefinícióban *prefix* helyzetben fordulnak elő. Két megoldás is van:

1. Az op kulcsszó alkalmazásával az esetleg *infix* helyzetűnek deklarált nevet *átmenetileg prefix* helyzetűvé tesszük:

   ```
   - fun op &&& (b, j) = b andalso j;
   - infix 2 &&&;
   - fun op ||| (b, j) = b orelse j;
   - infix 1 |||;
   ```

2. A deklarációk és a definíciók sorrendjét megváltoztatjuk, és már a definícióban infix helyzetben használjuk a `&&&` és `|||` nevet:

   ```
   - infix 2 &&&;
   - fun b &&& j = b andalso j;
   - infix 1 |||;
   - fun b ||| j = b orelse j;
   ```

Következő példánk a *kizáró-vagy* művelet definíciója:

```
- infix 1 xor;
- fun p xor q = (p andalso not q) orelse (q andalso not p);
```

vagy ha így jobban tetszik:

```
- fun p xor q = (p orelse q) andalso not (p andalso q);
> val xor = fn : (bool * bool) -> bool
```

Egy függvény infix állapota csak a szintaxisra van hatással, a szemantikára nincs. Az infix állapot a `nonfix` kulcsszóval tartósan megszüntethető:

```
- nonfix xor;
```

A `nonfix` deklarációt a fejlesztők saját maguk számára találták ki. Használatát nem javasoljuk a programozói gyakorlatban, mert például a `nonfix +` deklaráció után a `+` jel szintaxisa szokatlan lenne, és használata váratlan hibajelzésekhez vezetne. Az `op` kulcsszóval, amint egy előző példában láttuk, *infix* helyzetű név *lokális érvénnyel* alakítható át *prefix* helyzetűvé, ezért biztonságos a használata.

4.1.1. Infix operátor kötése

Tudjuk, hogy a nagyobb precedenciájú operátor erősebben köt. A kifejezés kiértékelése szempontjából az sem közömbös, hogy az operátor balra avagy jobbra köt-e.

Az operátorok többsége *balra köt,* például az összeadás és a kivonás : $a+b+c = (a+b)+c$ és $a-b-c = (a-b)-c$. A *jobbra kötő* operátorok közül a legismertebb a hatványozás: $a^{b^c} = a^{(b^c)}$. A balra kötő operátorokat a már ismert `infix`, a jobbra kötőket az `infixr` kulcsszóval deklaráljuk. Példák:

```
- infix 6 plus;
- fun a plus b = "(" ^ a ^ "+" ^ b ^ ")";
- infix 7 times;
- fun a times b = "(" ^ a ^ "*" ^ b ^ ")";
- infixr 8 pwr;
- fun a pwr b = "(" ^ a ^ "**" ^ b ^ ")";
- "1" plus "2" plus "3";
> val it = "((1+2)+3)" : string
- "m" times "n" times "3" plus "i" plus "j" times "k";
> val it = "((((m*n)*3)+i)+(j*k))" : string
- "m" times "i" pwr "j" pwr "2" times "n";
> val it = "((m*(i**(j**2)))*n)" : string
```

Egy *infix* operátor az `op` kulcsszóval nemcsak függvénydeklarációban alakítható át átmenetileg *prefix* helyzetűvé, amint korábban láttuk, hanem tetszőleges kifejezésben. Például:

```
- op plus ("a", "b");
> val it = "(a+b)" : string
- op + (1, 2);
> val it = 3 : int
```

Jegyezzük meg, hogy a szóköz jellegű formázó karaktereknek (l. a 2.2.2.1 szakaszban a 3 megjegyzést) itt is csak elválasztó szerepük van, ti. a lexikai egységek felismerését teszik lehetővé az SML-fordító számára. Ezért az alábbi kifejezések jelentése azonos:

```
op plus ("a", "b");
op plus("a", "b");
op plus("a","b");
```

```
op          plus          ("a",          "b");
```

a következő kifejezés azonban *hibás,* mert `opplus`-t új (alfanumerikus) névnek tekinti az SML-értelmező, és mivel definiálatlan, hibát jelez:

```
- opplus ("a", "b");
! Toplevel input:
! opplus("a","b");
! ^^^^^^
! Unbound value identifier: opplus
```

Az alábbi kifejezések jelentése ugyancsak azonos, hiszen a nevek vagy alfanumerikusak lehetnek, vagy írásjelekből állhatnak, és a ´(´ nem használható írásjelekből álló nevek képzésére (l. 2.1.3.1):

```
- op + (1, 2);
- op+ (1, 2);
- op+(1, 2);
- op+(1,2);
- op        +          (1,          2);
```

4.2. Kifejezések kiértékelése az SML-ben

A kifejezések kiértékelési sorrendje, mint már említettük, alapvetően kétféle lehet: *mohó* és *lusta.*

Sztatikus kötésről[1] beszélünk, ha egy függvény (eljárás) fordításakor az értelmező a formális paraméter minden előfordulását az argumentum (az aktuális paraméter) értékével helyettesíti a függvény (eljárás) törzsében. (Ne feledjük, hogy az argumentum és a formális paraméter ennes, azaz összetett is lehet!) Kérdés, hogy az aktuális paraméterként átadott kifejezést az értelmező mikor értékeli ki: a behelyettesítés *előtt* vagy *után.*

Mohó (azaz érték szerinti, applikatív sorrendű, angolul eager, strict, call-by-value, applicative-order) kiértékelésről beszélünk akkor, ha egy függvény (eljárás) *összes argumentumát* kiértékeljük a behelyettesítés, azaz a függvény (eljárás) tényleges meghívása *előtt.*

Tisztán funkcionális nyelvekben sokszor alkalmazzák a *lusta* (szükség szerinti, normál sorrendű, angolul lazy, call-by-need, normal-order) kiértékelést: egy függvény (eljárás) argumentumát *csak akkor* értékeljük ki, *ha és amikor szükség van rá* a behelyettesítés után.[2] Nézzünk két egyszerű függvényt!

```
(* sq x = x négyzete
   sq : int -> int
*)
fun sq x = x * x;
```

Ha az `sq` függvényt meghívjuk, *lusta kiértékelés* esetén *kétszer* is kiszámítjuk az argumentumát.

```
(* zero x = x-től függetlenül mindig 0
   zero : int -> int
*)
```

[1]Nem sztatikus, hanem *dinamikus* az olyan kötés, amely a formális paraméter minden előfordulását a függvény (eljárás) meghívásakor (végrehajtásakor) helyettesíti az argumentummal (az aktuális paraméterrel).

[2]Ma már ritkán találkozni az ALGOL-60 *név szerinti* (call-by-name) paraméterátadásával, ahol a kiértékelésre szintén a behelyettesítés *után* kerül sor. A név szerinti paraméterátadás az argumentumként átadott kifejezést betű szerint adja át a meghívott eljárásnak. Később elmagyarázzuk a különbséget a név szerinti paraméterátadás és a lusta kiértékelés között.

A teljesség kedvéért említjük a *hivatkozás szerinti* (call-by-reference) paraméterátadást, amelyet számos programozási nyelv alkalmaz, többek között a Pascal és a C. Ez az átadási mód kétségtelenül hatékony, hiszen főtárbeli címet vesz át a hívott eljárás, ahonnan közvetlenül olvashat, ill. ahova közvetlenül írhat. De éppen ez a hátulütője is a módszernek, hiszen pl. egy eljárás sikertelensége esetén nehéz visszaállítani a korábbi állapotot.

```
fun zero x = 0;
```

Ha a `zero` függvényt meghívjuk, *mohó kiértékelés* esetén az argumentumát *feleslegesen* számítjuk ki, hiszen nem használjuk semmire.

4.2.1. Mohó kiértékelés

Az SML-ben egy kifejezés állandókból, változókból, függvényhívásokból és feltételes kifejezésekből állhat.

$f(E)$ értékének kiszámításához először az E kifejezés értékét határozzuk meg, majd f törzsében ezzel az argumentummal helyettesítjük a formális paraméter minden előfordulását. A *mintaillesztés* nem okoz gondot: az E kifejezés értékét most is ki kell számítani, majd az argumentumminta szerint fel kell bontani, és az egyes összetevőit kell behelyettesíteni a függvény törzsében a megfelelő helyre. Legyen pl. `fun f (x, y, z) =` *`törzs`*, ekkor az E-t fel kell bontani az `(x, y, z)` összetevőkre, majd a törzsben `x`-et, `y`-t és `z`-t kell helyettesíteni a megfelelő értékkel .

Példaképpen nézzük az `sq(sq(sq(2)))` kifejezés kiértékelését, más szóval egyszerűsítését, redukcióját! (Az egyszerűsítés eredményének nyilvánvalóan olyan kifejezésnek kell lennie, amelyik tovább már nem egyszerűsíthető, ún. *kanonikus* kifejezés.) A három függvényhívásból csak a legbelső hívásnak érték az argumentuma, ezért:

```
sq(sq(sq(2)))→sq(sq(2*2))→sq(sq(4))→sq(4*4)
   →sq(16)→16*16→256
```

A `zero(sq(sq(sq(2))))` kifejezés egyszerűsítési lépései hasonlók, pedig az eredmény nyilvánvalóan 0! Az adott esetben mohó kiértékelés mellett a számítógép feleslegesen dolgozik.

Bár nem mindig ilyen könnyű felismerni, sokszor nem kellene kiértékelni egy függvény argumentumának összes elemét, mert az eredmény nem függ az argumentum összes elemétől.

4.2.1.1. Mohó kiértékelés rekurzív függvények esetén

A *faktoriális* matematikai definíciója:

fac $0 = 1$

fac $n = n*$fac$(n-1)$

A faktoriális-függvény megvalósítása SML-ben:

```
(* fac n = n!
   fac : int -> int
*)
fun fac n = if n = 0 then 1 else n * fac(n-1)
```

Mohó kiértékelésének menete $n = 4$ mellett:

```
fac(4)→4*fac(4-1)→4*fac(3)→4*(3*fac(3-1))
   →4*(3*fac(2))→···→4*(3*(2*1))→···→24
```

A rekurzív kiértékelés szigorúan követi a matematikai definíciót.

4.2.1.2. Iteratív függvények

A fenti kiértékelésben az a rossz, hogy a rekurzív végrehajtás során minden részeredményt tárolni kell. Ha a szorzás asszociativitását kihasználnánk, nem kellene tárolni az összes tényezőt, csak az aktuális részeredményt. A számítógép a szorzás e tulajdonságát (és bármely más tulajdonságot) persze csak akkor alkalmaz, ha utasítjuk rá. Írjunk ilyen függvényt!

Először a

```
(* faci(n, p)= p*n!
   faci : int * int -> int
```

```
*)
fun faci (n, p) = if n = 0 then p else faci(n-1, n*p)
```

segédfüggvényt definiáljuk, majd felhasználjuk az eredetivel azonos hívási felületű függvény megvalósítására:

```
(* fac n = n!
   fac : int -> int
*)
fun fac n = faci(n, 1)
```

Nézzük a kiértékelését (egyes triviális lépéseket összevonunk):

```
faci(4,1)→faci(4-1,4*1)→faci(3,4)→faci(3-1,3*4)
   →faci(2,12)→···→faci(0,24)→24
```

Kiértékelés közben a `p` segédváltozóban (az ún. *gyűjtőargumentumban, akkumulátorban)* gyűjtjük a részeredményt, ezért a tárigény állandó marad. A kiértékelés tehát *iteratív* jellegű. Az ilyen rekurziót *terminális rekurziónak* vagy *jobbrekurziónak* (angolul: *tail* vagy *terminal recursion)* nevezzük. A jó fordítóprogramok felismerik az iteratívvá alakítható rekurziót, és még hatékonyabb tárgykódot állítanak elő. A `faci(n-1, n*p)` rekurzív hívás eredménye – további számítások nélkül – közvetlenül `faci(n, p)` eredményét adja. Az ilyen, ún. *terminális hívást* végre lehet hajtani úgy, hogy az értelmező- vagy fordítóprogram az `n` és a `p` lokális változókba beírja az `n` és a `p` új értékét, majd visszaugrik a kód elejére ahelyett, hogy ténylegesen, újból meghívná a függvényt. Az előző változatban a `fac(n-1)` hívás *nem* terminális hívás, mert az eredményét még meg kell szorozni `n`-nel.

`faci`-t rekurzív függvényként definiáltuk, ezért viszonylag könnyű belátni a helyességét, ugyanakkor a kiértékelése a segédváltozó bevezetésével iteratívvá vált. Nagyon sok rekurzív függvény (de nem mind!) iteratívvá alakítható segédváltozó bevezetésével, és így tárterületet takaríthatunk meg. Sokszor a végrehajtási idő is csökken, de sajnos néha nőhet is. Ha nem nyilvánvaló a nyereség, a lehető legtermészetesebb módon kell felírni az algoritmust. Esetünkben `faci` valamivel hatékonyabb `fac`-nál, ugyanakkor a működése nehezebben érthető meg.

4.2.1.3. Feltételes kifejezések speciális kiértékelése

A feltételes operátor (`if-then-else`) nem függvényhívás: *a részkifejezések kiértékelésére* csak akkor kerül sor, ha és amikor szükség van rájuk:

```
if E then E1 else E2
```

`E1`-re akkor van szükség, ha `E` igaz, `E2`-re akkor, ha `E` hamis.

Az `andalso` és az `orelse` logikai operátorok sem függvények, csupán kényelmesen használható *rövidítések,* mégpedig

```
E1 andalso E2 = if E1 then E2 else false
E1 orelse E2 = if E1 then true else E2
```

Nézzünk most egy olyan példát, amelyben a végtelen rekurziót kerüljük el a használatukkal:

```
(* even n = igaz, ha n páros
   even : int -> bool
*)
fun even n = (n mod 2 = 0);
(* isPwrOf2 : int -> bool
*)
fun isPwrOf2 n = n = 1 orelse even n andalso isPwrOf2(n div 2);
```

`isPwrOf2` megvizsgálja, hogy egy szám 2 egész hatványa-e. Kiértékelése azonnal véget ér, mihelyst eldönthető az eredménye.

Ha `andalso` és `orelse` fenti alkalmazásakor minden alkalommal mind a két operandusukat ki kellene értékelni, a rekurzió sohasem fejeződne be. `andalso` és `orelse` éppen azért lusta kiértékelésű, hogy az ilyen és hasonló eseteket elegánsan lehessen kezelni.

4.2.1.4. Gyakorló feladat

Írja át `isLetter` és segédfüggvényei korábbi definícióját (l. 2.2.4.3) `if`-ekkel, `andalso` és `orelse` nélkül!

4.2.2. Lusta kiértékelés

Láttuk, hogy műveletvégzés, függvényhívás előtt sokszor fölösleges vagy éppen káros előre kiszámítani az operandusokat, mert az végtelen rekurzióhoz, illegális művelethez (indexhatáron túli indexeléshez, 0-val való osztáshoz stb.) vezethet. Az SML-ben, mint láttuk, a programozó nem írhat olyan függvényt, amely lusta kiértékelésű lenne. Vannak azonban olyan nyelvek, amelyekben az argumentumot nem értékként, hanem kifejezésként adjuk át.

A procedurális programozást megteremtő Algol60 a korábban már említett *név szerinti (call-by-name)* paraméterátadást alkalmazza: a függvény törzsében a formális paraméter összes előfordulását az aktuális paraméterként átadott teljes kifejezéssel helyettesíti. (Ne tévesszük össze a számos procedurális nyelvben, így például a Pascalban és a C-ben alkalmazott *hivatkozás szerinti (call-by-reference)* paraméterátadással!) Például a

```
zero(sq(sq(sq(2))))
```

hívás *név szerinti paraméterátadás esetén* azonnal, az argumentum kiértékelése nélkül 0-t ad eredményül!

Sajnos, a név szerinti paraméterátadás viselkedése sem mindig kedvező: pl. az `sq(sq(sq(2)))` hívás esetén `sq` minden egyes meghívása megkétszerezi az argumentumok számát:

```
sq(sq(sq 2))→sq(sq 2) * sq(sq 2)→(sq 2 * sq 2) * sq(sq 2)
   →((2*2) * sq 2) * sq(sq 2)→···→(4*(2*2) * sq(sq 2))→···
```

Aligha ezt akarjuk. A *lusta kiértékelés* (azaz a *szükség szerinti hívás)* garantálja, hogy minden argumentumot csak egyszer kelljen kiértékelni: akkor, amikor *először* van rá szükség. Nem a kifejezést helyettesítjük tehát a törzsbe, hanem egy, a *kifejezésre utaló hivatkozást* (egy olyan mutatót, amely *el van rejtve,* amelyhez a programozó nem férhet hozzá, és ezért biztonságos). Amikor a futtatórendszer az argumentumot kiszámítja, a kapott értéket elrakja, és később az összes olyan helyen, ahol szükség lesz rá, felhasználja.

A számítógépben a függvényeket és argumentumaikat irányított gráffal szokás ábrázolni: a gráf egy részének kiértékelésekor a gráfot az eredményül kapott értékkel frissítik. A lusta kiértékelés működési elvének megértéséhez irányított gráf helyett most jelöljük $x = [E]$-vel azt, hogy x *összes előfordulása* osztozik az E értéken. Nézzük pl. `sq(sq(sq 2))` lusta kiértékelését!

```
sq(sq(sq 2))→x * x [x = sq(sq 2)]→x * x [x = y * y] [y = sq 2]
   →x * x [x = y * y] [y = 2 * 2]→x * x [x = y * y] [y = 4]
   →x * x [x = 4 * 4]→x * x [x = 16]→16 * 16→256
```

4.2.3. A mohó és a lusta kiértékelés összevetése

Sajnos, mint látjuk, a lusta kiértékeléshez (gyakran bonyolult) nyilvántartást kell vezetni. De más oka is van annak, hogy az SML-ben nincs lusta kiértékelés.

1. `zero(E) = 0` értelmes lenne akkor is, amikor `E`-t nem lehet kiértékelni. Ez ellentmond a hagyományos matematikai szemléletnek, amely szerint egy kifejezés csak akkor értékelhető ki, ha minden részkifejezése kiértékelhető.

2. A végtelen adatszerkezetek bonyolulttá teszik a program helyességének igazolását. A lusta kiértékelésű program eredménye nem valamilyen meghatározott érték, hanem egy, *csak részben kiértékelt kifejezés.* De ha állandóan a kiértékelési mechanizmusra kell gondolnunk programozás közben, akkor nem sokkal jutottunk előbbre, mint a változók pillanatnyi állapotát (azaz a program állapotterét) figyelembe vevő imperatív programozás esetén.

3. A hatékonysággal is vannak bajok: néha nyerünk, máskor vesztünk a lusta kiértékeléssel. Mint láttuk, `faci` *mohó kiértékelés mellett* hatékonyabb `fac`-nál, mert az `n*p` szorzást azonnal végrehajtja. *Lusta kiértékelés mellett* `faci(n, p)` ugyan `n`-et azonnal kiszámítja, hiszen szüksége van rá az `n=0` vizsgálat elvégzéséhez, a `p` kiértékelését azonban késlelteti és a szorzásokat akkumulálja:

   ```
   faci(4,1)→faci(4-1,4*1)→faci(3-1,3*(4*1))
      →faci(2-1,2*(3*(4*1)))···→24
   ```

A lusta kiértékelés fontos kutatási téma, a szerepe még nem jelentős – de egyre növekszik! – a gyakorlati programozásban.

5. fejezet

Lokális kifejezés, lokális és egyidejű deklaráció

5.1. Lokális kifejezés

A lokális kifejezést a `let` kulcsszó vezeti be. Szintaxisa a következő:

`let` D `in` E `end`

D sokszor nem egyetlen deklaráció, hanem D_1 ; D_2 ; $\ldots$; D_n alakú deklarációsorozat, más néven *szekvenciális deklaráció*, ahol a ; (pontosvessző) opcionális.

Lokális kifejezésben *érték, függvény, típus* és *kivétel* deklarálható. Az így deklarált lokális érték csak magában a lokális kifejezésben látható.

Most arra mutatunk példát, hogy mikor **ne** használjunk lokális kifejezést:

```
let val a = sin x
    val b = cos x
in
   if a < b then a else b
end
```

Ez a programrészlet azért nem szerencsés, mert az `if-then-else` feltételes kifejezésben már nem látszik, hogy az `a` valójában a `sin x`-et, a `b` pedig a `cos x`-et jelöli. Inkább az alábbi megoldást javasoljuk:

```
(* min(a, b) = a és b közül a kisebb
   min : real * real -> real
*)
fun min (a, b) : real = if a < b then a else b;
min(sin x, cos x)
```

Ebben a változatban a `min` név világosan utal arra, hogy a `sin x` és a `cos x` közül a minimálisat, vagyis a kisebbiket kell eredményül adni.

Törekedjünk arra, hogy értelmes jelentésű (és értelmes nevű) függvényeket, definiáljunk, és használjuk ki a programozási nyelv absztrakciós lehetőségeit!

5.2. Lokális deklaráció

A lokális deklarációt a `local` kulcsszó vezeti be:

`local` D_1 `in` D_2 `end`

A lokális kifejezéssel szemben a lokális deklarációban az `in` és az `end` kulcsszavak között is *deklaráció* áll, nem kifejezés. A lokális deklaráció célja az, hogy egy deklarációt egy másik deklaráción belül lokálissá tegyen, elrejtsen a külvilág elől. D_1 és D_2 deklarációsorozat (szekvenciális deklaráció) is lehet.

5.3. Egyidejű deklaráció

Az *egyidejű* (más néven *szimultán)* deklaráció elsősorban kölcsönösen rekurzív függvények definiálására használható. Kölcsönösen rekurzív függvényeket szekvenciális deklarációval nem lehet deklarálni.

`val` $id_1 = E_1$ `and` ... `and` $id_n = E_n$

Az egyidejű deklaráció előbb kiszámítja az *összes* E_i-t (kiszámításuk sorrendje, mivel nem lehet mellékhatásuk, közömbös), majd a kiszámított értékeket balról jobbra haladva, rendre hozzárendeli a megfelelő id_i-hez. Példaként bemutatunk egy nem igazán hatékony megoldást az egész számok páros, ill. páratlan voltát tesztelő `even`, ill. `odd` függvény megvalósítására:

```
(* even n = igaz, ha n páros
   even : int -> bool
   odd n = igaz, ha n páratlan
   odd : int -> bool
*)
fun even 0 = true
  | even n = odd(n-1)
and odd 0 = false
  | odd n = even(n-1)
```

Az egyidejű deklaráció azonosítók értékének felcserélésére is használható, például:

```
- val alma = "PIROS";
- val korte = "BARNA";
- val alma = korte and korte = alma;
> val alma = "BARNA" : string
> val korte = "PIROS" : string
```

6. fejezet

Listák

A *lista* azonos típusú elemek végtelen (gyakorlatilag véges) sorozata. Példák:

```
[3, 5, 9, 13, 17, 21]
["alma", "meggy", "szilva"]
```

A listát *rekurzív adatszerkezetnek* is tekinthetjük. A rekurzív definíció szerint a lista

- vagy üres,
- vagy egy elemből és ezt az elemet követő listából áll.

Az üres listát – a listaműveletek *egységelemét* – `[]`-lel vagy `nil`-lel jelöljük. A legalább egy elemből álló lista első elemét a lista *fejének,* a többi elemből álló listát a lista *farkának* nevezzük.

A listában az elemek sorrendje fontos. Egyes elemek ismétlődhetnek. Az elemek típusa tetszőleges, de egy listának csak azonos típusú elemei lehetnek.[1]

```
> val it = [3, 5, 9, 13, 17, 21] : int list
> val it = ["alma","meggy","szilva"] : string list
```

6.1. Típuskifejezések és típusoperátorok

Ha egy lista elemeinek a típusa `´a`, akkor a lista típusa `´a list`. Az üres lista típusa is `´a list`, hacsak nem alkalmazunk típusmegkötést. Az `´a list` az `int list`-hez hasonlóan *típuskifejezés;* a `list`, akárcsak a `*` és a `->`, *típusoperátor.*

A típusoperátoroknak is van *precedenciája:* a `list` precedenciája a legnagyobb, nála kisebb a `*`, a legkisebb pedig a `->` precedenciája. A `list` *postfix,* a `*` és a `->` *infix* pozíciójú típusoperátor.

Nézzünk néhány típuskifejezést, figyeljük meg bennük a típusoperátorok precedenciáját! Az egyenlőségjel bal és jobb oldalán egymással ekvivalens típuskifejezések vannak.

```
(string * string) list
string * string list = string * (string list)
int list list = (int list) list
```

[1]Más funkcionális nyelvekben, pl. a LISP-ben a listának különböző típusú elemei is lehetnek.

6.2. Lista létrehozása

Két konstruktorművelet használható lista létrehozására: a `[]` (`nil`) *konstruktorállandó* és az *infix* pozíciójú `::` *konstruktoroperátor.*[2]

Egy lista vagy üres (`[]`) lehet, vagy `x::xs` alakú, ahol `x`-szel a lista *fejét*, `xs`-sel pedig a lista *farkát,* azaz az eredetinél eggyel rövidebb részlistáját jelöljük. Könnyen elérhető egy lista *első,* sok munkával az *utolsó* eleme.

A `[3, 5, 9]` jelölés[3] rövidítés, mégpedig a `3::(5::(9::nil)` jelölés rövidítése. Azért, hogy ne kelljen zárójelet használni, az *infix* `::` (négyespont) operátor *jobbra köt:* `3::5::9::nil`. Egy lista elemeiként tetszőleges kifejezéseket adhatók meg.

Első példánk a lista alkalmazására legyen egy olyan rekurzív függvény, amely az `m` és `n` közötti egészek listáját adja eredményül. Ha `m>n`, az eredmény legyen az üres lista.

```
(* upto(m ,n) = az [m,n] tartományba eső egészek listája
   upto : int * int -> int list
*)
fun upto (m, n) =
      if m > n then [] else m :: upto(m+1, n)
```

6.3. Egyszerű műveletek listákkal

6.3.1. Lista elemeinek szorzata

A rekurzív megoldást általában az előforduló esetek elemzésével, a jellemző esetek szétválasztásával találjuk meg: listák esetén általában az *üres* és a *nem üres* lista esetét kell megkülönböztetnünk. Az üres lista nem létező elemeinek *szorzatát* célszerű 1-nek választani (miért is?): az 1 a szorzás egységeleme.

A `[]` *minta* csak az üres listára illeszkedik. Az `n::ns` minta csak olyan listára illeszkedik, amelynek legalább egy eleme van; a mintát zárójelbe kell rakni, mert *a függvényalkalmazás precedenciája nagyobb a négyesponténál.*

```
(* prod xs = az xs egészlista elemeinek szorzata
   prod : int list -> int
*)
fun prod [] = 1
  | prod (n::ns) = n * prod ns
```

Az üres, ill. a nem üres listát kezelő *változatok* (a Prolog szóhasználatával: *klózok)* a jelen esetben *kölcsönösen kizárják* egymást (a minták diszjunktak), ezért a két klóz sorrendje *az eredmény szempontjából* közömbös. De esetleg nem közömbös a sorrendjük *hatékonysági szempontból:* mivel a vizsgált lista mindaddig nem üres, amíg a rekurzív feldolgozás során el nem fogynak az elemei, az üreslista-vizsgálat az utolsó eset kivételével meghiúsul. Egyes SML-fordítók az összetettebb eseteket nem kezelik elég hatékonyan, ezért a hatékonyság javítása érdekében a klózokat célszerű lehet fordított sorrendben felírni:

```
fun prod (n::ns) = n * prod ns
  | prod [] = 1
```

Ha a függvény meghívásakor már az első minta illeszkedik az argumentumra, a második klóz kiértékelésére nem kerül sor. (Amint tudjuk, a kifejezések kiértékelése balról jobbra és felülről lefelé halad.) Hasonló esetekben mindig törekedni fogunk arra, hogy hatékony megoldást alkalmazzunk.

[2] `[]` és `nil` jelentése azonos. A `::` konstruktoroperátor helyett sokszor a vele azonos hatású, de prefix pozíciójú `cons` *konstruktorfüggvényről* beszélünk, amely azonban nincs belső függvényként definiálva az SML-ben.

[3] Az SML-lista szintaxisa *csak hasonló* a Prolog-listáéhoz! A Prologban ui. `[5|[6]]` és `[5,6]` azonos listát jelölnek. Az SML-ben az `[5::[6]]`-tal jelölt lista `[[5,6]]`-tal azonos.

A listaelemek *összege* hasonlóan képezhető. Az összeadás egységeleme a 0.

6.3.2. Lista legnagyobb eleme

Kicsit más a feladat egy lista *legnagyobb* (legkisebb) elemének megkeresésekor:

- üres listának nincs legnagyobb eleme,
- egyelemű listában az egyetlen elem a legnagyobb,
- legalább kételemű lista esetén a legnagyobb elemet úgy kapjuk meg, hogy vesszük a két elem közül a nagyobbat, továbbá a maradéklista elemei közül a legnagyobbat, és e kettő közül kiválasztjuk a nagyobbat.

```
(* maxl ns = az ns egészlista legnagyobb eleme
   maxl : int list -> int
*)
fun maxl (m::n::ns) =
      if m > n then maxl(m::ns) else maxl(n::ns)
  | maxl [m] = m
```

Az SML-értelmező erre a függvénydefinícióra figyelmeztető üzenettel válaszol:

```
! Warning: pattern matching is not exhaustive
```

Megjegyzések:

1. `maxl` üres listára nem alkalmazható; erre figyelmeztet a fenti üzenet. Később megmutatjuk, hogyan kell kezelni az ilyen, ún. *kivételeket.*
2. az `[m]` minta csak *egyetlen elemből* álló listára illeszkedik.
3. az `(m::n::ns)` minta csak olyan listára illeszkedik, amelynek legalább két eleme van.
4. Az algoritmus szempontjából mindegy lenne, hogy a lista elemei milyen típusúak, de a `>` reláció, mint tudjuk, többszörösen terhelhető módon polimorf (l. 7.1 szakasz). Mivel a programozó nem hozhat létre többszörösen terhelhető neveket az SML-ben, a függvény definiálásakor el kell dönteni, hogy a `>` relációnak melyik változatát kell beépíteni `maxl`-be (alapértelmezés szerint `int` a többszörösen terhelhető műveletek argumentumainak típusa). Később megmutatjuk, hogyan kell ún. *generikus* algoritmusokat írni.

6.3.3. Karakter, füzér és lista

Az SML-ben a füzér egydimenziós karaktertömb (karaktersorozat), nem lista. A füzért a rekurzív feldolgozás során esetleg többször át kell alakítani listává, majd vissza füzérré. Két belső függvény van erre a célra az SML-ben:

```
- explode "mosml";
> val it = [#"m", #"o", #"s", #"m", #"l"] : char list
```

A kapott lista minden eleme egyetlen karakter.

```
- implode it;
> val it = "mosml" : string
```

Füzérekből álló lista elemeit egyesíteni a `concat`-tal lehet:

```
- concat ["mo","sml"];
> val it = "mosml" : string
```

A következő szakaszokban néhány fontos listakezelő függvényt definiálunk.

6.4. Listák vizsgálata és darabokra szedése

Három függvényt mutatunk be ebben a csoportban: a `null` egy lista üres voltát vizsgálja, a `hd` egy nem üres lista első elemét, a `tl` egy nem üres lista első elemét követő részlistáját (a lista farkát) adja eredményül.[4]

```
(* null xs = igaz, ha az xs lista üres
   null : ´a list -> bool
*)
fun null (_::_) = false
  | null [] = true
```

Az aláhúzást (_) *mindenesjelnek* nevezzük. A mindenesjel-minta *mindenre* illeszkedik. Olyankor használhatjuk, amikor az illeszkedő értékre nem kell hivatkoznunk a függvény törzsében.

Az eredmény szempontjából a minták felírásának sorrendje közömbös ebben a függvényben is.

```
(* hd xs = a nem üres xs lista feje
   hd : ´a list -> ´a
*)
fun hd (x::_) = x
```

Ez a `hd` csak nemüres listára alkalmazható, amire az SML-értelmező figyelmeztet. Később bemutatjuk az üres listát is kezelni képes változatát.

```
(* tl xs = a nem üres xs lista farka
   tl : ´a list -> ´a list
*)
fun tl (_::xs) = xs
```

Ez a `tl` is csak nemüres listára alkalmazható, amire az SML-értelmező figyelmeztet. Később bemutatjuk az üres listát is kezelni képes változatát.

`hd` és `tl` *szelektorfüggvény,* `null` pedig *tesztelőfüggvény.*

6.5. Listák és egész számok

Ebben a csoportban is három függvényt mutatunk be: `length` egy lista hosszát adja eredményül; `take` egy lista elejéről vett adott számú elemből, `drop` egy lista elejéről adott számú elem elhagyásával képez eredménylistát.[5] Ha

$xs = [x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}]$

akkor

`length xs=` n
`take(xs,i)=` $[x_0, x_1, \ldots, x_{i-1}]$
`drop(xs,i)=` $[x_i, x_{i+1}, \ldots, x_{n-1}]$

`length` naív változata a következő:

```
(* nlength xs = xs elemeinek száma
   nlength : ´a list -> int
*)
fun nlength (_::xs) = 1 + nlength xs
  | nlength [] = 0
```

Egy példa a függvény alkalmazására:

[4] `hd` a *head* (fej), `tl` a *tail* (farok) szóból származik. `null`, `hd` és `tl` fenti definíciója csak illusztráció, ugyanis mind a három *belső* függvényként van definiálva az SML-ben.

[5] `length` *belső* függvény, `take` és `drop` pedig a List könyvtárban van definiálva az SML-ben.

```
- nlength [[1,2,3],[4,5,6]];
> val it = 2 : int
```

`nlength` rossz hatékonyságú, mert nem iteratív: az 1-esek a veremben csak gyűlnek, gyűlnek, amíg a maradéklista ki nem ürül. A függvény javított, iteratív változata:

```
(* length xs = xs elemeinek száma
   length : ´a list -> int
*)
local
   fun addlen (n, _::xs) = addlen (n+1, xs)
     | addlen (n, []) = n
in
   fun length xs = addlen(0, xs)
end
```

A *nagyon gyakran használt* (pl. könyvtárazott) függvények *hatékonysága* és *robosztussága*[6] fontos, még ha kevésbé szépek, kevésbé olvashatók is. A speciális feladatokra írt, *ritkábban használt* függvények azonban legyenek *könnyen olvashatók,* és a *helyességüket* is *egyszerűen* lehessen belátni, bizonyítani.

`take` első változata:

```
(* take(xs, i) = az xs első i db eleméből álló lista, ha i>=0;
                 az üres lista, ha i<0
   take : ´a list * int -> ´a list
*)
fun take (x::xs, i) = if i > 0 then x::take(xs, i-1) else []
  | take ([], _) = []
```

A bemutatott változat az `if-then-else` alkalmazása miatt nem olyan elegáns, de robosztus: negatív `i`-re az üres listát adja eredményül. Majdnem ugyanez valamivel szebben:

```
(* take(xs, i) = az xs első i db eleméből álló lista, ha i>=0;
                 xs, ha i<0
   take : ´a list * int -> ´a list
*)
fun take (_, 0) = []
  | take ([], _) = []
  | take (x::xs, i) = x::take(xs, i-1)
```

`take` második változata negatív `i`-re *a teljes listát* visszaadja. (Miért?) *Figyelem:* ebben a definícióban a klózok sorrendje nem közömbös! (Miért nem?)[7]

Nézzünk egy példát `take` egyszerűsítésére (egyes triviális lépéseket összevonunk)!

```
take([9,8,7,6],3)→9::take([8,7,6],2)→9::8::take([7,6],1)
   →···→9::8::7::[]→9::8::[7]→9::[8,7]→[9,8,7]
```

[6]Egy eljárás, függvény, program stb. akkor robosztus, ha szélsőséges körülmények között is a specifikációjának megfelelően, megbízhatóan, kiszámíthatóan viselkedik. Szélsőséges körülménynek számít például, ha egy függvényt ritkán előforduló, extrém értékre alkalmazunk.

Az SML-ben egy függvény robosztussága azt jelenti, hogy a függvény az értelmezési tartományába eső minden lehetséges argumentumra specifikiálva van, és e specifikáció szerint viselkedik. Például a belső `hd` és `tl` függvény, ha üres listára alkalmazzuk, meghatározott *kivételt* jelez az SML-ben. A kivételkezelésről később lesz szó.

[7]Robosztusnak tekinthető-e `take` itt bemutatott két változata? Abban az értelemben igen, hogy `i` minden lehetséges értékére definiálva vannak: a rekurzió minden esetben biztosan befejeződik; ha `i` negatív, az első az üres listát, a második az eredeti listát adja eredményül. Ugyanakkor ennek a robosztus megoldásnak hátránya is van: valószínű, hogy `i<0`-ra szándékosan ritkán fogják alkalmazni `take`-et, ha pedig valamilyen hiba folyományaként lesz negatív az `i`, az eredeti hiba hatása majd csak jóval később, a program egy egészen más pontján jelentkezik, és ezért nehezebb lesz felderíteni. Ezért a könyvtári `List.take` (és `List.drop`) `i<0`-ra *kivételt* jelez.

Az egyszerűsítési folyamatot bemutató példában a négyespontot (`::`) nem lista létrehozására használjuk, mint a programokban, hanem olyan listakifejezéseket írunk fel vele, amelyeket az egyszerűsítés során az SML-értelmezőnek ki kell értékelnie. A lista elemeit az SML-értelmező előbb egyesével berakja a verembe, majd hátulról visszafelé haladva megint elővesz őket az eredménylista előállításához.

Következő kérdésünk az, hogy érdemes-e megírni `take` iteratív változatát? Próbáljuk meg: a részeredményeket gyűjtsük az egyik argumentumban.

```
(* rtake(i, xs, zs) =
   rtake : int * ´a list * ´a list -> ´a list
*)
fun rtake (_, [], taken) = taken
  | rtake (i, x::xs, taken) =
      if i>0 then rtake(i-1, xs, x::taken) else taken;
```

Van-e valami furcsa ebben a megoldásban? Igen, van: az `x::taken` művelet miatt a listaelemek sorrendje megfordul! Ha ez nem engedhető meg, nem nyerünk semmit, mert a visszafordítása legalább ugyanannyiba kerülne, mint a `take` végrehajtása.

```
(* drop(xs, i) = az xs első i db elemének elhagyásával
                 előálló lista, ha i>0; xs, ha i<=0
   drop : ´a list * int -> ´a list
*)
fun drop (_, [])= []
  | drop (i, x::xs) = if i>0 then drop (i-1, xs) else x::xs;
```

Ez a megoldás kézenfekvő és szerencsére iteratív is. Az `else` ágban lévő `x::xs` lista ugyanaz, mint a `drop` második argumentuma; később az összefésülő rendezésről szóló 13.3 szakaszban bemutatunk tömörebb jelölést – a *réteges mintát* –, amely ilyen esetekben alkalmazható.

6.6. Listák összefűzése és megfordítása

Ebben a szakaszban két függvényt, az `append`-et és a `rev`-et mutatjuk be. Az `append` *infix* változatát a `@` jellel jelöljük. A két listát egybefűző `append` így specifikálható:

$$[x_1, \ldots, x_m] \mathbin{@} [y_1, \ldots, y_n] = [x_1, \ldots x_m, y_1, \ldots y_n]$$

Az `xs`-t először az elemeire bontjuk, majd hátulról visszafelé haladva fűzzük az elemeket az `ys`-hez, ugyanis a listákat csak elölről tudjuk felépíteni.

```
(* append(xs, ys) = xs összes eleme ys elé fűzve
   append : ´a list * ´a list -> ´a list
*)
fun append ([], ys) = ys
  | append (x::xs, ys) = x::append(xs, ys);
```

Infix változatát pl. így definiálhatjuk:

```
- infix 5 @;
- val @ = append;
```

Itt is, akárcsak `take`-nél, a lista építésének költsége meghaladja a veremhasználat – ti. az adatok veremben való tárolásának – költségét, ezért nem érdemes iteratív változatot kidolgozni.

A Pascal, a C explicit mutatókkal kezeli a listákat, ezért az egyik lista végén a mutató átirányítható a másik listára. Az ilyen, ún. destruktív frissítés gyorsabb, mint a másoló frissítés. Csakhogy ez veszélyes lehet! Pl. mi van akkor, ha mindkét argumentum ugyanarra a listára mutat?

Most nézzük a listát megfordító `rev` egy naív megoldását:

```
(* nrev xs = xs megfordítva
   nrev : ´a list -> ´a list
*)
fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x];
```

és egy példát `nrev` redukciójára:

```
nrev([1,2,3,4])→nrev([2,3,4])@[1]→nrev([3,4])@[2]@[1]
   →nrev([4])@[3]@[2]@[1]→nrev([])@[4]@[3]@[2]@[1]
   →[]@[4]@[3]@[2]@[1]→[4]@[3]@[2]@[1]→[4,3]@[2]@[1]→...
```

Ez eddig n lépés volt. A továbbiakban az aktuális bal szélső listát megint elemeire kell bontani, majd összerakni $0, 1, 2, \ldots, n-1$ lépésben.

`nrev` nagyon rossz hatékonyságú: $O(n^2)$. De emlékezzünk csak vissza `rtake`-re: ott megfordult a listaelemek sorrendje, bár nem akartuk. Most *pontosan ezt* akarjuk, használjunk tehát segédargumentumot a `revto` segédfüggvényben:

```
(* revto(xs, ys) = xs elemei fordított sorrendben ys elé fűzve
   revto : ´a list * ´a list -> ´a list
*)
fun revto ([], ys) = ys
  | revto (x::xs, ys) = revto(xs, x::ys);
```

`revto` lépésszáma arányos a lista hosszával. Segítségével `rev` definíciója (`revto` lokális is lehetne `rev`-ben):

```
(* rev xs = xs megfordítva
   rev : ´a list -> ´a list
*)
fun rev xs = revto (xs, []);
```

Egy 1000 elemű listát `rev` 1000 lépésben, `nrev` $\frac{1000 \cdot 1001}{2} = 50500$ lépésben fordít meg. Hatalmas a nyereség!

6.7. Listákból álló lista, párokból álló lista

Ebben a szakaszban megint három függvényt definiálunk. `flat` kétszeres mélységű listából egyszeres mélységűt készít; `combine` két azonos hosszúságú lista elemeiből egyetlen, párokból álló listát állít elő; `split` pedig `combine` inverz függvénye.

`flat` specifikációja és definíciója:

$$\mathrm{flat}([x_1, x_2, \ldots, x_m], [y_1, y_2, \ldots, y_n]) = [x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_m]$$

```
(* flat xss = a kétszeres mélységű xss lista részlistáinak
              elemeiből képzett lista
   flat : ´a list list -> ´a list
*)
fun flat [] = []
  | flat (ls::lss) = ls @ flat lss;
```

Az algoritmus elég gyors, ha `ls` jóval rövidebb `lss`-nél. `combine` specifikációja és definíciója:

$$\mathrm{combine}([x_1, x_2, \ldots, x_m], [y_1, y_2, \ldots, y_m]) = [(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)]$$

```
(* combine(xs,ys) = az xs és ys elemeiből képzett párok listája
   combine : ´a list * ´b list -> (´a * ´b) list
*)
fun combine ([], []) = []
  | combine (x::xs, y::ys) = (x,y)::combine(xs, ys);
```

Az SML-értelmező figyelmeztet: nem fedtünk le minden esetet (ui. az argumentumként átadott listák különböző hosszúságúak lehetnek). Az ilyen esetek kezelésére a később ismertendő *kivétel-kezelés* alkalmas.

`split`-et `combine` inverz függvényeként definiáljuk:

```
(* split xys = a párok listájából előállított listapár
   split : (´a * ´b) list -> ´a list * ´b list
*)
fun split [] = ([], [])
  | split ((x, y)::pairs) =
      let val (xs, ys) = split pairs
      in (x::xs, y::ys)
      end;
```

Iteratív változata segédargumentumokban gyűjti külön-külön a két listát, de sajnos megfordítva:

```
(* rsplit : (´a * ´b) list * ´a list * ´b list -> ´a list * ´b list
*)
fun rsplit ([], xs, ys) = (xs, ys)
  | rsplit ((x, y)::pairs, xs, ys) = rsplit(pairs, x::xs, y::ys);
```

7. fejezet

Polimorfizmus

A polimorfizmus több válfaját alkalmazzuk a programozásban.

- Egy *polimorf név* **egyetlen** olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus.*
- Egy *többszörösen terhelt név* **több** algoritmust azonosít: ahány típusú argumentumra alkalmazható, annyifélét; ez az ad-hoc vagy *többszörös terheléses polimorfizmus.*
- A polimorfizmus harmadik változatát *öröklődéses polimorfizmusnak* nevezzük. (Öröklődéses polimorfizmust használ az objektum-orientált programozás.)

7.1. Polimorf típusellenőrzés

A típus nélküli és a gyengén típusos nyelvek a programozónak nagyobb szabadságot adnak, de a tévedés lehetősége is nagyobb. Az erősen típusos nyelvek a programozót jobban korlátozzák, de biztonságosabbak. Az SML-ben *polimorf típusellenőrzés* van: a szigorú típusellenőrzés flexibilis, automatikus típuslevezetéssel (type derivation, type inference) társul.

Nézzük a következő definíciót!

```
fun id x = x
```

Milyen típusú itt az `x`? Mindegy! Az `id` függvény ún. *polimorf függvény,* az `x` pedig *politípusú* azonosító. A típussémák elméletében a politípus jele α, β, γ stb., az SML-ben `´a`, `´b`, `´c` stb. Az SML-értelmező válasza a fenti definícióra tehát a következő:

```
> val id = fn : ´a -> ´a
```

A *percjellel* kezdődő típusneveket (`´a`-t, `´b`-t, `´c`-t stb.) *típusváltozónak* nevezzük, és *alfának, bétának, gammának* stb. olvassuk.

Az egyenlőségvizsgálatot is megengedő ún. *egyenlőségi típusok* (equality types) típusváltozóinak jelölésére *két percjellel* kezdődő neveket használunk: `´´a`, `´´b`, `´´c` stb.

A polimorf típus: *típusséma.* Amikor a típusváltozót konkrét típussal helyettesítjük, e séma egy-egy példányát kapjuk.

Nézzünk két újabb, nagyon egyszerű példát polimorf függvények definiálására! Egy pár első, ill. második tagjának kiválasztására használhatók az alábbi *projekciós* függvények, ahol `´a` és `´b` nem feltétlenül különböző típusok:

```
(* fst : ´a * ´b -> ´a
*)
fun fst(x, _) = x
```

```
(* snd : ´a * ´b -> ´b
*)
fun snd(_, y) = y
```

7.2. Egyenlőségvizsgálat polimorf függvényekben

Képzeljük el azt a függvényt, amelyik megvizsgálja, hogy egy `ls` listában benne van-e egy bizonyos `e` elem. Polimorf-e ez a függvény? A lista minden eleméről el kell tudni dönteni, hogy egyenlő-e `e`-vel. Csakhogy az egyenlőségvizsgálatot nem minden függvényre és absztrakt típusra lehet elvégezni! Miért is nem?

- Egy f és egy g függvény akkor és csak akkor egyenlő, ha $\forall x \bullet f(x) = g(x)$. Ezt *általánosságban* lehetetlen eldönteni.
- Az absztrakt típusok közül az `abstype` deklarációval deklaráltakra csak az absztrakt típussal együtt definiált műveletek alkalmazhatók, és egyáltalán nem biztos, hogy az egyenlőség szerepel e műveletek között. A `datatype` deklarációval deklarált absztrakt típusokon az egyenlőségvizsgálat akkor végezhető el, ha az adattípus konstruktorain elvégezhető az egyenlőségvizsgálat.[1]

Az egyenlőség tehát csak *korlátozott értelemben* polimorf. *Egyenlőségi típusnak* (equality type) nevezzük az olyan típust, amelyen az egyenlőségvizsgálat elvégezhető. Amint már említettük, az ilyen típusváltozókat az SML *két percjelből* (prime-ból) és egy betűből álló azonosítóval (`´´a`, `´´b`, `´´c` stb.) jelöli. Pl.

```
- op =;
> val it = fn : ''a * ''a -> bool
```

Most már definiálhatjuk az `isMem` (eleme) függvényt, ill. operátort:

```
(* isMem(x, ys) = x eleme-e ys-nek
   isMem : ''a * ''a list -> bool
*)
fun isMem (x, y::ys) = x = y orelse isMem (x, ys)
  | isMem (_, []) = false;
infix isMem;
```

7.3. Polimorf halmazműveletek

A `newMem` függvény egy új elemet rak be egy listába, ha az elem még nincs benne:

```
(* newMem(x, xs) = [x] és xs listaként ábrázolt uniója
   newMem : ''a * ''a list -> ''a list
*)
fun newMem (x, xs) = if x isMem xs then xs else x::xs;
```

`newMem`, ha a sorrendtől eltekintünk, halmazt hoz létre. A `setof` függvény halmazt készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket:

```
(* setof xs = xs elemeinek listaként ábrázolt halmaza
   setof : ''a list -> ''a list
*)
```

[1] Az `abstype` és a `datatype` deklarációról később lesz szó a jegyzetben.

```
fun setof (x::xs) = newMem (x, setof xs)
  | setof [] = [];
```

A `setof` függvénynek elég rossz a hatékonysága. Szerencsésebb, ha a halmazokat a megszokott halmazműveletekkel kezeljük. Most továbbra is egyszerű listaként ábrázoljuk őket, de később valamilyen hatékonyabb tárolást választhatunk, pl. rendezett listát vagy bináris fát.

Öt halmazműveletet definiálunk: unió (`union`, $S \bigcup T$), metszet (`inter`, $S \bigcap T$), részhalmaza-e (`isSubset`, $T \subseteq S$), egyenlők-e (`isSetEq`, $S = T$), hatványhalmaz (`powerset`, pS).

```
(* union(xs, ys) = az xs és ys elemeiből álló halmazok uniója
   union : ''a list * ''a list -> ''a list
*)
fun union (x::xs, ys) = newMem(x, union(xs, ys))
  | union ([], ys) = ys;

(* inter(xs, ys) = az xs és ys elemeiből álló halmazok metszete
   inter : ''a list * ''a list -> ''a list
*)
fun inter (x::xs, ys) =
          let val zs = inter(xs, ys)
          in
             if x isMem ys then x::zs else zs
          end
  | inter ([], _) = [];

(* isSubset (xs, ys) = az xs elemeiből álló halmaz részhalmaza-e
                       az ys elemeiből álló halmaznak
   isSubset : ''a list * ''a list -> bool
*)
fun isSubset (x::xs, ys) = (x isMem ys) andalso isSubset(xs, ys)
  | isSubset ([], _) = true;
infix isSubset;
```

A listák egyenlőségvizsgálata belső művelet az SML-ben. Halmazokra mégsem használható, mert pl. a `[3, 4]` és a `[4, 3, 4]` listák ugyan különböznek, de mint halmazok egyenlőek. Halmazként egyenlő pl. `[3, 4]` és `[4, 3]` is.

```
(* isSetEq(xs, ys) = az xs és ys elemeiből álló halmazok egyenlők-e
   isSetEq : ''a list * ''a list -> bool
*)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs);
```

A hatványhalmaz egy halmaz *összes* részhalmazának a halmaza, az eredeti halmazt és az üres halmazt is beleértve. Jelöljük S-sel az eredeti halmazt. S hatványhalmazát úgy állíthatjuk elő, hogy S-ből kiveszünk egy x elemet, és azután *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát. Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \bigcup \{x\} \subseteq S$, így mind T, mind $T \bigcup \{x\}$ eleme S hatványhalmazának. A `pws` függvényben a `base` argumentum gyűjti a hatványhalmaz elemeit; kezdetben üresnek kell lennie.

```
(* pws(xs, base) = az xs halmaz hatványhalmazának és
                   a base halmaznak az uniója
   pws : ´a list * ´a list -> ´a list list
*)
fun pws (x::xs, base) = pws(xs, base) @ pws(xs, x::base)
  | pws ([], base) = [base];
```

A `pws(xs, base) @ pws(xs, x::base)` kifejezésben `pws(xs, base)` valósítja meg az $S - \{x\}$ rekurzív hívást (hiszen `x::xs` felel meg S-nek), azaz állítja elő az összes olyan halmazt, amelyekben `x` nincs benne, `pws(xs, x::base)` pedig ugyancsak rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van. Halmazegyenlettel `pws` eredménye így adható meg:

$$\mathtt{pws}(S, B) = \{T \bigcup B | T \subseteq S\}$$

```
(* powerset xs = az xs halmaz hatványhalmaza
   powerset : ´a list -> ´a list list
*)
fun powerset xs = pws(xs, []);
```

8. fejezet

Adattípusok

8.1. A `datatype` deklaráció

A következő példa `person` néven új *összetett típust* hoz létre. Az új típusnak négy *adatkonstruktora* (röviden: konstruktora) van: `King`, `Peer`, `Knight` és `Peasant`; közülük `King` ún. *adatkonstruktorállandó*, a másik három ún. *adatkonstruktorfüggvény.*

```
datatype person = King
                | Peer of string * string * int
                | Knight of string
                | Peasant of string
```

Az adatkonstruktoroknak is van típusuk. `King` (király) csak egy van, ezért definiálhattuk konstruktorállandóként. A `Peer`-t (főnemest) nemesi címe (`string`), birtokának neve (`string`) és sorszáma (`int`), a `Knight`-ot (lovagot) és a `Peasant`-ot (parasztot) csupán a neve (`string`) azonosítja.

```
King :    person
Peer :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

Az új típusba tartozó értékek *teljes jogú* értékek, így például lista is képezhető belőlük:

```
- val persons = [King,
                 Peasant "Jack Cade",
                 Knight "Gawain",
                 Peer ("Duke", "Norfolk", 9)];
> val persons = [...] : person list
```

Ha függvényt írunk az új típusú adatok kezelésére, az esetek mintaillesztéssel választhatók szét. Itt pl. a `title (Peasant name)` a *minta* (pattern), és benne a `name` a *mintaazonosító* (pattern identifier).

```
(* title p = p megszólítása
   title : person -> string
*)
fun title King = "His Majesty the King "
  | title (Peer (deg, ter, _)) = "The " ^ deg ^ " of " ^ ter
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

Minden esetet le kell fedni mintával, különben hibaüzenetet kapunk. A minták tetszőlegesen összetettek lehetnek (lehetnek bennük ennesek, listák, rekordok stb.). Például a `sirs` függvény az összes `Knight` nevét összegyűjti a `person` típusú személyek egy listájából:

```
(* sirs ps = az összes Knight nevének listája
   sirs : person list -> string list
*)
fun sirs [] = []
  | sirs ((Knight s)::ps) = s::sirs ps
  | sirs (_::ps) = sirs ps
```

Itt a változatok sorrendje *fontos,* mert ha más lenne, a `_::ps` minta nemcsak `King`-re, `Peer`-re és `Peasant`-ra illeszkedne (ti. ezek helyett áll itt!), hanem `Knight`-ra is.

Az összes diszjunkt eset fölsorolása segíti az algoritmus helyességének belátását, bizonyítását. Miért vontunk össze mégis három esetet egyetlen változatban? Azért, mert a három eset részletezése hosszabbá tenné a program szövegét is, végrehajtását is. A bizonyítás sem okoz gondot, ha a harmadik sort *feltételes egyenletnek* tekintjük:

$$\texttt{sirs(p::ps) = sirs ps if } \forall \texttt{s} \cdot \texttt{p} \neq \texttt{Knight s}$$

A sorrend még fontosabb az alábbi példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetnünk: azokat, amelyek *igaz* eredményt adnak:

```
(* superior (p, r)= igaz, ha p magasabb rangú r-nél
   superior : person * person -> bool
*)
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false
```

8.2. A felsorolásos típus

Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes bevezetni a *felsorolásos típust* (enumeration type). A `datatype` deklaráció használható felsorolásos típus létrehozására is, pl. így

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
```

A felsorolásos típusnak csak *konstruktorállandói* vannak. Az új típus alkalmazásához a `person` típust újra deklarálnunk kell:

```
datatype person = King
                | Pear of degree * string * int
                | Knight of string
                | Peasant of string
```

`degree` típusú adatok feldolgozásakor külön-külön elemezzük az előforduló eseteket, pl.:

```
(* lady p = p főnemes hitvesének rangja
   lady : degree -> string
*)
```

```
fun lady Duke     = "Duchess "
  | lady Marquis  = "Marchioness"
  | lady Earl     = "Countess"
  | lady Viscount = "Viscountess"
  | lady Baron    = "Baroness"
```

A belső `bool` típushoz hasonló `Bool` típust és hozzá a `Not` függvényt például így hozhatjuk létre:

```
datatype Bool = True | False;
(* Not b = b negáltja
   Not : Bool -> Bool
*)
fun Not True = False
  | Not False = True
```

8.3. Polimorf adattípusok

Láttuk, hogy a `list` nem típus, hanem *postfix* pozíciójú *típusoperátor* (szőrszálhasogatóbban: *típuskonstruktorfüggvény)*, `int list`, `int list list`, `(string * string) list` stb. azonban már típusok. A `datatype` deklarációval tehát az adatkonstruktorok mellett *típuskonstruktort* (pontosabban *típuskonstruktor állandót* vagy *típuskonstruktorfüggvényt)* is létrehozunk.

A belső `´a list` típushoz hasonló `´a List` listát és vele együtt a `Nil` és a `Cons` *adatkonstruktorokat* például így definiálhatnánk:

```
datatype ´a List = Nil | Cons of ´a * ´a List
```

A `Cons` *adatkonstruktorfüggvény* alkalmazásával elég körülményes a listák létrehozása. Az 1, 2, 3, 4 sorozatot például így kell megadni:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))))
```

Bevezethetjük az *infix* pozíciójú `:::` (hatospont) *adatkonstruktoroperátort,* hogy kényelmesebb jelölést használhassunk:[1]

```
infix 5 ::: ; val op ::: = Cons
```

A `:::` adatkonstruktoroperátort közvetlenül a típusdeklarációban is definiálhatjuk:

```
infix 5 ::: ; datatype ´a List = Nil | ::: of ´a * ´a List;
```

Következő példánk legyen két típus *megkülönböztetett egyesítése,* más néven diszjunkt uniója:

```
datatype (´a, ´b) disun = In1 of ´a | In2 of ´b
```

Itt három dolgot definiáltunk:

1. a kétargumentumú `disun` típusoperátort,
2. az `In1 : ´a -> (´a, ´b) disun` és
3. az `In2 : ´b -> (´a, ´b) disun` adatkonstruktorfüggvényeket.

`(´a, ´b) disun` az `´a` és `´b` típusok megkülönböztetett egyesítése. *Megkülönböztetettnek* nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy `(´a, ´b) disun` típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek `In1 x` alakúak, ha `x` `´a` típusú, és `In2 y` alakúak, ha `y` `´b` típusú. Az `In1` és `In2` konstruktorfüggvények olyan

[1] A `:::` és az `=` közé szóközt kell rakni, különben a fordítóprogram egyetlen (írásjelekből álló) névnek tekinti a jelsorozatot.

címkének tekinthetők, amelyek az ´a típust megkülönböztetik a ´b típustól. (Megkülönböztetett egyesítés például a Pascal variábilis rekordja is.)

A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (v.ö. az objektum-orientált programozással, ahol például egy *alakzat* osztálynak *téglalap, háromszög* vagy *kör* nevű leszármazottai lehetnek). Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni például *különböző típusú elemekből álló listát:*

```
[In2 King, In1 "Skócia"] : ((string, person) disun) list
[In1 "zsarnok", In2 1040] : ((string, int) disun) list
```

A lehetséges eseteket most is *mintaillesztéssel* elemezhetjük, pl.

```
(* concat d = a d diszjunkt unió In1 címkéjű elemeinek konkatenációja
   concat : (string, ´a) disun list -> string
*)
fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls

- concat [In1 "Ó! ", In2 (1040, 1057), In1 "Skócia"];
> val it = "Ó! Skócia : string
```

Nézzük, mi a típusa az `In1 "Ó! Skócia"` kifejezésnek!

Az `In1` konstruktorfüggvény `´a -> (´a, ´b) disun` típusú, ezért `string` típusú argumentumra alkalmazva `(string, ´b) disun` típusú értéket ad eredményül. Az `In2 King` kifejezés típusa nyilvánvalóan `(´a, person) disun` lesz.

Az `[In2 King, In1 "Skócia"]` kifejezésben mindkét alaptípust lekötjük, ezért ennek a kételemű listának a típusa a fent is látható `((string, person) disun) list`. Ugyanez lesz a háromelemű `[In1 "Ó", In2 King, In1 "Skócia"]` lista típusa is, hiszen az `´a` típusváltozót az `In1` konstruktorfüggvénnyel mindkét esetben `string`-nek adjuk meg.

Az `[In2 "Ó", In2 King, In1 "Skócia"]` kifejezés viszont hibajelzést eredményez, mert a `´b` típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötni.

8.4. A case-kifejezés

Szintaxisa a következő:

```
case E of P1 => E1 | P2 => E2 | ··· | Pn => En
```

Az SML-értelmező – balról jobbra és fölülről lefelé haladva – megpróbálja `E`-t `P1`-re illeszteni, ha nem sikerül, `P2`-re s.í.t. A `case`-kifejezés eredménye az `E` kifejezésre illeszkedő első `Pi` mintához tartozó `Ei` kifejezés lesz . Például a `lady` függvényt így is definiálhattuk volna:

```
(* lady p = p főnemes hitvesének rangja
   lady : degree -> string
*)
fun lady p =
  case p of Duke     => "Duchess "
          | Marquis  => "Marchioness"
          | Earl     => "Countess"
          | Viscount => "Viscountess"
          | Baron    => "Baroness"
```

Az a lehetőség tehát, hogy a `fun` függvénydefinícióban változatokat definiálhatunk, nem egyéb, mint rövidítés, *szintaktikai édesítőszer.*

9. fejezet

Részlegesen alkalmazható függvények

9.1. Az `fn` jelölés

A 2.1.2 szakaszban már találkoztunk a (gyakran *lambdának* ejtett) `fn` kulcsszóval. Névtelen függvény például az `fn x => E` *függvénykifejezés,* ahol `E`-nek (esetleg `x`-től függő), kiértékelhető kifejezésnek kell lennie. Ha `x ´a`, `E` pedig `´b` típusú érték, akkor a függvénykifejezés `´a -> ´b` típusú. Mivel ennek a függvénynek *nincs neve,* rekurzív függvény íly módon nem hozható létre. Mintaillesztéssel több változat is megadható:

```
fn p1 => E1 | p2 => E2 | ··· | pn => En
```

Nézzünk egy példát a függvénykifejezés alkalmazására!

```
- (fn n => n * 2) 9;
> val it = 18 : int
```

A függvénykifejezést – precedenciaokok miatt – általában zárójelbe kell rakni. A névtelen függvénynek nevet többek között így adhatunk:

```
(* double n = az n egész kétszerese
   double : int -> int
*)
- val double = fn n => n * 2;
- double 9;
> val it = 18 : int
```

Az `if-then-else`, `andalso` és `orelse` logikai operátorok *rövidítések,* szemantikailag ekvivalensek az alábbi függvényalkalmazásokkal:

```
if E then E1 else E2 = (fn true => E1 | false => E2) E
E1 andalso E2 = (fn false => false | true => E2) E1
E1 orelse E2 = (fn true => true | false => E2) E1
```

Figyeljük meg, hogy a λ-kalkulustól örökölt `fn`-jelölésnek milyen kifejező ereje van! `fn`-jelöléssel szinte az összes megszokott programozási jelölés pótolható, többségük nem más, mint *szintaktikai édesítőszer.*

A korábban látott `lady` függvényt `fn`-jelöléssel is definiálhattuk volna:

```
(* lady p = p főnemes hitvesének rangja
   lady : degree -> string
*)
val lady = fn p => case p of Duke     => "Duchess "
                           | Marquis  => "Marchioness"
```

```
                                  | Earl     => "Countess"
                                  | Viscount => "Viscountess"
                                  | Baron    => "Baroness"
```

9.1.1. Függvény definiálása `fun`, `val` és `val rec` kulcsszóval

A `fun`, ill. a `val` kulcsszóval kezdődő függvénydefiníciók között az a különbség, hogy `fun` esetén a név után argumentumnak *kell állnia,* `val` esetén pedig a név után *nem állhat* argumentum.

```
fun double n = n * 2;
val double = fn n => n * 2;
```

A `fun` kulcsszóval kezdődő függvénydefiníció lehet rekurzív is:

```
(* replist(n, x) = n db x értékből álló lista
   replist : int * ´a -> ´a list
*)
fun replist (n, x) =
      if n = 0 then [] else x::replist(n-1, x)
```

A `val` kulcsszóval kezdődő függvénydefiníció csak akkor lehet rekurzív, ha ezt a `val` után álló `rec` szócskával jelezzük:

```
(* replist(n, x) = n db x értékből álló lista
   replist : int * ´a -> ´a list
*)
val rec replist =
      fn (n, x) => if n = 0 then [] else x::replist(n-1, x)
```

9.2. Részlegesen alkalmazható függvények

Tudjuk, hogy az SML-ben egy függvénynek csak egyetlen argumentuma van, de ez egy pár, egy ennes, egy másik függvény stb. is lehet. Több argumentumú függvényt olyan függvénnyel is megvalósíthatunk, amely függvényt ad eredményül.

A *részlegesen alkalmazható* (partially applicable) függvényeket H. B. Curry amerikai matematikus után *curried* függvényeknek is nevezik, noha a jelölést egy másik amerikai matematikusnak, Schönfinkelnek köszönhetjük. Egy részlegesen alkalmazható függvény argumentumait egymástól egy vagy több szóköz jellegű karakterrel kell elválasztani.

Nézzük a következő függvénydefiníciókat:

```
(* prefix pre post = pre és post konkatenációja
*)
- fun prefix pre post =
        let fun cat post = pre ^ post
        in cat post
        end;
> val prefix = fn : string -> (string -> string)

- val prefix = fn pre => (fn post => pre ^ post);
> val prefix = fn : string -> (string -> string)
```

A két definíció ekvivalens, mindkettő a részlegesen alkalmazható `prefix` függvényt definiálja. A függvény típusát leíró *típuskifejezésből* kiolvasható, hogy ha a `prefix` függvényt `string` típusú argumentumra alkalmazzuk, *függvényt* ad eredményül, amely ugyancsak `string` típusú argumentumra alkalmazható és `string` típusú értéket ad eredményül.

Egy részlegesen alkalmazható függvény alkalmazható csak az első, az első és a második stb. argumentumára. A részleges alkalmazás eredménye maga is *függvény.*

A részlegesen alkalmazható `prefix` függvény *kétargumentumú függvényként* viselkedik. Nézzünk néhány példát a használatára:

```
- prefix "Sir ";
> val it = fn : string -> string
- it "Georg Solti"
> val it = "Sir Georg Solti" : string

- val knightify = prefix "Sir "
- val dukify = prefix "The Duke of "
- val lordify = prefix "Lord "
```

`prefix` fenti definíciói nehézkesek, az alábbi változat jóval olvashatóbb:

```
- fun prefix pre post = pre ^ post;
> val prefix = fn : string -> (string -> string)
```

Természetesen a részlegesen alkalmazható függvények is lehetnek rekurzívak, pl.

```
(* replist n x = n db x értékből álló lista
   replist : int -> ´a -> ´a list
*)
fun replist 0 x = []
  | replist n x = x::replist (n-1) x
```

`replist` olyan függvény, amelyet `int` típusú értékre alkalmazva olyan függvényt kapunk, amely `´a` típusú értékre alkalmazva `´a list` típusú értéket ad eredményül. Gyűjtőargumentummal javíthatunk `replist` hatékonyságán:

```
fun replist n x =
      let fun rpl 0 xs = xs
            | rpl n xs = rpl (n-1) (x::xs)
      in rpl n []
      end
```

Összefoglalva, a függvényalkalmazás olyan $E\,E_1$ alakú összetett kifejezés, amelyben az E függvényértéket eredményező, az E_1 pedig tetszőleges kifejezés. Az $E\,E_1\,E_2\,\cdots\,E_n$ kifejezés nem más, mint a $(\cdots((E\,E_1)\,E_2)\,\cdots\,E_n)$ kifejezés rövidítése.

Mint tudjuk, az SML-értelmezők a kifejezéseket *balról jobbra* haladva értékelik ki. A függvényalkalmazás *erősen köt,* precedenciája a lehető legnagyobb. A `->` típusoperátor (a *leképezés* jele) *jobbra köt,* ezért például a `string -> (string -> string)` típuskifejezés ekvivalens a `string -> string -> string` típuskifejezéssel (az SML-értelmezők a típuskifejezést redundáns zárójelek nélkül írják ki a képernyőre).

A részlegesen *nem* alkalmazható függvényt *uncurried* függvénynek is nevezik. Ha egy részlegesen nem alkalmazható függvény típusa `(´a * ´b) -> ´c`, akkor vele ekvivalens, részlegesen alkalmazható változatának a típusa `´a -> (´b -> ´c)`.

9.3. Függvény mint argumentum és mint eredmény

Nézzük a beszúró rendezést megvalósító `inssort` függvényt! (A rendezésről részletesen később, a 13. fejezetben lesz szó.)

```
(* inssort LE ls = ls elemeinek az LE reláció szerint rendezett listája
   inssort : ** vezesse le önállóan a függvény típusát!  **
*)
fun inssort LE ls =
      let infix LE
          (* ins(x, ys) = az ys elemeiből és az ys-be az LE reláció
                          szerint beszúrt x-ből álló, rendezett lista
             ins : ´a * ´a list -> ´a list
             PRE: ys az LE reláció szerint rendezve van
          *)
          fun ins (x, []) = [x]
            | ins (x, y::ys) = if x LE y
                               then x::y::ys
                               else y::ins(x, ys)
          (* sort xs = xs elemeinek LE szerint rendezett listája
             sort : ´a list -> ´a list
          *)
          fun sort [] = []
            | sort (x::xs) = ins(x, sort xs)
      in
         sort ls
      end
```

`sort` végigmegy a rendezendő lista minden elemén, és `ins` segítségével egyesével beilleszti az elemeket a helyükre. A magyarázatban használt `PRE` szócska *előfeltételt* (precondition) jelöl: az utána álló (logikai) *kijelentésnek* a függvény kiértékelése előtt *teljesülni kell* ahhoz, hogy a függvény helyes értéket adjon eredményül. Az `ls` paraméter elhagyható a definícióból, elhagyását mégsem javasoljuk, mert nélküle nehezebb megérteni a függvény működését:

```
fun inssort LE =
      let
          ...
      in
         sort
      end
```

Mielőtt folytatná az olvasást, gyakorlásképpen vezesse le `inssort` típusát!

```
inssort : (´a * ´a -> bool) -> ´a list -> ´a list
```

`inssort` itt bemutatott változata *generikus* (általános), mert a rendezési relációt megvalósító függvényt *paraméterként* adjuk át. Nézzünk két példát `inssort` alkalmazására:

```
inssort op <= [5, 3, 7, 5, 9]
inssort op >= [5, 3, 7, 5, 9]
```

Mint tudjuk, az `op` szócska és a műveleti jel közötti szóköz elmaradhat. `inssort` újabb alkalmazásához új relációt definiálunk füzérek lexikografikus rendezésére:

```
(* leStrPair(s, t) = igaz, ha s lexikografikusan nem nagyobb t-nél
   leStrPair : (string * string) * (string * string) -> bool
*)
fun leStrPair ((a, b), (c : string, d : string)) =
      a < c orelse (a = c andalso b <= d)
```

`inssort leStrPair` füzérpárokból álló listák lexikografikus rendezésére használható, pl.

```
inssort leStrPair [("Kovács", "Tibor"), ("Bihari", "Tamás")]
```

10. fejezet

Magasabb rendű függvények

10.1. Magasabb rendű függvények

A magasabb rendű függvények (angolul *higher-order functions* vagy *functionals)* többek között arra használhatók, hogy előre definiált magasabb rendű függvények alkalmazásával elkerüljük az explicit rekurziót, ezáltal könnyebben olvasható és bizonyítható programokat írjunk.

10.1.1. `secl` és `secr`

Gyakran hasznos, ha egy *infix* operátor egyik operandusát rögzítjük, a másikat szabadon hagyjuk, például

`("Sir " ^)`	ekvivalens a `knightify` függvénnyel,
`(/ 2.0)`	olyan függvény, amely `2.0`-vel oszt.

Sajnos, ezek a jelölések így nem használhatók az SML-ben, `secl` és `secr` segítségével azonban éppen ilyen függvényeket definiálhatunk. A nevet záró `l`, ill. `r` betű arra utal, hogy a bal *(left),* ill. a jobb *(right)* operandust kötjük-e le.

```
(* secl x f y = f bal oldali argumentumát lekötő függvény
   secl : ´a -> (´a * ´b -> ´c) -> ´b -> ´c
*)
fun secl x f y = f(x, y)

(* secr f y x = f jobb oldali argumentumát lekötő függvény
   secr : (´a * ´b -> ´c) -> ´b -> ´a -> ´c
*)
fun secr f y x = f(x, y)
```

`secl` és `secr` paramétereinek nevét és sorrendjét úgy választottuk meg, hogy egyrészt utaljanak a paraméterek szerepére és pozíciójára, másrészt tegyék lehetővé `secl` és `secr` *részleges* alkalmazását. Mielőtt folytatná az olvasást, vezesse le önállóan a két függvény típusát!

```
secl : ´a -> ((´a * ´b) -> ´c) -> ´b -> ´c
secr : ((´a * ´b) -> ´c) -> ´b -> ´a -> ´c
```

Végül bemutatunk néhány példát a két függvény alkalmazására.

```
(* knightify n = a "Sir " és n konkatenációja
   knightify : string -> string
```

```
*)
val knightify = secl "Sir " op^

(* recip r = az r valós szám reciproka
   recip : real -> real
*)
val recip = secl 1.0 op/

(* halve r = az r valós szám fele
   halve : real -> real
*)
val halve = secr op/ 2.0
```

10.1.2. Kombinátorok

A kombinátoroknak elsősorban *elvi* szempontból nagy a jelentősége (v.ö. λ-kalkulus).

10.1.2.1. Két függvény kompozíciója

Két függvény kompozícióját általában a $\circ$ operátorral jelölik, mi az o betűt fogjuk használni.

```
(* f o g = az f és g függvények kompozíciója
   o : (´a -> ´b) * (´c -> ´a) -> ´c -> ´b
*)
infix o;
fun (f o g) x = f(g x)
```

Nézzünk egy példát o alkalmazására, írjunk összegző függvényt a $\sum_{i=0}^{m-1} f(i)$ kifejezés kiszámítására! A kifejezésben az m és az f ún. *szabad* változók, az i ún. kötött változó, a 0 és az 1 pedig állandók.

```
(* summa f m = az f(i) értékek összege a 0<=i<m tartományban
   summa : (int -> real) -> int -> real
*)
fun summa f m =
      let
         fun sum (i, z) : real =
               if i = m then z else sum(i+1, z + f i)
      in
         sum(0, 0.0)
      end
```

A `sum` segédfüggvény a hatékonyságot javítja, mert *iteratív:* a `z` argumentumban gyűjti az eredményt. Most alkalmazzuk a függvényt a $\sum_{k=0}^{10-1} \sqrt{k}$ kifejezés kiszámítására![1]

```
summa (sqrt o real) 10
```

Talán nem kell sokat bizonygatni, hogy az `sqrt o real` kompozíció alkalmazásával a felírt kifejezés jobban kifejezi a lényeget, és olvashatóbb is, mint a `summa(sqrt(real 10)))` alak, többek között azért, mert jobban hasonlít a matematikában megszokott jelölésmódra.

[1] `sqrt` gyanánt vagy a 14.4 szakaszban definiált `sqroot`, vagy a `Math` könyvtárbeli `Math.sqrt` függvény használható.

10.1.2.2. Az `S`, a `K` és az `I` kombinátor

Az `I` kombinátor a közismert *identitásfüggvény*.

```
fun I x = x
```

A `K` kombinátor egy állandóra alkalmazva ún. *konstansfüggvényt* állít elő: olyan függvényt, amelyet bármilyen argumentumra alkalmazunk is, mindig ezt az állandót adja eredményül.

```
fun K x y = x
```

Ha tehát a `K x` konstansfüggvényt tetszőleges `y` argumentumra alkalmazzuk, `x`-et kapunk eredményül. Vajon mi lesz az alábbi kifejezés eredménye? Miért nem írhatunk egyszerűen 7-et az első argumentum helyére?

```
summa (K 7.0) 5
```

Az `S` kombinátor neve: *általános kompozíció.* SML-definíciója:

```
fun S x y z = x z (y z)
```

A λ-kalkulus szerint *minden függvény* felírható *kizárólag* `K` és `S` alkalmazásával, változók nélkül! Ez `K` és `S` igazi jelentősége, gyakorlati szempontból nem olyan fontosak. Az érdekesség kedvéért megmutatjuk, milyen egyszerű `I` definíciója `K`-val és `S`-sel:

```
fun I x = S K K x
```

Nézzünk egy példát `S K K 7` egyszerűsítésére:

```
S K K 7 → K 7 (K 7) → 7
```

10.1.3. `map` és `filter`

`map` egy paraméterként átadott függvényt alkalmaz egy lista minden elemére, eredménye egy új lista. `filter` egy listából összegyűjti és egy új listába fűzi azokat az elemeket, amelyek a paraméterként átadott *predikátumot* kielégítik.

```
(* map f ls = az ls elemeiből az f transzformációval előálló elemek listája
   map : (´a -> ´b) -> ´a list -> ´b list
*)
fun map f [] = []
  | map f (x::xs) = f x :: map f xs

(* filter p ls = ls elemei közül a p predikátumot kielégítő elemek listája
   filter : (´a -> bool) -> ´a list -> ´a list
*)
fun filter p [] = []
  | filter p (x::xs) = if p x then x :: filter p xs else filter p xs
```

Lássunk néhány példát az alkalmazásukra!

```
- map (map (fn n => n * 2)) [[1], [2, 3], [4, 5, 6]];
> val it = [[2], [4, 6], [8, 10, 12]]: int list list
```

Válaszoljon önállóan a következő kérdésekre: mi az alábbi függvénykifejezések típusa, és mi a kiértékelésük eredménye, ha mindkettőt az `[["abc","def"], ["mnoprq"], ["","xy"]]` listára alkalmazzuk?

1. `map (map (implode o rev o explode))`
2. `map (filter (secr op< "m"))`

Két halmaz *metszete* ($S \cap T$) például így definiálható `filter`-rel (`ss`-ben S, `ts`-ben T elemeit tároljuk):

```
(* inter(ss, ts) = az ss és ts halmazok metszete
   inter : ''a list * ''a list -> ''a list
```

   PRE: $\forall i, j \bullet i \neq j,\ s_i \in ss,\ s_j \in S \bullet s_i \neq s_j, \forall i, j \bullet i \neq j,\ t_i \in T,\ t_j \in T \bullet t_i \neq t_j$

```
*)
fun inter (ss, ts) = filter (secr (op isMem) ts) ss
```

Az `isMem` függvényt a 7.2 szakaszban *infix* operátorként definiáltuk.

10.1.4. `takewhile` és `dropwhile`

Korábban `take`-kel és `drop`-pal találkoztunk: `take` egy lista elejéről vett adott számú elemből, `drop` egy lista elejéről adott számú elem elhagyásával képez listát. Néha olyan függvényekre van szükség, amelyek egy lista elejéről vett, adott predikátumot kielégítő elemekből, ill. egy lista elejéről adott predikátumot kielégítő elemek elhagyásával képeznek listát. Ilyen függvényeket írunk most `takewhile`, ill. `dropwhile` néven.

```
(* takewhile p xs = az xs elejéről vett, p-t kielégítő elemek listája
   takewhile : (´a -> bool) -> ´a list -> ´a list
*)
fun takewhile p [] = []
  | takewhile p (x::xs) = if p x then x::takewhile p xs else []

(* dropwhile p xs = xs elejéről a p-t kielégítő elemek elhagyásával
                    előálló lista
   dropwhile : (´a -> bool) -> ´a list -> ´a list
*)
fun dropwhile p [] = []
  | dropwhile p (x::xs) = if p x then dropwhile p xs else x::xs
```

`dropwhile`-ban *réteges mintát* is alkalmazhatunk:

```
fun dropwhile p [] = []
  | dropwhile p (xxs as x::xs) = if p x then dropwhile p xs else xxs
```

10.1.5. `exists` és `forall`

`exists` és `forall` a logikából jól ismert *kvantorok* ($\exists$, $\forall$) megvalósítása SML-ben:

```
(* exists p xs = igaz, ha xs-nek van p-t kielégítő eleme
   exists : (´a -> bool) -> ´a list -> bool
*)
fun exists p [] = false
  | exists p (x::xs) = p x orelse exists p xs

(* forall p xs = igaz, ha xs összes eleme kielégíti p-t
   forall : (´a -> bool) -> ´a list -> bool
*)
fun forall p [] = true
  | forall p (x::xs) = p x andalso forall p xs
```

A korábban már definiált `isMem` függvényt újradefiniálhatjuk `exists` alkalmazásával:

```
(* x isMem xs = igaz, ha x eleme xs-nek
   isMem : ''a * ''a list -> bool
*)
infix isMem;
fun x isMem xs = exists (secl x op=) xs
```

`forall` segítségével definiálhatjuk a halmazok diszjunkt voltát tesztelő `disjoint` függvényt:

```
(* disjoint(xs, ys) = igaz, ha xs és ys metszete üres
   disjoint : ´a list * ´a list -> bool
*)
fun disjoint (xs, ys) = forall (fn x => forall (fn y => x <> y) ys) xs
```

Az utóbbit első ránézésre elég nehéz megérteni. Próbáljuk meg együtt! A függvénynek akkor kell igaz értéket adnia, ha az `xs` és az `ys` által ábrázolt halmazoknak egyetlen közös eleme sincs.

Az `fn y => x <> y` függvény akkor ad igaz értéket, ha `y` nem egyenlő valamilyen – e függvény számára külső és rögzített – `x` értékkel. Ezt a függvényt a zárójelen belüli `forall` hívás az összes `ys`-beli értékkel meghívja, és akkor ad igazat eredményül, ha egyetlen `ys`-beli érték sem egyenlő `x`-szel.

Az `fn x => forall ··· ys` függvény vezeti be `x`-et mint argumentumot. A külső `forall` az összes `xs`-beli értékkel meghívja ezt a függvényt, és akkor igaz az eredménye, ha nincs olyan `ys`-beli érték, amely egyenlő lenne valamely `xs`-beli értékkel.

10.1.6. `foldl` és `foldr`

`foldl` balról jobbra (**l**eft to right), `foldr` jobbról balra (**r**ight to left) haladva egy *kétargumentumú prefix* függvényt (pl. `op+`, `op*`) alkalmaz egy lista minden elemére. A két függvény specifikációját *infix* operátorral (⊕) írjuk föl, mert *infix* jelöléssel könnyebb megérteni a működésüket (a⊕ tetszőleges infix operátort helyettesít).

```
foldl op⊕ e [x1, x2, ···, xn] = (xn⊕···⊕(x2⊕(x1⊕e))···)
foldr op⊕ e [x1, x2, ···, xn] = (x1⊕(x2⊕···⊕(xn⊕e)···))
```

Egyes műveletekben (pl. lista elemeinek összeadása és szorzása, elem lista elé fűzése) az `e` gyakran az adott művelet egységeleme. Látható, hogy ha elvégezzük a kijelölt műveleteket, mindkét függvény *egyszerűsíti* az eredeti kifejezést. Kifejezéseket általában – de nem mindig! – jobbról balra haladva egyszerűsítünk, ezért `foldr`-t néha `reduce` néven definiálják.

Asszociatív műveletek (pl. összeadás és szorzás) esetén mindegy, hogy `foldl`-t vagy `foldr`-t alkalmazzuk-e.

`foldl`-t és `foldr`-t nem specifikáltuk arra az esetre, amikor a lista üres, pótoljuk:

`foldl op⊕ e [] = e` és `foldr op⊕ e [] = e`

Jól látszik, hogy `e` valóban lehet az `op⊕` művelet egységeleme, hiszen `op⊕`-t az üres listára alkalmazva `e`-t kapjuk eredményül. Most már definiálhatjuk `foldl` és `foldr` SML-változatát.[2] Vegyük észre, hogy `e` *gyűjtőargumentumként* viselkedik.

```
(* foldl f e xs = az xs elemeire balról jobbra haladva alkalmazott,
                  kétoperandusú, e gyűjtőargumentumú f művelet eredménye
   foldl : (´a * ´b -> ´b) -> ´b -> ´a list -> ´b
*)
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
  | foldl f e [] = e
```

[2]Mindkettő belső függvény az SML-ben.

```
(* foldr f e xs = az xs elemeire jobbról balra haladva alkalmazott,
                  kétoperandusú, e gyűjtőargumentumú f művelet eredménye
   foldr : (´a * ´b -> ´b) -> ´b -> ´a list -> ´b
*)
fun foldr f e (x::xs) = f(x, foldr f e xs)
  | foldr f e [] = e
```

Mindkét függvénynek `´a * ´b -> ´b` típusú függvény az első argumentuma. Vegyük észre, hogy `foldl` *jobbrekurzív.* Sajnos, `foldr` a veremben gyűjti a részeredményeket, és majd csak a lista kiürülésekor hajtja végre a kijelölt műveleteket.

A két függvény definícióját és típusát nem is olyan könnyű megjegyezni. Javasoljuk az olvasónak, hogy csukja be a jegyzetet, és próbálja meg *a specifikáció alapján* rekonstruálni a két definíciót és levezetni a típust!

Számos függvény írható fel `foldl` és `foldr` alkalmazásával, ha megadjuk a lista szomszédos elemein végrehajtandó műveletet, valamint az egységelemet, ill. a gyűjtőargumentum kezdőértékét. Lássunk néhányat:

```
(* sum xs = xs elemeinek összege
   sum : int list -> int
*)
fun sum xs = foldl op+ 0 xs

(* prod xs = xs elemeinek szorzata
   prod : int list -> int
*)
fun prod xs = foldl op* 1 xs

(* flat xss = az xss részlistáinak konkatenálásával előálló lista
   flat : ´a list list -> ´a list
*)
fun flat xss = foldl op@ [] xss
```

A `length` függvény egy iteratív változata (`inc` olyan kétargumentumú segédfüggvény, amelyik nem használja a második argumentumát):

```
local (* inc(n,_) = n+1
         inc : int * ´a -> int
      *)
      fun inc (n, _) = n + 1
in
   (* length ls = az ls lista hossza
      length : ´a list -> int
   *)
   fun length ls = foldl inc 0 ls
end
```

Az `append` függvény is felírható így, de `foldr`-rel, mert a `::` művelet nem asszociatív és jobbra köt. Gyűjtőargumentumnak azt a listát vesszük, amelyhez a bal oldali lista elemeit – jobbról balra haladva – egyesével fűzzük hozzá .

```
(* append xs ys = az xs ys elé fűzésével előálló lista
   append : ´a list -> ´a list -> ´a list
*)
fun append xs ys = foldr op:: ys xs
```

A beszúró rendezés egy újabb változata `foldr`-rel (vö. a 13. fejezet):

```
(* ins(x, ys) = az ys rendezett egészlista elemeiből és az ys-be a
               <= reláció szerint beszúrt x-ből álló lista
   ins : int * int list -> int list
*)
fun ins (x, []) = [x]
  | ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)

(* inssort ls = az ls egészlista elemeinek <= szerint rendezett listája
   inssort : int list -> int list
*)
fun inssort xs = foldr ins [] xs
```

10.1.7. További rekurzív függvények

Egy függvény n-edik hatványát így specifikálhatjuk: $f^n = f(\cdots f(f(x))\cdots)$, ha $n \geq 0$ (f n-szer ismétlődik). Definiáljunk SML-függvényt ilyen ismétlések felírására!

```
(* repeat f n x = f n-edik hatványa az x helyen
   repeat : (´a -> ´a) -> int -> ´a-> ´a
*)
fun repeat f n x = if n > 0 then repeat f (n-1) (f x) else x
```

Sok függvény fejezhető ki `repeat` segítségével. Példák:

```
(* drop(xs, k) = xs első k elemének elhagyásával előálló lista
   drop : ´a list * int -> ´a list
*)
fun drop (xs, k) = repeat tl k xs

(* replist k = füzér, amelyben "Ha!" k-szor ismétlődik
   replist : int -> string list
*)
fun replist k = repeat (secl "Ha!" op::)  k []
```

10.1.8. `curry` és `uncurry`

Most már könnyen definiálhatunk egy-egy olyan függvényt, amelyik egy részlegesen alkalmazható függvényt részlegesen nem alkalmazható függvénnyé, ill. egy részlegesen nem alkalmazható függvényt részlegesen alkalmazható függvénnyé alakít:

```
(* curry f x y = f részlegesen nem alkalmazható alakban
   curry : (´a * ´b -> ´c)-> ´a -> ´b -> ´c
*)
fun curry f x y = f(x,y)

(* uncurry f(x, y) = f részlegesen alkalmazható alakban
   uncurry : (´a -> ´b -> ´c) -> (´a * ´b) -> ´c
*)
fun uncurry f(x,y)= f x y
```

10.1.9. `map` újradefiniálása `foldr`-rel

`map` definiálásához mintául szolgálhat a listaelemek összegét képező `sum` függvény definíciója pl. `foldr`-rel:

```
fun sum xs = foldr op+ 0 xs
```

Idézzük föl `map` rekurzív definícióját is:

```
fun map f (x::xs) = f x :: map f xs
  | map f [] = []
```

A definícióban a `::`-ot használjuk inkább *prefix* alakban, hogy jobban lássuk, mit kell tennünk:

```
fun map f (op::(x, xs)) = op::(f x, map f xs)
  | map f [] = []
```

Látható, hogy az `xs` lista minden elemére alkalmazni kell előbb az `f`, majd az `op::` függvényt, jó lenne tehát a kompozíciójukat használni. Csakhogy `op::` részlegesen *nem* alkalmazható, argumentumként párt váró függvény, az egyargumentumú `f`-fel pedig csak részlegesen alkalmazható változatát komponálhatjuk: `(curry op:: o f)`. Igen ám, de `foldr` első argumentuma argumentumként párt váró, részlegesen nem alkalmazható függvény, ezért a `(curry op:: o f)` függvényt `uncurry` segítségével még részlegesen *nem* alkalmazhatóvá kell tennünk `map` új változatához:

```
fun map f xs = foldr (uncurry(curry op:: o f)) [] xs
```

Vegyük jobban szemügyre az `uncurry(curry op:: o f)` kifejezést (ahol `f` még lekötetlen azonosító)! Argumentuma egy pár, e pár első tagja valamilyen érték, második tagja egy lista, az eredménye pedig egy ugyanilyen típusú lista. A függvény a pár első tagján elvégzi az `f` transzformációt, majd a kapott értéket a pár második tagjához fűzi:

```
- fn f => uncurry(curry op:: o f);
> val it = fn : (´a -> ´b) -> (´a * ´b list -> ´b list)
```

Mit tagadjuk, `uncurry(curry op:: o f)` elég csúnyácska kifejezés, írjuk fel inkább más alakban:

```
- fn f => fn (x, ys) => (op:: (f x, ys))
> val it = fn : (´a -> ´b) -> (´a * ´b list -> ´b list)
```

Ezzel eljutottunk `map` egy újabb, tisztább változatához:

```
fun map f xs = foldr (fn (x, ys) => op::(f x, ys)) [] xs
```

11. fejezet

Kivételkezelés

Kivételnek (exception) nevezzük a különleges elbánást igénylő eseteket: a különféle futási hibák (0-val való osztás, túlcsordulás, lista kiürülése, nemlétező állomány megnyitása stb.) fellépését, a programmegszakítást stb. A kivételkezeléshez három szintaktikai elem – `exception`, `raise`, `handle` – jelentésével és használatával kell megismerkednünk.

Az SML-ben a kivételt a függvények mindaddig továbbpasszolják az őket hívó függvényeknek, végső esetben az SML keretrendszernek, amíg egy *kivételkezelő* fel nem ismeri, hogy a kivételt neki kell feldolgoznia.

A kivételkezelő olyan speciális, a `case`-hez hasonló kifejezés, amelyik megmondja, hogy egyes kivételek jelentkezése esetén mit kell tenni és milyen értéket kell eredményül adni.

11.1. Kivétel deklarálása az `exception` kulcsszóval

A belső típusok között van egy különleges típus, az `exn`. Különleges, mert más adattípusokkal ellentétben a *kivételkonstruktorok* halmaza *bővíthető*. Például az

```
exception Failure
```

deklaráció a `Failure` *kívételállandóval* (különleges típuskonstruktorállandóval) bővíti az `exn` típusú értékek (a kivételkonstruktorok) halmazát. A következő példa két *kivételfüggvénnyel* (különleges típuskonstruktorfüggvénnyel) bővíti a konstruktorhalmazt:

```
exception FailedBecause of string;
exception BadValue of int
```

`Failedbecause` típusa `string -> exn`, `Badvalue` típusa `int -> exn`.

Kivételt lokálisan is lehet deklarálni, de nem célszerű, hiszen ha pl. ugyanazon a néven több kivételt is jelezhet a futtatórendszer, az nehezíti a hiba lokalizálását, a program megértését.

Az `exn` típusú érték sok szempontból ugyanolyan, mint más értékek: listába fűzhető, függvény argumentuma és eredménye lehet stb. Különleges a szerepe azonban a `raise` és a `handle` kulcsszóval kezdődő kifejezésekben.

11.2. Kivétel jelzése a `raise` kulcsszóval

A `raise` kulcsszó olyan ún. *kivételcsomagot* (exception packet) hoz létre, amelyben `exn` típusú érték van.

Ha az `E` kifejezés kiértékelése `exn` típusú `e` értéket ad eredményül, akkor a `raise E` kifejezés az `e` értéket tartalmazó kivételcsomagot eredményez. Az ilyen kivételcsomagot csak a `handle` kulcsszóval kezdődő kifejezések képesek kezelni, általános értelemben nem tekinthetők értéknek az SML-ben. Az `exn` típus „közvetít” a kivételcsomagok és más SML-értékek között.

A kivételkezelés során az SML-értelmező a kivételcsomagokat az érték szerinti paraméterátadás szabályai szerint adja tovább. Ha egy `E` kifejezés eredménye egy kivételcsomag, akkor tetszőleges `f`-re `f(E)` eredménye is egy kivételcsomag. `f(raise E)` ekvivalens `raise E`-vel (pl. `raise(Badvalue(raise Failure))` ekvivalens `raise Failure`-rel).

Tudjuk, hogy az SML minden kifejezést balról jobbra és fölülről lefelé haladva értékel ki. Ezért ha `E1` eredménye egy kivételcsomag, akkor az `(E1, E2)` párban `E2` kiértékelésére nem is kerül sor. Ha `E2` eredménye a kivételcsomag, `E1`-é pedig valamilyen más érték, akkor az `(E1, E2)` pár eredménye a kivételcsomag lesz.

Az `if E then E1 else E2` feltételes kifejezésben nemcsak `E1` és `E2`, hanem `E` kiértékelésének is lehet kivételcsomag az eredménye. Ha a `let val p = E1 in E2 end` kifejezésben `E1` eredménye egy kivételcsomag, akkor az egész `let`-kifejezés eredménye is ez a kivételcsomag.

11.2.1. Belső kivételek

Az SML legfontosabb belső kivételkonstruktorait az alábbi táblázatban soroljuk föl.

Megnevezés	*Művelet, amely a kivételt kiválthatja*
`Bind`	
`Chr`	`chr pred succ`
`Div`	`/ div mod`
`Domain`	
`Empty`	`hd tl last`
`Fail`	`compile load loadOne`
`Interrupt`	
`Io`	
`Match`	
`Option`	
`Ord`	
`Overflow`	`~ + - * / div mod abs ceil floor round trunc`
`Size`	`^ array concat fromList implode tabulate translate vector`
`Subscript`	`copy drop extract nth sub substring take update`

11.3. Kivétel feldolgozása a `handle` kulcsszóval

A kivétel feldolgozása a `case`-szerkezetre emlékeztet:

```
E handle P1 => E1 | ··· | Pn => En
```

A fenti kifejezésben a `handle` kulcsszóval kezdődő részkifejezést *kivételkezelő* nevezzük. Ha `E` “közönséges” értéket ad eredményül, akkor a kivételkezelő, mintha ott sem lenne, egyszerűen továbbadja az eredményt. De ha `E` *kivételcsomagot* eredményez, akkor a tartalmát az SML-futtatórendszer megpróbálja a megadott mintákra illeszteni. Ha az első illeszkedő minta a `Pi` ($i = 1, 2, \ldots, n$) , akkor a kivételkezelő eredménye az `Ei` kifejezés eredménye lesz. Ha egyetlen minta sem illeszthető a kivételcsomagra, akkor a kivételkezelő továbbpasszolja a kivételcsomagot az előző hívási szintre.

11.4. Néhány példa a kivételkezelésre

Három kis példát mutatunk be. Mindhárom esetben először deklaráljuk a kivételt, majd olyan függvényt írunk, amely jelzi a kivétel bekövetkezését, végül olyan próbafüggvényeket készítünk, amelyek a kivétel feldolgozását illusztrálják.

Az első példában a `Demo1` konstruktorfüggvény `string -> exn` típusú. `test1` normális működés esetén `string` típusú értéket ad eredményül, ezért a `handle` kivételkezelőnek is `string` típusú értéket *kell* eredményeznie.

```
exception Demo1 of string;
(* demo1 : int * string -> string
*)
fun demo1 (5,s) = s
  | demo1 (_,s) = raise Demo1("Not five but " ^ s);
(* test1 : int * string -> string
*)
fun test1 (x,s) = demo1(x,s) handle Demo1 m => "Exception1: " ^ m;

- test1(5,"five");
> val it = "five" : string
- test1(3,"three");
> val it = "Exception1: Not five but three" : string
```

A második példában a `Demo2` konstruktorfüggvény `int * string -> exn` típusú. `test2` normális működés esetén `int * string` típusú értéket ad eredményül, ezért a `handle` kivételkezelőnek is `int * string` típusú értéket *kell* eredményeznie.

```
exception Demo2 of int * string;
(* demo2 : int * string -> int * string
*)
fun demo2 (5,s) = (5,s)
  | demo2 (x,s) = raise Demo2(~5555,"Not five but " ^ s);
(* test2 : int * string -> int * string
*)
fun test2 (x,s) = demo2(x,s) handle Demo2(y,m) => (y, "Exception2: " ^ m);

- test2(5,"five");
> val it = (5,"five") : int * string
- test2(3,"three");
> val it = (3,"Exception2: Not five but three") : int * string
```

A harmadik példában a `Demo3` konstruktorfüggvény, `Demo1`-hez hasonlóan, `string -> exn` típusú. `test3` normális működés esetén, `test2`-höz hasonlóan, `int * string` típusú értéket ad eredményül, ezért a `handle` kivételkezelőnek most is `int * string` típusú értéket *kell* eredményeznie.

```
exception Demo3 of string;
(* demo3 : int * string -> int * string
*)
fun demo3 (5,s) = (5,s)
  | demo3 (_,s) = raise Demo3("Not five but " ^ s);
(* test3 : int * string -> int * string
*)
fun test3 (x,s) = demo3(x,s) handle Demo3 m => (x,"Exception3: " ^ m);

- test3(5,"three");
> val it = (5,"five") : int * string
- test3(3,"three");
> val it = (3,"Exception3: Not five but three") : int * string
```

demo1, demo2 és demo3 is meghívható kivételkezelő alkalmazása nélkül, de ilyenkor a program végrehajtása kivétel fellépésekor félbeszakad. Tegyük föl például, hogy a test30.sml állományban az alábbi program van:

```
fun test30 (x,s) = demo3(x,s);
test30(5,"five");
test30(3,"three");
val s = "s már nem kap értéket";
```

Amikor az SML-értelmező a use "test30.sml" függvényalkalmazás hatására ezt a programot beolvassa, a feldolgozás a test30(3, "three") kifejezés kiértékelése közben a kivétel fellépése miatt félbeszakad, az s név deklarására sohasem kerül sor.

```
- use "test30.sml";
[opening file "test30.sml"]
> val test30 = fn : int * string -> int * string
> val it = (5,"five") : int * string
!  Uncaught exception:
!  Demo3 "Not five but three"
[closing file "test30.sml"]
```

12. fejezet

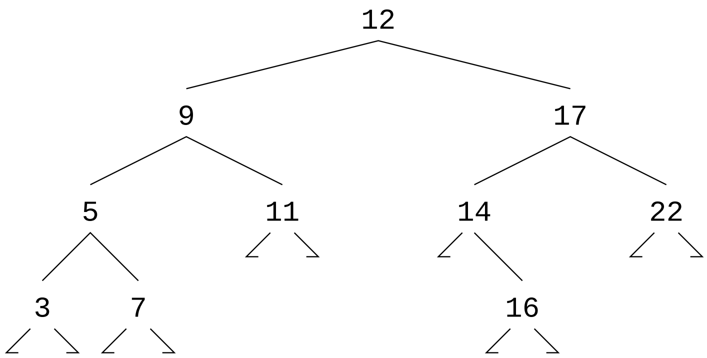
Bináris fák

A listához hasonlóan rekurzív adattípus a fa. Ebben a fejezetben bináris fák deklarációját és használatát mutatjuk be.

Először olyan bináris fát deklarálunk, amelynek a levelei üresek, a csomópontjaiban pedig előbb a bal részfa, majd az `´a` típusú érték, és végül a jobb részfa van:

```
datatype ´a tree = L | B of ´a tree * ´a * ´a tree
```

Tekintsük például az alábbi fát:



Az `´a tree` adattípus `L` és `B` adatkonstruktoraival ez a fa így írható le:

```
B(B(B(B(L,3,L),
      5,
      B(L,7,L)
      ),
    9,
    B(L,11,L)
   ),
  12,
  B(B(L,
      14,
      B(L,16,L)
     ),
    17,
    B(L,22,L)
   )
 )
```

A fastruktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorokat:

```
                        B( 12

          B( 9                        B( 17

   B( 5        B( 11         B( 14          B( 22

                L,  ,L)),    L,              L,  ,L)))
B( 3  B( 7                      B( 16

L, ,L),L,,L)),                  L,  ,L)),
```

Ezt bizony elég nehéz átlátni! A leírás áttekinthetőbbé tehető, ha az egyes részfáknak nevet adunk:

```
val tr3 = B(L,3,L);
val tr7 = B(L,7,L);
val tr5 = B(tr3,5,tr7);
val tr11 = B(L,11,L);
val tr9 = B(tr5,9,tr11);
val tr16 = B(L,16,L);
val tr14 = B(L,14,tr16);
val tr22 = B(L,22,L);
val tr17 = B(tr14,17,tr22);
val tr12 = B(tr9,12,tr17);
```

Természetesen másféle fastruktúrákat is deklarálhatunk, pl. kezdhetjük az ´a típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával. Felhasználhatjuk a levelet is értékek tárolására, vagy előírhatjuk, hogy csak a levélben lehet érték stb. A

```
datatype ´a tree = E | L of ´a | B of ´a tree * ´a * ´a tree
```

deklaráció például abban különbözik a korábban már látott

```
datatype ´a tree = L | B of ´a tree * ´a * ´a tree
```

deklarációtól, hogy a bináris fa leveleiben is tárolunk értéket, az értéket nem tároló üres csonkokat pedig `E`-vel jelöljük.

A rekurzív függvényekhez hasonlóan a rekurzív adattípusoknak is kell hogy legyen triviális esete. Szintaktikailag helyesek az alábbi deklarációk is, de a triviális eset hiánya miatt alkalmatlanok adatok létrehozására:

```
datatype ´a badtree = B of ´a badtree * ´a * ´a badtree

datatype ´a badtree = L of ´a badtree | B of ´a badtree * ´a * ´a badtree
```

12.1. Egyszerű műveletek bináris fákon

Most bináris fákra alkalmazható, jól ismert műveletekre írunk SML-függvényeket. A példákban az alábbi típusdeklarációt használjuk:

```
datatype ´a tree = L | N of ´a * ´a tree * ´a tree
```

`nodes` egy fa csomópontjait számlálja meg. Ehhez hasonlóan számolhatók meg a fa levelei.

```
(* nodes f = az f fa csomópontjainak a száma
   nodes : ´a tree -> int
*)
fun nodes L = 0
  | nodes (N(_, t1, t2)) = 1 + nodes t1 + nodes t2
```

`nodes` gyűjtőargumentumot használó változatának kisebb a tárigénye:

```
(* nodes f = az f fa csomópontjainak a száma
   nodes : ´a tree -> int
*)
fun nodes f =
      let (* nodes0(f, n) = n + a csomópontok száma f-ben
                nodes0 : ´a tree * int -> int
          *)
          fun nodes0 (L, n) = n
            | nodes0 (N(_, t1, t2), n) = nodes0(t1, nodes0(t2, n+1))
      in
          nodes0(f, 0)
      end
```

`depth` egy fa mélységét (más szóhasználattal a magasságát) határozza meg. A fa gyökeréből a leveléhez vezető úton az élek számát (az út hosszát) az adott levél szintjének is nevezzük. A szintek közül a legnagyobbat a fa mélységének hívjuk.

```
(* depth f = az f fa mélysége
   depth : ´a tree -> int
*)
fun depth L = 0
  | depth (N(_,t1,t2)) = 1 + Int.max(depth t1, depth t2)
```

`depth` gyűjtőargumentumot használó változata:

```
(* depth f = az f fa mélysége
   depth : ´a tree -> int
*)
fun depth f =
      let fun depth0 (L, d) = d
            | depth0 (N(_,t1,t2), d) =
                           Int.max(depth0(t1, d+1), depth0(t2, d+1))
      in
          depth0(f, 0)
      end
```

`fulltree` n mélységű *teljes bináris fát* épít, és a fa csomópontjait 1-től $2^n - 1$-ig beszámozza. Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és összes levele ugyanazon a szinten van.

```
(* fulltree n = n mélységű teljes fa
   fulltree : int -> ´a tree
*)
fun fulltree n =
      let fun ftree (_, 0) = L
            | ftree (k, n) = N(k, ftree(2*k, n-1), ftree(2*k+1, n-1))
      in ftree(1, n)
      end
```

`reflect` a fát a függőleges tengelye mentén tükrözi.

```
(* reflect =
   reflect : ´a tree -> ´a tree
*)
fun reflect L = L
  | reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
```

12.2. Lista előállítása bináris fa elemeiből

`preorder`, `inorder` és `postorder` *bináris fából listát* állít elő. Ahogy a nevük is sugallja, a három függvény abban különbözik egymástól, hogy az egy csomópontból az ott tárolt értéket mikor veszik ki, és milyen sorrendben járják be a bal, ill. a jobb részfát.

`preorder` először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát. `inorder` először bejárja a bal részfát, majd kiveszi az értéket, és végül bejárja a jobb részfát. `postorder` először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.

Az alábbi megvalósítások egyszerűek, érthetőek, de nem elég hatékonyak (a `@` operátor használata miatt).

```
(* preorder f = az f fa elemeinek preorder sorrendű listája
   preorder : ´a tree -> ´a list
*)
fun preorder L = []
  | preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2

(* inorder f = az f fa elemeinek inorder sorrendű listája
   inorder : ´a tree -> ´a list
*)
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)

(* postorder f = az f fa elemeinek postorder sorrendű listája
   postorder : ´a tree -> ´a list
*)
fun postorder L = []
  | postorder (N(v,t1,t2)) = postorder t1 @ postorder t2 @ [v]
```

A gyűjtőargumentum használata miatt nehezebben érthetőek, de *hatékonyabbak* következő változataik.

```
(* preord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                   preorder sorrendű listája
   preord : ´a tree * ´a list -> ´a list
*)
fun preord (L, vs) = vs
  | preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))

(* inord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                  inorder sorrendű listája
   inord : ´a tree * ´a list -> ´a list
*)
fun inord (L, vs) = vs
  | inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))
```

```
(* postord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                    postorder sorrendű listája
   postord : ´a tree * ´a list -> ´a list
*)
fun postord (L, vs) = vs
  | postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))
```

12.3. Bináris fa előállítása lista elemeiből

Listát *kiegyensúlyozott bináris fává* alakít a következő függvény. (A `balPreorder` név a ***bal****anced* (kiegyensúlyozott) és a ***preorder*** szavakból származik.)

```
(* balPreorder xs = az xs lista elemeiből álló, preorder bejárású,
                    kiegyensúlyozott fa
    balPreorder: ´a list -> ´a tree
*)
fun balPreorder (x::xs) =
      let val k = length xs div 2
      in N(x, balPreorder(List.take(xs, k)), balPreorder(List.drop(xs, k)))
      end
  | balPreorder [] = L
```

A hatékonyságot kisebb mértékben rontja, hogy `List.take` és `List.drop` egymástól függetlenül *kétszer* mennek végig a lista első felén. Írjunk `take´ndrop` néven olyan függvényt, amelynek egy `xs` listából és egy `k` egészből álló pár az argumentuma, és ugyancsak egy pár az eredménye. E pár első tagja a lista első `k` db eleme, második tagja pedig a lista többi eleme legyen.

```
(* take´ndrop(xs, k) = olyan pár, amelynek első tagja xs első k db eleme,
                      második tagja pedig xs maradéka
   take´ndrop : ´a list * int -> ´a list * ´a list
*)
fun take´ndrop (xs, k) =
      let fun td (xs, 0, ts) = (rev ts, xs)
            | td (x::xs, k, ts) = td(xs, k-1, x::ts)
            | td ([], _, ts) = (rev ts, [])
      in
          td(xs, k, [])
      end
```

`take´ndrop` egy párt ad eredményül, ezért `balPreorder`-t módosítani kell.

```
(* balPreorder xs = az xs lista elemeiből álló, preorder bejárású,
               kiegyensúlyozott fa
    balPreorder: ´a list -> ´a tree
*)
fun balPreorder (x::xs) =
      let val k = length xs div 2
          val (ts, ds) = take´ndrop(xs, k)
      in N(x, balPreorder ts, balPreorder ds)
      end
  | balPreorder [] = L
```

Hatékonyságromlást okoz az is, hogy `balPreorder` minden meghívásakor kiszámítja `xs` aktuális hosszát. Ennek elkerülésére `balPreorder`-t egy segédfüggvénnyel egészítjük ki, amely `k`-t paraméterként kapja.

```
fun balPreorder xs =
      let fun bpo (x::xs, k) =
                let val (ts, ds) = take´ndrop(xs, k)
                    val k = (k - 1) div 2
                in N(x, bpo(ts, k), bpo(ds, k))
                end
            | bpo ([], _) = L
      in bpo(xs, (length xs - 1) div 2)
      end
```

`balPreorder`-hez hasonló `balInorder` és `balPostorder`; a lényegi különbség közöttük most is a bejárási sorrendben van.

```
(* balInorder xs = az xs lista elemeiből álló, inorder bejárású,
                   kiegyensúlyozott fa
    balInorder: ´a list -> ´a tree
*)
fun balInorder (xxs as x::xs) =
      let val k = length xxs div 2
          val (y::ys) = List.drop(xxs, k)
      in N(y, balInorder(List.take(xxs, k)), balInorder ys)
      end
  | balInorder [] = L
```

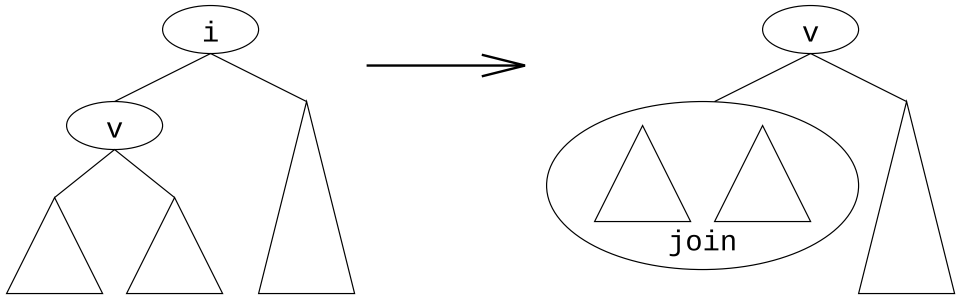
`balInorder` `take´ndrop`-pal való definiálását gyakorló feladatként az olvasóra bízzuk.

```
(* balPostorder xs = az xs lista elemeiből álló, postorder bejárású,
                 kiegyensúlyozott fa
    balPostorder: ´a list -> ´a tree
*)
fun balPostorder xs = balPreorder(rev xs)
```

12.4. Elem törlése bináris fából

Bináris fában adott értékű *elemet* rekurzív módszerrel *megkeresni* egyszerű feladat. *Új elemet beszúrni* sem nehéz: rekurzív módszerrel keresünk egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezve van, ügyelnünk kell arra, hogy a rendezettség megmaradjon.

Bináris fából adott értékű *elemet* vagy *elemeket* rekurzív módszerrel *kitörölni* valamivel nehezebb: ha a törlendő érték az éppen vizsgált részfa gyökerében van, a két részre széteső fa részfáit valamilyen módon *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



A vázolt műveletre mutat példát az alábbi `remove` függvény: rendezetlen bináris fából törli az `i` értékű értékű elem összes előfordulását. A `join` segédfüggvénnyel egyesítjük a törlés hatására létrejövő két részfát, mégpedig úgy, hogy a bal fát lebontjuk, és közben az elemeit berakjuk a jobb fába.

```
(* join(b, j) = a b és a j fák egyesítésével létrehozott fa
   join : ´a tree * ´a tree -> ´a tree
*)
fun join (L, tr) = tr
  | join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)

(* remove(i, f) = i összes előfordulását törli f-ből
   remove : ´a * ´a tree -> ´a tree
*)
fun remove (i, L) = L
  | remove (i, N(v,lt,rt)) = if i<>v
                             then N(v, remove(i,lt), remove(i,rt))
                             else join(remove(i,lt), remove(i,rt))
```

Megtehetjük azt is, hogy előbb egyesítjük a két részfát, majd az eredményül kapott fából töröljük az adott értékű elemet. Ezt a feladatot gyakorlásként az olvasóra bízzuk.

12.5. Bináris keresőfák

Ebben a szakaszban *bináris keresőfákon* alkalmazható műveleteket definiálunk. Rendszerint adott kulcsú elemet keresünk, ehhez értékeket kell összehasonlítanunk egymással; a keresett kulcsnak tehát *egyenlőségi típusúnak* kell lennie. A példákban a `string` típust használjuk, de a típus természetesen tetszőleges más egyenlőségi típus is lehet. Szebb lenne, ha *generikus függvényeket* írnánk; ezt a feladatot gyakorlásképpen meghagyjuk az olvasónak. A függvények *kivételes helyzetet* jeleznek, ha a keresett kulcsú elem nincs a keresőfában.

```
exception Bsearch of string
```

A `blookup` függvény adott kulcshoz tartozó értéket ad vissza egy rendezett bináris fából:

```
(* blookup(f, b) = az f fában a b kulcshoz tartozó érték
   blookup : (string * ´a) tree * string -> ´a
*)
fun blookup (N((a,x), t1, t2), b) =
      if b < a      then blookup(t1,b)
      else if a < b then blookup(t2, b)
                    else x
  | blookup (L, b) = raise Bsearch("LOOKUP: " ^ b)
```

A `binsert` függvény egy új kulcsú elemet rak be egy rendezett bináris fába, ha még nincs benne:

```
(* binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa
    binsert : (string * ´a) tree * (string * ´a) -> (string * ´a) tree
*)
fun binsert (L, (b,y)) = N((b,y), L, L)
  | binsert (N((a, x), t1, t2), (b,y)) =
      if b < a      then N((a, x), binsert(t1, (b,y)), t2)
      else if a < b then N((a, x), t1, binsert(t2, (b,y)))
                    else (* a=b *) raise Bsearch("INSERT: " ^ b)
```

A `bupdate` függvény meglévő kulcsú elembe új értéket ír be egy rendezett bináris fában:

```
(* bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén
                       az y értékkel
   bupdate : (string * ´a) tree * (string * ´a) -> (string * ´a) tree
*)
fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b)
  | bupdate (N((a,x), t1, t2), (b,y)) =
      if b < a      then N((a,x), bupdate(t1, (b,y)), t2)
      else if a < b then N((a,x), t1, bupdate(t2, (b,y)))
                    else (* a=b *) N((b,y), t1, t2)
```

13. fejezet

Listák használata: rendezés

Ebben a fejezetben rendezőalgoritmusokat mutatunk be SML-ben: a beszúró rendezést (`inssort`), a gyorsrendezést (`quicksort`), a fölülről lefelé haladó és az alulról fölfelé haladó összefésülő rendezést (`tmsort` és `bmsort`), valamint a simarendezést (`smsort`).

13.1. Beszúró rendezés

Az `ins` segédfüggvény az `x` elemet a megfelelő helyre rakja be az `ys` listában:

```
(* ins (x, ys) = az x értékkel a <= reláció szerint bővített ys
   ins : real * real list -> real list
   PRE: ys a <= reláció szerint rendezve van
*)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
  | ins (x : real, []) = [x]
```

`inssort`-tal rekurzívan rendezzük a lista maradékát; végrehajtási ideje $O(n^2)$:

```
(* inssort xs = az xs elemeinek a <= reláció szerint rendezett listája
   inssort : real list -> real list
*)
fun inssort (x::xs) = ins(x, inssort xs)
  | inssort [] = []
```

13.1.1. Generikus megoldások

Ha a következő elemet a helyére rakó `f` függvényt paraméterként adjuk át, az `inssort` tetszőleges típusú adatok rendezésére használható:

```
(* inssort f xs = az xs elemeinek az f segítségével rendezett listája
   inssort : (´a * ´b list -> ´b list) -> ´a list -> ´b list
*)
fun inssort f (x::xs) = f(x, inssort f xs)
  | inssort _ [] = []
```

Még jobb, ha magát az `ins` függvényt tesszük generikussá:

```
(* ins cmp (x, ys) = az x értékkel a cmp reláció szerint bővített ys
   ins : (´a * ´a -> bool) -> ´a * ´a list -> ´a list
   PRE: ys a cmp reláció szerint rendezve van
```

```
*)
fun ins cmp (x, ys) =
      let infix cmp
          fun ins0 (y::ys) = if x cmp y then x::y::ys else y::ins0 ys
            | ins0 [] = [x]
      in
         ins0 ys
      end
```

Ezzel `inssort` egy újabb változata:

```
(* inssort xs = az xs elemeinek a cmp reláció szerint rendezett listája
   inssort : (´a * ´a -> bool) -> ´a list -> ´a list
*)
fun inssort cmp (x::xs) = ins cmp (x, inssort cmp xs)
  | inssort _ [] = []
```

`inssort` eddig bemutatott változatai előbb elemeire szedik szét a rendezendő listát, majd hátulról visszafelé haladva, rendezés közben építik fel az újat. Jobbrekurziót és gyűjtőargumentumot használó változatának kisebb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét a 13.1.3 szakaszban hasonlítjuk össze).

```
fun inssort2 cmp xs =
             (* sort xs zs = az xs már feldolgozott elemeinek a cmp
                             reláció szerint rendezett listája zs
                sort : ´a list -> ´a list -> ´a list
             *)
      let fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
            | sort [] zs = zs
      in sort xs []
      end
```

13.1.2. Beszúró rendezés `foldr`-rel és `foldl`-lel

A második argumentumát gyűjtőargumentumként használó `foldl` sokkal kisebb vermet használ, mint `foldr`, ezért `inssort2` hosszabb listák rendezésére alkalmas. A futási időkről a 13.1.3 szakaszban lesz szó.

```
fun inssort cmp  = foldr (ins cmp) []
fun inssort2 cmp = foldl (ins cmp) []
```

13.1.3. A futási idők összehasonlítása

Véletlenszerűen előállított listák, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk. Véletlen eloszlású listát állít elő a `Random` könyvtárbeli `rangelist` függvény. Növekvő sorrendű egészlistát állít elő a `--` operátor:

```
infix --;
fun fm -- to =
      let fun upto to zs = if to < fm then zs else upto (to-1) (to::zs)
      in upto to []
      end;
```

A futási idők méréséhez 2000 elemet tartalmazó listákat állítunk elő:

```
val xs = Random.rangelist (1, 100000) (2000, Random.newgen());
```

illetve

```
val xs = 1 -- 2000;
```

A futási időket az alábbi programmal mérjük meg és íratjuk ki:

```
app load ["Int", "Random", "Time", "Timer"];
let val starttime = Timer.startCPUTimer()
    val zs = inssort op>= xs
    val {usr=tim,...} = Timer.checkCPUTimer starttime
    val t1 = "Integer sort with inssort: length = " ^
             Int.toString(length xs) ^ ", time = " ^
             Time.fmt 2 tim ^ " sec\n";
    val starttime = Timer.startCPUTimer()
    val zs = inssort2 op>= xs
    val {usr=tim,...} = Timer.checkCPUTimer starttime
    val t2 = "Integer sort with inssort2: length = " ^
             Int.toString(length xs) ^ ", time = " ^
             Time.fmt 2 tim ^ " sec\n"
in
    print(t1 ^ t2)
end;
```

A 2000 elemet tartalmazó, eredetileg fordított sorrendű listák rendezése `inssort` fenti változataival több mint 5 s-ig, a gyűjtőargumentumos `inssort2` bemutatott változataival viszont csak 0.01-0.02 s-ig tart (linux alatt, 233 MHz-es Pentium processzorral). Hatalmas a nyereség, nem beszélve arról, hogy gyűjtőargumentum nélkül a verem hamar megtelik! De eltűnik a különbség a kétféle változat között, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk: a futási idő bármelyik változat esetén kb. 2 s.

13.2. Gyorsrendezés

A gyorsrendezés a növekvő sorrendben rendezendő sorozatot három részre osztja: az m mediánra, a mediánnál nem nagyobb, ill. nagyobb elemekre:

$\leq$	m	$<$

Mediánnak az adott lépésben rendezendő lista fejét választjuk.

```
(* quicksort cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort : (´a * ´a -> bool) -> ´a list -> ´a list
*)
fun quicksort cmp xs =
      let (* qs : ´a list -> ´a list
          *)
          fun qs (m::ys) = partition ([], [], m, ys)
            | qs ys = ys
          (* partition : ´a list * ´ list * ´a * ´a list -> ´a list
          *)
          and partition (ls, rs, m, x::xs) =
                if cmp(x, m)
                then partition(x::ls, rs, m, xs)
                else partition(ls, x::rs, m, xs)
```

```
              | partition (ls, rs, m, []) = qs ls @ (m::qs rs)
        in
            qs xs
        end
```

`partition` iteratív függvény, amely két eredménylistát épít fel, `ls`-t és `rs`-t.

`quicksort` a 2000 elemet tartalmazó, eredetileg fordított sorrendű listát majdnem 14 s alatt rendezi. Az ugyanilyen hosszú, de véletlenszerűen előállított lista rendezéséhez ugyanakkor csak 0.1 s-ra van szüksége.
A `@` művelet kiküszöbölhető újabb segédargumentum bevezetésével. Az algoritmus hatékonysága tovább javítható pl. úgy, hogy az m értékű elemeket egy harmadik listában gyűjtjük, amikor a listát két részre bontjuk.

13.3. Összefésülő rendezés

Az összefésülő rendezéshez szükségünk van egy olyan függvényre, amely két listát növekvő sorrendben egyesít:

```
(* merge(xs, ys) = xs és ys elemeinek <= szerint egyesített listája
   merge : int list * int list -> int list
*)
fun merge (x::xs, y::ys) = if x <= y
                           then x::merge(xs, y::ys)
                           else y::merge(x::xs, ys)
  | merge ([], ys) = ys
  | merge (xs, []) = xs
```

vagy ún. *réteges minta* (...`as`...) alkalmazásával:

```
fun merge (xxs as x::xs, yys as y::ys)= if x <= y
                                        then x::merge(xs, yys)
                                        else y::merge(xxs, ys)
...
```

Hatékonyságromlást okoz, hogy a részeredményeket itt is a veremben tároljuk. Sajnos, ezen nem lehet segíteni, mert iteratív megoldás esetén vissza kellene fordítani az eredményül kapott listát.

13.3.1. Fölülről lefelé haladó összefésülő rendezés

A fölülről lefelé haladó (*top-down*) összefésülő rendezés akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük. (A `tmsort` név elején a `t` betű utal a *top-down* rendezésre.)

```
(* tmsort xs = az xs elemeinek a <= reláció szerint rendezett listája
   tmsort : int list -> int list
*)
fun tmsort xs = let val h = length xs
                    val k = h div 2
                in
                   if h > 1
                   then merge(tmsort(take(xs, k)), tmsort(drop(xs, k)))
                   else xs
                end
```

A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség. A módszer egyszerű és elég gyors.

13.3.2. Alulról fölfelé haladó összefésülő rendezés

Az alulról fölfelé haladó (*bottom-up*) összefésülő rendezés legegyszerűbb változata az eredeti l hosszúságú listát l darab egyelemű listára bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő. A megoldás egyszerű, de pazarló.

R. O´Keefe algoritmusa (1982) lépésről lépésre futtatja össze az egyforma hosszú részlistákat, de csak az utolsó lépésben rendezi az összeset. Az alábbi példában az összefuttatott részlistákat aláhúzással jelöljük:

$\underline{\texttt{A B}}$ `C D E F G H I J K`
$\underline{\texttt{A B}}$ $\underline{\texttt{C D}}$ `E F G H I J K`
$\underline{\texttt{A B C D}}$ `E F G H I J K`
$\underline{\texttt{A B C D}}$ $\underline{\texttt{E F}}$ `G H I J K`
$\underline{\texttt{A B C D}}$ $\underline{\texttt{E F}}$ $\underline{\texttt{G H}}$ `I J K`
$\underline{\texttt{A B C D}}$ $\underline{\texttt{E F G H}}$ `I J K`
$\underline{\texttt{A B C D E F G H}}$ `I J K`
$\underline{\texttt{A B C D E F G H}}$ $\underline{\texttt{I J}}$ `K`

Vagyis az algoritmus először az `A`-t futtatja össze a `B`-vel, majd a `C`-t a `D`-vel, ezt követően pedig az `AB`-t a `CD`-vel, hiszen most már ezek hossza is egyforma. Ezután `E` és `F`, `G` és `H`, `EF` és `GH`, majd `ABCD` és `EFGH` összefuttatása következik s.í.t.

Most `bmsort` néven definiáljuk az összefésülő rendezés alulról fölfelé haladó változatát, amely a `quicksort`-tal kb. azonos idő alatt rendez (a név első betűje, `b` utal a *bottom-up* rendezésre):

```
(* bmsort xs = az xs elemeinek a <= reláció szerint rendezett listája
   bmsort : int list -> int list
*)
fun bmsort xs = sorting(xs, [], 0)
```

`bmsort` a `sorting` segédfüggvényt használja. Ennek első argumentuma a rendezendő lista, második argumentumában a már rendezett részlistákat gyűjtjük (kezdőértéke `[]`) , harmadik argumentuma pedig az adott lépésben összefuttatandó elem sorszáma (kezdetben 0).

Ha a rendezendő lista még nem fogyott el, soron következő eleméből `sorting` egyelemű listát (`[x]`) képez, és ezt a már rendezett részlisták listája (`lss`) elé fűzve meghívja a `mergepairs` segédfüggvényt. `mergepairs`, amint látni fogjuk, az argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek. `k` az éppen átadott elem sorszáma. Ha a rendezendő lista kiürült, `sorting` a kétszintű lista egyetlen elemét, a rendezett listát adja eredményül.

```
(* sorting(xs, lss, k) = a még rendezetlen xs lista elemeit berakja
                         a k elemet tartalmazó, már rendezett lss listába
   sorting : int list * int list list * int -> int list
   PRE: k >= 0
*)
fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```

`mergepairs` egyetlen listában gyűjti a már összefuttatott részlistákat. Nem a listák hosszát hasonlítja össze, hanem az éppen átadott elem `k` sorszámából dönti el, hogy mit kell csinálni a következő részlistával.

```
(* mergepairs(llss, n)= az n elemet tartalmazó, már rendezett lss lista
                        első két részlistáját, ha egyforma a hosszuk,
                        összefuttatja
   mergepairs : int list list -> int list list
   PRE: n >= 0
```

```
*)
fun mergepairs (llss as ls1::ls2::lss, n) =
      (* legalább kételemű a lista *)
      if n mod 2 = 1
      then llss
      else mergepairs(merge(ls1, ls2)::lss, n div 2)
  | mergepairs (ls, _) = ls (* egyelemű a lista *)
```

Ha `n` páratlan, `mergepairs` a listát változtatás nélkül adja vissza, ha pedig páros, akkor az `llss` lista elején álló két, egyforma hosszú listát egyetlen rendezett listává futtatja össze. `n=0`-ra `mergepairs` az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.

A függvények működését az alábbi példán követhetjük. A kezdőhívás legyen

```
bmsort [1,2,3,4,5,6,7,8,9] =
         sorting ([1,2,3,4,5,6,7,8,9], [], 0)
```

Amíg `sorting` első argumentuma a nem üres `x::xs` lista, `sorting` saját magát hívja meg. A rekurzív hívás első argumentuma a lépésenként egyre rövidülő `xs` lista, harmadik argumentuma `(k+1)` a már feldolgozott listaelemek száma, második argumentuma pedig a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss` = `[]`.

A következő táblázatos elrendezés `mergepairs` mindkét argumentumát, valamint a rekurzív `sorting` hívás itt `j`-vel jelölt harmadik argumentumát, `k+1`-et és bináris számként `k`-t mutatja lépésről lépésre. (Ne feledjük, hogy `mergepairs`-nek listák listája az első argumentuma.) A `sorting` függvény hívja `mergepairs`-t azokban a sorokban, amelyekben a `j` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív. A táblázat utolsó oszlopa a vonatkozó magyarázatra hivatkozik.

Vegyük észre, hogy kapcsolat van az `llss` első eleme utáni listaelemek hossza és a `k` bitjei között! Ha `k` valamelyik bitje 1, akkor (balról jobbra haladva) az `llss` megfelelő listaelemének a hossza az adott bit helyiértékével egyenlő. A 0 értékű biteknek megfelelő listaelemek „hiányoznak” `llss`-ből.

```
llss                          n j    k
[[1]]                         1 1    0  m1
[[2],[1]]                     2 2    1  m2
[[1,2]]                       1         m3
[[3],[1,2]]                   3 3   10  m3
[[4],[3],[1,2]]               4 4   11  m2
[[3,4],[1,2]]                 2         m2
[[1,2,3,4]]                   1         m3
[[5],[1,2,3,4]]               5 5  100  m3
[[6],[5],[1,2,3,4]]           6 6  101  m2
[[5,6],[1,2,3,4]]             3         m3
[[7],[5,6],[1,2,3,4]]         7 7  110  m3
[[8],[7],[5,6],[1,2,3,4]]     8 8  111  m2
[[7,8],[5,6],[1,2,3,4]]       4         m2
[[5,6,7,8],[1,2,3,4]]         2         m2
[[1,2,3,4,5,6,7,8]]           1         m3
[[9],[1,2,3,4,5,6,7,8]]       9 9 1000  m3
[[9],[1,2,3,4,5,6,7,8]]       0 0       m4
[[1,2,3,4,5,6,7,8,9]]
```

Megjegyzések.

m1: Az argumentumként átadott listának egyetlen eleme van (ez maga is egy lista), ezért az argumentumot `mergepairs` második klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.

m2: `n` páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme egyforma hosszú lista, amelyeket `merge` egyetlen rendezett listává futtat össze, majd az eredménnyel `mergepairs` első klóza meghívja saját magát.

m3: `n` páratlan, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú lista, ezért az argumentumot `mergepairs` első klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.

m4: n=0, az összes listák listáját olyan listává kell összefuttatni, amelynek egyetlen lista az eleme.

A 2000 elemből álló listákat `bmsort` kevesebb mint 0.1 s alatt rendezi mind eredetileg fordított sorrendű, mint véletlenszerűen előállított listák esetén.

13.4. Simarendezés

A simarendezés (*smooth sort*) végrehajtási ideje $O(n)$, azaz arányos az elemek számával, ha a bemeneti lista csaknem rendezve van, és a legrosszabb esetben is legfeljebb $O(n \cdot log\, n)$.

Az applikatív simarendezés algoritmusa O´Keefe alulról fölfelé haladó rendezéséhez hasonló, de nem egyelemű listákat, hanem növekvő futamokat (*run*) állít elő.

Ha a futamok száma n-től független, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje $O(n)$.

```
(* nextrun(run, xs) = ???
   nextrun : int list * int list -> int list * int list
*)
fun nextrun (run, x::xs) =
      if x < hd run
      then (rev run, x::xs)
      else nextrun(x::run, xs)
  | nextrun (run, []) = (rev run, [])
```

`nextrun` eredménye egy pár. A pár első tagja a futam (egy növekvő számsorozat), a második tagja pedig a rendezendő lista maradéka. Mivel a futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani.

`smsorting` a futamokat ismételten előállítja és összefuttatja:

```
(* smsorting (xs, lss, k) = ???
   smsorting : int list * int list list * int -> int list
*)
fun smsorting (x::xs, lss, k) =
        let val (run, tail) = nextrun([x], xs)
        in
           smsorting(tail, mergepairs(run::lss, k+1), k+1)
        end
  | smsorting ([], lss, k) = hd(mergepairs(lss, 0))

(* smsort xs = az xs elemeinek a <= reláció szerint rendezett listája
   smsort : int list -> int list
```

```
*)
fun smsort xs = smsorting(xs, [], 0);
```

A simarendezés egy változata `sort` néven megtalálható a `Listsort` könyvtárban. A függvény specifikációja a következő:

```
sort ordr xs = xs elemei ordr szerint nem csökkenő sorrendben
               (Richard O´Keefe applikatív simarendezése)
val sort : (´a * ´a -> order) -> ´a list -> ´a list
```

Az `ordr` olyan függvény, amelynek egy pár az argumentuma, és e pár két elemének az összehasonlításával kapott `order` típusú érték az eredménye:

```
datatype order = LESS | EQUAL | GREATER
```

Az `order` típust a `General` könyvtár deklarálja. `compare` néven többek között a `Char`, az `Int`, a `Real`, a `String`, a `Word` és a `Word8` könyvtárban található olyan függvény, amely `sort` első argumentumaként használható.

A 2000 elemből álló listákat `Listsort.sort Int.compare` kevesebb mint 0.1 s alatt rendezi mind eredetileg fordított sorrendű, mint véletlenszerűen előállított listák esetén.

14. fejezet

Rekurzív függvények

Rekurzív megoldás esetén a megoldandó feladatot olyan részfeladatokra bontjuk, amelyek közül egyet vagy többet (de nem mindet!) az eredeti feladathoz hasonló módon oldunk meg.

Egyszerű rekurzióra már láttunk példákat, és láttuk azt is, hogyan lehet iteratívvá tenni a rekurziót. Most további, összetettebb példákat mutatunk be.

14.1. Egész kitevőjű hatványozás

Az SML könyvtári függvényei között van hatványozás (`Math.pow : real * real -> real`), a belső függvények között nincs. Most olyan, a `Math.pow`-nál egyszerűbb függvényt definiálunk, amely egész kitevőjű hatványozásra használható. Az alábbi `real * int -> real` típusú, infix pozíciójú, 8-as (a szorzásénál és az osztásénál magasabb) precedenciaszintű, *jobbra kötő* `**` függvény `k`<0-ra nincs értelmezve.

```
(* x ** k = x k-adik hatványa, k>=0
   ** : real * int -> real
*)
infixr 8 **;
fun _ ** 0 = 1.0
  | x ** k = x * x ** (k-1)
```

Az aláhúzás (`_`) a már jól ismert *mindenesjel.* A fenti definíció, mint tudjuk, azonos a következővel:

```
fun x ** k = if k = 0 then 1.0 else x * x ** (k-1)
```

A mintaillesztés definíciószerűen azonos a megfelelő `if-then-else` szerkezettel, ugyanakkor átláthatóbb, világosabban mutatja az esetek szétválasztását – csak éppen nem mindig alkalmazható. Ha például a `**` operátort negatív `k`-kra is definiálni akarjuk, nem használhatjuk a mintaillesztést, csak az `if-then-else` szerkezetet.

A bemutatott megoldás nem elég hatékony. Hogyan javíthatunk a hatékonyságán? Felhasználhatjuk pl. az

$x^1 = x$

$x^{2k} = (x^2)^k,\ k > 0$

$x^{2k+1} = x \cdot x^{2k},\ k > 0$

azonosságokat. Nézzünk egy példát:

$2^{10} = 4^5 = 4 * 4^4 = 4 * 16^2 = 4 * 256 = 1024$

```
(* pwr(x, k) = x k-adik hatványa, k>=0
   pwr : real * int -> real
*)
```

```
fun pwr (x, k) =
      if k = 0 then 1.0
      else if k = 1 then x
      else if k mod 2 = 0 then pwr(x*x, k div 2)
      else x * pwr(x*x, k div 2)
```

A `pwr` függvényben a második `else if` utáni `then` ág iteratív, az `else` ágat csak segédváltozóval lehetne iteratívvá tenni. (Miért?) Lehetne javítani az algoritmus *robosztusságán* és *hatékonyságán* is:

1. `k<0` esetén *kivételkezeléssel*,
2. a `k=0` vizsgálat felesleges ismétlődését kiküszöbölő *lokális deklarációval*,
3. a `pwr(x*x, k div 2)` függvényalkalmazás kétszeri kiszámítását kiküszöbölő *lokális kifejezéssel*.

A `pwr` függvény definíciója pl. így módosul, ha lokális kifejezést használunk (részlet):

```
...  else
      let val pwr0 = pwr(x*x, k div 2)
      in if k mod 2 = 0 then pwr0 else x * pwr0
      end;
```

14.2. Fibonacci-számok

A Fibonacci-számok jól ismert definíciója:
$F_0 = 0$,
$F_1 = 1$,
$F_n = F_{n-2} + F_{n-1},\ n > 1$.
Ha ezt így, ahogy van, átírjuk SML-re, használhatatlan programot kapunk: pl. F_{35}-öt egy 133 MHz-es Pentium processzoros számítógépen, linux alatt csaknem 35 s alatt számolja ki az *mosml* és csaknem 12 s alatt az *smlnj!* Keressünk jobb megoldást!

Írjunk `nextfib` néven olyan függvényt, amely egy Fibonacci-számpárból előállítja a következő Fibonacci-számpárt:

```
(* netxfib(p,c) = a (p,c) Fibonacci-számpárt követő
                  Fibonacci-számpár
   nextfib : int * int -> int * int
*)
fun nextfib (prev, curr) = (curr, prev + curr)
```

A megoldás jó, hiszen az SML-ben a függvény eredménye is tetszőleges típusú érték lehet! `nextfib` felhasználásával:

```
(* fibpair n = az n-edik Fibonacci-számpár (n>0)
   fibpair : int -> int * int
*)
fun fibpair n =
      if n = 1 then (0, 1) else nextfib(fibpair(n-1))
```

A kiértékelése elég nehezen követhető, ugyanis `fibpair` a

```
nextfib(nextfib (··· (nextfib(0, 1) ···)))
```

hívássorozatot állítja elő. Ugyanakkor a végrehajtása nagyon gyors, például a `fibpair 44` hívás azonnal kiírja a `(433494437, 701408733)` számpárt, amelynek a második tagja az `Int.Maxint`-nál még éppen nem nagyobb Fibonacci-szám. A keresett Fibonnaci-számot a számpár második tagjának kiválasztásával kapjuk, például:

```
- #2(fibpair 44);
> val it = 701408733 : int
```

Szép, áttekinthető és gyors megoldást kapunk *iterációval*:

```
(* iterfib(n,p,c) = a (p,c) Fibonacci-számpárt követő
                    n-edik Fibonacci-szám (n>0)
   iterfib : int * int * int -> int
*)
fun iterfib (1, prev, curr) = curr
  | iterfib (n, prev, curr) = iterfib(n-1, curr, prev + curr)
```

Az `else` ágban a rekurzió terminális. `fib` `iterfib`-et hívja meg az `n`-edik Fibonacci-szám előállításához:

```
(* fib n = az n-edik Fibonacci-szám
   fib : int -> int
*)
fun fib 0 = 0
  | fib n = iterfib(n, 0, 1)
```

Nézzünk egy példát `fib` redukciójára:

```
fib 7→iterfib(7,0,1)→iterfib(6,1,1)→iterfib(5,1,2)
   →···→iterfib(1,8,13)→13
```

Az `iterfib` függvényt célszerű lokálissá tenni `fib`-ben:

```
(* fib n = az n-edik Fibonacci-szám
   fib : int -> int
*)
local
   (* iterfib(n,p,c) = a (p,c) Fibonacci-számpárt követő
                       n-edik Fibonacci-szám (n>0)
      iterfib : int * int * int -> int
   *)
   fun iterfib (n, prev, curr) =
         if n = 1 then curr else iterfib(n-1, curr, prev+curr)
in
   fun fib 0 = 0
     | fib n = iterfib(n, 0, 1)
end
```

Ebben a példában a lokális deklaráció helyett lokális kifejezést is használhatnánk, de ez nem mindig van így .

14.3. Egész négyzetgyök közelítéssel

Az SML-ben négyzetgyökvonásra a `Math` könyvtárbeli `Math.sqrt` függvény használható. Most egy egész szám egész négyzetgyökének közelítésére írunk SML-függvényt. Az n szám k egész négyzetgyöke kielégíti az alábbi egyenlőtlenséget:

$k^2 \leq n < (k+1)^2$

Hogyan számíthatjuk ki a k-t rekurzióval? Meg kell találnunk a megfelelő részfeladatot, amely az eredetihez *hasonló*. A *lineáris* mellett a *felezéses* a másik sokszor alkalmazható alapmódszer, próbálkozzunk most az utóbbival.

$\sqrt{4n} = 2\sqrt{n}$, tehát ha n-et 4-gyel osztjuk, egyszerűsítjük a feladatot. Nem biztos, hogy n osztható 4-gyel, ezért az $n = 4m + v$ egyenletet fogjuk használni, ahol $v = 0, 1, 2, 3$ lehet. Mivel $m < n$, m egész négyzetgyökét a megírandó függvény rekurzív alkalmazásával kereshetjük:

$i^2 \leq m < (i+1)^2$

m is, i is egész, tehát ha m-hez 1-et adunk, legfeljebb egyenlő lehet $(i+1)^2$-nel:

$m + 1 \leq (i+1)^2$

Vonjuk össze a fenti egyenlőtlenségeket, és szorozzuk be minden tagját 4-gyel (mivel $n = 4m + v$ és $v < 4$, ezért $n < 4m + 4 = 4(m+1)$):

$(2i)^2 \leq 4m \leq n < 4m + 4 \leq (2i+2)^2$

Egészekről lévén szó, n négyzetgyöke $2i$ vagy $2i + 1$ lehet. Ezért a programnak az i kiszámítása után még meg kell vizsgálnia, hogy

$(2i+1)^2 \leq n$

teljesül-e, mert ha igen, akkor az eredményül kapott értékhez még 1-et kell adnia. Erre szolgál az `increase` függvény:

```
(* increase(j, n) = j+1, ha n négyzetgyöke nem kisebb j+1-nél,
                    egyébként j
   increase : int * int -> int
*)
fun increase (j, n) =
      j + (if (j+1)*(j+1) <= n then 1 else 0)
```

A rekurzió akkor fejeződik be, amikor n 0-vá válik. Mivel az egész osztás ismételt végrehajtásával az osztandó előbb-utóbb mindenképpen 0-vá válik, a rekurzió biztosan befejeződik (azaz a megállási feltétel teljesül):

```
(* introot n = n egész négyzetgyöke
   introot : int -> int
*)
fun introot n =
      if n = 0
      then 0
      else increase(2*introot(n div 4), n)
```

A bemutatott algoritmus elég gyors, nagyon egyszerű és a helyessége is könnyen belátható. Tanulság: a hatékonyságromlás oka általában a választott algoritmusban, pontosabban annak szerkezetében, nem pedig rekurzív voltában keresendő.

14.4. Valós szám négyzetgyöke Newton-Raphson módszerrel

Ha a négyzetgyökének egy közelítése x, akkor $\frac{\frac{a}{x}+x}{2}$ a négyzetgyök egy jobb közelítése. x kezdeti értéke legyen 1. A közelítés akkor ér véget, ha $\left|\left(x - \frac{\frac{a}{x}+x}{2}\right)/x\right|$ egy előre meghatározott ε értéknél, az előírt *relatív pontosságnál* kisebbé válik. A függvényt negatív számmal nem hívjuk meg, de 0.0-ra még jó eredményt kell adnia. Egy megvalósítása SML-ben (az ε neve a programban `eps`):

```
(* findroot(a, x, eps) = a négyzetgyöke Newton-Raphson közelítéssel,
                         eps relatív pontossággal, ahol x az előző
                         közelítő érték és a > 0.0
   findroot : real * real * real -> real
*)
```

```
fun findroot (a, x, eps) =
      let val nextx = (a/x + x) / 2.0
      in if abs(x - nextx) < eps * x
         then nextx
         else findroot(a, nextx, eps)
      end
(* sqroot a = a négyzetgyöke Newton-Raphson közelítéssel
   sqroot : real -> real
*)
fun sqroot 0.0 = 0.0
  | sqroot a = findroot(a, 1.0, 1E~10)
```

A programhoz két megjegyzést fűzünk:

1. `a` és `eps` állandók, ezért paraméterként való átadásuk felesleges, rontja a hatékonyságot. A javított változatban legyen mindkettő *globális* `findroot` számára.
2. Jobb, ha `findroot` kívülről nem látszik. A javított változatban `sqroot`-on belül lokális eljárásként definiáljuk.

A javított változat:

```
(* sqroot a = a négyzetgyöke Newton-Raphson közelítéssel
   sqroot : real -> real
*)
fun sqroot 0.0 = 0.0
  | sqroot a =
      let val eps = 1E~10
          (* findroot x = a négyzetgyöke eps relatív pontossággal,
                          ahol x az előző közelítő érték és a > 0.0
             findroot : real -> real
          *)
          fun findroot x =
                let val nextx = (a/x + x) / 2.0
                in
                   if abs(x - nextx) < eps * x
                   then nextx
                   else findroot nextx
                end
      in
         findroot 1.0
      end
```

14.5. $\pi/4$ közelítő értéke kölcsönös rekurzióval

Végül lássunk egy példát olyan kölcsönösen rekurzív függvények használatára, amelyek $\pi/4$ értékét határozzák meg (nem igazán hatékony módon) az alábbi közelítő polinom alapján:
$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \cdots + 1/(4k+1) - 1/(4k+3) + \cdots$
`pos`-sal a sorozat pozitív, `neg`-gel pedig a negatív előjelű tagjait számíttatjuk ki. `d`-vel a soron következő tag nevezőjét jelöljük.

```
(* pos d = pi/4.0 közelítő értékének pozitív előjelű,
          d nevezőjű tagja (d = 1.0, 5.0, 9.0, ...)
```

```
   pos : real -> real
   neg d = pi/4.0 közelítő értékének negatív előjelű,
           d nevezőjű tagja (d = 3.0, 7.0, 11.0, ...)
   neg : real -> real
*)
fun pos d = neg(d - 2.0) + 1.0/d
and neg d = if d > 0.0 then pos(d - 2.0) - 1.0/d else 0.0
```

`pos` és `neg` felhasználásával számítja ki π értékét a `sum` függvény:

```
(* sum n = pi n-edik közelítő értéke (n>=0)
   sum : int -> real
*)
fun sum n =
      let val d = real(2*n+1)
      in
         4.0 * (if n mod 2 = 0 then pos d else neg d)
      end
```

Segédargumentum alkalmazásával a kölcsönösen rekurzív függvények sokszor egyetlen függvénnyel helyettesíthetők:

```
(* sum n = pi n-edik közelítő értéke (n>=0)
   sum : int -> real
*)
local
   (* pi4(d, s)= pi/4.0 d nevezőjű, s előjelű közelítő értéke
      pi4 : real * real -> real
   *)
   fun pi4 (d, s) =
         if d > 0.0 then pi4(d-2.0, ~s) + s/d else 0.0
in
   fun sum n =
         let val d = real(2*n+1)
             val s = if n mod 2 = 0 then 1.0 else ~1.0
         in
            4.0 * pi4(d, s)
         end
end
```

14.6. A következő permutált

Adott egészek egy sorozata. Permutáljuk úgy a sorozatot, hogy az eredmény *lexikografikusan eggyel nagyobb* legyen az eredetinél! Tekintsük például a következő sorozatokat:

$1234 \to 12\underline{43} \to 1\underline{324} \to 13\underline{42} \to 1\underline{423} \to 14\underline{32} \to \underline{2134} \to \ldots \to 43\underline{21}$

Látható, hogy a jobb szélső, kisebb helyiértékű elemek változnak gyakrabban (aláhúzással jelöljük a helyüket változtatott elemeket). 4321-gyel a permutálás véget ér, mivel nincs nála lexikografikusan nagyobb sorozat.

Lexikografikusan nagyobb sorozatot permutálással úgy állíthatunk elő, hogy a sorozat egy elemét felcseréljük egy tőle *jobbra álló,* nála *nagyobb* elemmel, pl.

$1243 \to 1\underline{42}3$.

Pontosan eggyel nagyobb sorozat – a következő permutált – úgy állítható elő, ha a cserében a legkisebb helyiértékű elemhez, azaz a sorozat jobb széléhez lehető legközelebb álló elem vesz részt. Ezzel az elemmel kell felcserélni a hozzá legközelebbi, tőle balra álló, nála kisebb elemet, pl.

1243→1342.

Ebben a példában a sorozat jobb szélső elemével, a 3-mal cseréltük fel a hozzá legközelebbi, tőle balra álló, nála kisebb elemet, a 2-t. Sajnos, az eredményül kapott 1342 sorozat nem jó, mert 1243 után 1324 jön lexikografikus sorrendben. De már látszik a megoldás: a lecserélt elemtől jobbra álló, monoton csökkenő részsorozatot meg kell fordítani, hiszen az így kapott monoton növekedő részsorozat lexikografikusan a lehető legkisebb, pl.

1243→1342→1324.

Mivel a lista olyan szerkezet, amelyet a fejétől kezdve lehet feldolgozni, a sorozatot úgy ábrázoljuk, hogy a legkisebb helyiértékű elem legyen a lista feje. A példában előforduló sorozatokat listaként így ábrázoljuk:

`[4,3,2,1]`→`[3,4,2,1]`→`[4,2,3,1]`→`[2,4,3,1]`→`[3,2,4,1]`→ ...

Fogalmazzuk meg a következő permutáltat előállító algoritmust!

1. Bontsuk három részre a listát:

   - a lehető leghosszabb monoton növekedő részsorozatra *(zs)* – ti. a lehető legkisebb olyan elemet keressük, amelynek a bal oldalán nála nagyobb elem található;
   - az ezt a sorozatot követő elemre *(y)* – ui. éppen ezt az elemet akarjuk másra cserélni;
   - a maradék listára *(ys)* – amely ebben a lépésben nem változik.

   Például

`[`	`2, 4`	`,`	`3`	`,`	`1`	`]`
	zs		*y*		*ys*	

1. Keressük meg *zs*-ben azt a legkisebb z_i-t, amely y-nál azért nagyobb (ui. *eggyel nagyobb* sorozatot keresünk), és állítsuk elő a $z_i{::}ys$ részlistát (az előző példa esetén `[4,1]`-et)!
2. Fordítsuk meg a $[z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_n]$ maradéklistát, amely így csökkenő sorrendű lesz, és szúrjuk bele y-t a megfelelő helyre (a példában a maradéklista `[2]`, és ha a 3-at beszúrjuk, `[3, 2]`-t kapunk)! Biztos, hogy lexikografikusan ennél kisebb részlista nincsen.
3. Állítsuk elő az eredménylistát a részlisták összefűzésével (a példában `[3, 2, 4, 1]`-et kapunk)!

Néhány megjegyzés a vázolt algoritmushoz:

a) Vegyük észre, hogy (3) egyik részfeladatát – ti. *zs* megfordítását – kényelmesen elvégezhetjük (1)-ben: *zs*-t könnyebb eleve megfordítva előállítani az eredeti listából; ez lesz az (1´) lépés. Így a módosított (3´) lépésben y-t csak be kell szúrni a megfelelő helyre.

   (1´) E lépés eredményeképpen tehát előáll

   - a lehető leghosszabb monoton csökkenő részsorozat *(zs)*,
   - a *zs*-t követő elem *(y)*,
   - a lista változatlan maradéka *(ys)*.

A `next` függvény első részét `zs`, `y` és `ys` előállítására írjuk meg. Feltételezzük, hogy `zs`-ben kezdetben a jelenlegi lista első eleme van, és addig rakunk bele újabb elemeket `(y::ys)`-ből, amíg `zs` *monoton csökkenő* marad.

```
(* next(zs, ys) = a következő permutált
   next : int list * int list -> int list
*)
fun next (zs, y::ys) =
      if hd zs <= y
      then next (y::zs, ys)
      else ...
```

`ys`-t most már változtatás nélkül használhatjuk fel a (3´) lépésben.

b) Most a (2), (3´) és (4) lépések megvalósítására írunk függvényt `swap` néven.

Adva van a monoton csökkenő $zs = z_n, \ldots, z_{i+1}, z_i, z_{i-1}, \ldots, z_1$ sorozat. Elő kell állítani az ugyancsak monoton csökkenő $z_n, \ldots, z_{i+1}, y, , z_{i-1}, \ldots, z_1$ sorozatot. y és ys e függvény számára globális állandók.

Ha előállt az új sorozat, akkor a következő permutált is megvan:

$$[z_n, \ldots, z_{i+1}, y, z_{i-1}, \ldots, z_1] \mathbin{@} [z_i] \mathbin{@} ys$$

Rekurzív függvényt írunk. `z::z´::zs`-nek kezdetben legalább két eleműnek kell lennie, hogy a sorozat monoton csökkenő legyen.

Ha már csak egyetlen elem marad a feldolgozandó listában, annak biztosan kisebbnek kell lennie `y`-nál, mert ha nem lenne kisebb, akkor `z>y>=z´` teljesült volna az előző lépésben; így megvan az eredmény.

Ha valamely lépésben `z>y>=z´` teljesül, a `zs` lista végét már nem kell megvizsgálni, ugyancsak megvan az eredmény.

Mindaddig, amíg `z´>y` $(i = n, \ldots, 2)$, a függvény saját magát hívja meg.

```
(* swap zs = a (2), (3´) és (4) lépések eredményeként előálló sorozat
   swap : int list -> int list
*)
fun swap [z] = y::z::ys
  | swap (z::z´::zs) =
      if z´ > y (* z>=z´>y, folytasd!  *)
      then z::swap(z´::zs)
      else (* z>y>=z´ *)
        (y::z´::zs) @ (z::ys);
```

Ezzel a feladat megoldása:

```
local
   fun next (zs, y::ys) =
         if hd zs <= y
         then next(y::zs, ys)
```

```
              else (* zs@[y]@ys, ahol zs monoton csökkenő sorozat *)
                let fun swap [z] = y::z::ys (* z>y *)
                      | swap (z::z´::zs)=
                             if z´ > y
                             then (* z>=z´>y *)
                                   z::swap(z´::zs)
                             else (* z>y>=z´ *)
                                   (y::z´::zs) @ (z::ys)
                in
                   swap(zs)
                end
   in
      (* nextperm zs = a következő permutált
         nextperm : int list -> int list
      *)
      fun nextperm (y::ys) = next([y], ys)
   end;
```

Mielőtt tovább olvasná a jegyzetet, próbáljon meg önállóan válaszolni az alábbi kérdésekre!

1. Megváltoztatható-e `swap`-ban az ágak sorrendje?

2. Mi van akkor, ha elérjük a legnagyobb permutált sorozatot, pl. előáll az `[1, 2, 3, 4]` lista?

3. Mi van akkor, ha kezdetben `length ys < 2` ?

4. Lefedtünk-e minden esetet a `swap` függvény definíciójában?

A válaszok:

(A) Igen, hiszen kölcsönösen kizárják egymást az egyes ágakat kiválasztó feltételek.

(B) Ilyenkor `zs` elemei monoton csökkennek, és már nincs következő elem (`y::ys` üres). Az eredménylistának monoton növekedő sorozatnak kell lennie, ezért `zs`-t meg kell fordítani. A `next` függvényt tehát az alábbiak szerint kell módosítani:

```
local fun next (zs, []) = rev zs
        | next (zs, y::ys) = if hd zs <= y ...
```

(C) A kezdetben üres sorozatok kezelésére a `nextperm` függvény definícióját át kell írni:

```
...in fun nextperm [] = []
        | nextperm (y::ys) = next([y], ys)
   end;
```

Kérdés: miért oldja meg ez a változtatás a `length ys < 2` eset kezelését?

(D) Nem, hiszen `[z]` csak pontosan egy elemű, `(z::z´::zs)` pedig legalább két elemű listákra illeszkedik. Az üres lista kezeléséről nem mondtunk semmit. Csak azért hagyhattuk el a `swap [] = []` változatot, mert a `swap` függvényt üres listával sohasem hívjuk meg.

A fenti javításokat is tartalmazó megoldás:

```
local
   fun next (zs, []) = rev zs
     | next (zs, y::ys) =
         if hd zs <= y
         then next(y::zs, ys)
         else (* zs@[y]@ys, ahol zs monoton csökken *)
           let fun swap [z] = y::z::ys(* z > y *)
                 | swap (z::z´::zs) =
                     if z´ > y
                     then (* z >= z´ > y *)
                           z::swap(z´::zs)
                     else (* z > y >= z´ *)
                          (y::z´::zs) @ (z::ys)
           in
              swap(zs)
           end
in
   fun nextperm [] = [] (* ha a sorozat eleve üres *)
     | nextperm (y::ys) = next([y], ys)
end;
```

15. fejezet

Lusta lista

Az SML alapvetően mohó kiértékelésű, de lehet vele *lusta listát* definiálni: a lusta lista *farka* függvény, ezáltal *késleltetjük* a kiértékelését. Ily módon *végtelen listákat* hozhatunk létre, amelyeknek előnyős tulajdonságaik mellett hátrányaik, veszélyeik is vannak, pl.

- egy lusta lista *bármely részét* megjeleníthetjük, de *sohasem az egészet;*
- két lusta lista elemeiből páronként képezhetünk egy harmadikat, de például *nem számíthatjuk ki* egy lusta lista *elemeinek összegét,* nem kereshetjük meg benne *a legkisebbet,* nem fordíthatjuk meg az *elemek sorrendjét*;
- úgy kell rekurziót definiálnunk, hogy *nincs alapeset;*
- egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény *tetszőleges véges része véges idő alatt* előáll.

Az SML-ben a lusta listákat kezelő függvények bonyolultabbak, mint egy lusta kiértékelést alkalmazó nyelvben, mert a lista farkában *explicite* hivatkoznunk kell a függvényre.

A lusta listát *sorozatnak* (sequence) fogjuk nevezni, és a `seq` típusoperátort használjuk a létrehozására.[1]

```
datatype ´a seq = Nil | Cons of ´a * (unit -> ´a seq)
```

Egy sorozat fejét, ill. farkát adják eredményül az alábbi `head`, ill. `tail` függvények ; mindkettő abortál, ha üres sorozatra alkalmazzák.

```
- fun head (Cons(x, _)) = x;
> val head = fn :´a seq -> ´a
```

A sorozat farka `unit -> ´a seq` típusú függvény, erre illesztjük az `xf` mintát; `tail` törzsében `xf`-et a `()` argumentumra kell alkalmazni:

```
- fun tail (Cons(_, xf)) = xf();
> val tail = fn : ´a seq -> ´a seq
```

`consq(x, xq)` `x`-et berakja az `xq` sorozatba:

```
- fun consq (x, xq) = Cons (x, fn () => xq);
> val consq = fn : ´a * ´a seq -> ´a seq
```

Ha a `consq` függvényt alkalmazzuk, mondjuk, az `(x,E)` argumentumra, a `consq(x,E)` kifejezést az SML *nem lustán* értékeli ki, hiszen az SML alapvetően mohó kiértékelésű. Ha `E` kiértékelésének

[1] Mivel az SML különbséget tesz a kis- és nagybetűk között, a hagyományos lista `nil` és a lusta lista `Nil` konstruktora nem azonosak. A `unit` típus, mint tudjuk, a típusműveletek *egységeleme;* egyetlen elemét `()`-lel jelöljük.

eredményét `xq`-val jelöljük, akkor `consq(x,E)` kiértékelése a fenti definíció szerint `Cons(x, fn () => xq)`-t eredményez. A `consq`-beli `fn () => xq` függvény *nem késlelteti* a farok (a példában `E`) kiértékelését `consq` alkalmazásakor. Az SML-ben a lusta kiértékelés érdekében a híváskor is a `Cons(x, fn () => E)` alakot kell használnunk, `consq(x,E)` nem jó. Az *explicit* `fn () => E` *késlelteti* a kiértékelést, és ezzel *szükség szerinti* hivatkozást valósít meg.

Példaként a korábban megismert `from` és `take` függvények lusta változatait mutatjuk be. A `fromq k` sorozat egészek `k`-tól induló végtelen sorozata:

```
- fun fromq k = Cons(k, fn () => fromq(k+1));
> val fromq = fn : int -> int seq
- head (fromq 1);
> val it = 1
- fromq 1;
> val it = Cons (1, fn): int seq;
- tail it;
> val it = Cons (2, fn): int seq;
- tail it;
> val it = Cons (3, fn): int seq;
```

`takeq(n, xq)` az `xq` sorozat első `n` eleméből képzett listát adja vissza:

```
- fun takeq (0, xq) = []
    | takeq (n, Cons (x, xf)) = x :: takeq(n-1, xf());
    | takeq (n, Nil) = []
> val takeq = fn : int * ´a seq -> ´a list
```

Nézzünk egy példát `takeq` egyszerűsítésére!

```
takeq(2, fromq 30) →
takeq(2, Cons(30, fn () => fromq(30+1))) →
30::takeq(1, fromq(30+1)) →
30::takeq(1, Cons(31, fn () => fromq(31+1))) →
30::(31::takeq(0, fromq(31+1))) →
30::31::takeq(0, Cons(32, fn () => fromq(32+1))) →
30::31::[] → [30, 31]
```

A 32-t az SML ugyan kiszámítja, de sohasem használja fel.

Az `´a seq` típus nem egészen lusta kiértékelésű: egy nemüres sorozat fejét a rendszer mindig feldolgozza, és ezt az SML-ben csak körülményesen lehet elkerülni.[2]

15.1. Elemi feldolgozási műveletek sorozatokkal

A kiszámíthatóság érdekében egy függvény eredményének tetszőleges, véges része az argumentum véges részétől függhet csak. Amikor az eredményre *szükség* van, akkor ez az igény *váltja ki* az argumentum feldolgozását.

Nézzünk egy példát, amelyben egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény farka – egy függvény – alkalmazza a `squareq` függvényt az argumentum farkára.

```
- fun squareq Nil: int seq = Nil
    | squareq (Cons (x, xf)) = Cons(x * x, fn () => squareq(xf()));
> val squareq = fn : int seq -> int seq
- squareq(fromq 1);
> val it = Cons(1, fn) : int seq
```

[2] Lásd L.C. Paulson: SML for the Working Programmer, Cambridge Press, 1991, 168. o.

```
- takeq(10, it);
> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] : int list
```

Két sorozat hasonlóan adható össze:

```
- fun addq (Cons (x, xf), Cons(y, yf)) =
                              Cons(x+y, fn () => addq(xf(), yf()))
    | addq _: int seq = Nil;
> val addq = fn : (int seq * int seq) -> int seq
- addq(fromq 1000, squareq(fromq 1));
> val it = Cons(1001, fn) : int seq
- takeq (5, it);
> val it = [1001, 1005, 1011, 1019, 1029] : int list
```

Az `appendq` függvény addig nem nyúl `yq`-hoz, amíg `xq` ki nem ürül – vagyis csak akkor nyúl hozzá, ha `xq` véges. Véges sorozatot `consq`-val készíthetünk. Most érthetjük meg, hogy miért kellett a típusdefinícióban a `Nil` konstruktorállandót definiálni.

```
- fun appendq (Cons (x, xf), yq) = Cons(x, fn () => appendq (xf(), yq))
    | appendq (Nil, yq) = yq;
> val appendq = fn : ´a seq * ´a seq -> ´a seq

- val finiteq = consq(25, consq(10, Nil));
> val it = Cons(25, fn) : int seq
- appendq(finiteq, fromq 1526);
> val it = Cons(25, fn) : int seq
- takeq (4, it);
> val it = [25, 10, 1526, 1527] : int list
```

15.2. Magasabb rendű függvények sorozatokra

Két magasabb rendű függvény – a `map` és a `filter` – lusta változatát definiáljuk ebben a szakaszban.

```
- fun mapq f (Cons (x, xf)) = Cons(f x, fn () => mapq f (xf()))
    | mapq f Nil = Nil;
> val mapq = fn : (´a -> ´b) -> ´a seq -> ´b seq

- fun filterq p (Cons (x, xf)) = if p x
                                 then Cons(x, fn () => filterq p (xf()))
                                 else filterq p (xf())
    | filterq p Nil = Nil;
> val filterq = fn : (´a -> bool) -> ´a seq -> ´a seq
```

Az előző szakaszban definiált `squareq` sokkal egyszerűbben definiálható `mapq`-val; `f`-ként egészek esetén `sq´i`-t, valósak esetén `sq´r`-et kell használni, pl.

```
- val squareq = mapq sq´i;
> val squareq = fn :  int seq -> int seq
```

Most olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egészek vannak:

```
- filterq (fn n => n mod 10 = 7) (fromq 50);
> val it = Cons(57, fn) : int seq
- takeq(8, it);
> val it = [57, 67, 77, 87, 97, 107, 117, 127] : int list
```

Az ugyancsak magasabb rendű `iterateq` függvény – a `fromq` egy általánosítása – a következő sorozatot állítja elő (v.ö. a korábban definiált `repeat`-tel):

$$[x, f(x), f(f(x)), \cdots, f^k(x), \cdots]$$

```
- fun iterateq f x = Cons(x, fn () => iterateq f (f x));
> val iterateq = fn : (´a -> ´a) -> ´a -> ´a seq
- iterateq (secr op/ 2.0) 1.0;
> val it = Cons (1.0, fn) : real seq
- takeq (5, it);
> val it = [1.0, 0.5, 0.25, 0.125, 0.625] : real list
```

`fromq`-t `iterateq`-val úgy kaphatjuk meg, hogy `f`-ként a `succ` függvényt alkalmazzuk:

```
- fun succ k = k + 1;
> val succ = fn :  int -> int
- val fromq = iterateq succ
> val fromq = fn :  int -> int seq
```

15.3. Néhány összetett példa

15.3.1. Álvéletlen-számok

A hagyományos álvéletlenszám-generátorok olyan eljárások, amelyek egy változóban tárolják a `seed` *(mag)* értéket – ebből állítják elő a következő hívásnál a következő álvéletlenszámot. Ha sorozatként valósítjuk meg az álvéletlenszámokat, akkor a következő álvéletlenszám csak *szükség esetén* áll elő.

```
- local val a = 16807.0 and m = 2147483647.0
        (* nextrandom : real -> real
        *)
        fun nextrandom seed =
              let val t = a * seed
              in t - real(floor(t/m)) * m
              end
  in
     fun randseq s = mapq (secr op/ m) (iterateq nextrandom (real s))
  end;
> val randseq = fn : int -> real seq
```

Ha `nextrandom`-ot 1.0 és 21474836467.0 közötti `seed`-értékre alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az `a * seed mod m` művelettel. A valós számokat csak a túlcsordulás elkerülésére használjuk.

A sorozat előállítására `iterateq`-t `nextrandom`-ra és `seed` valós számmá alakított kezdőértékére alkalmazzuk. `mapq` gondoskodik arról, hogy a sorozatban minden értéket elosszunk `m`-mel, és így `randseq` 0.0-nál nem kisebb és 1.0-nél kisebb értékeket adjon eredményül. Látható, hogy a sorozat a megvalósítás részleteit szépen elrejti a felhasználó elől.[3]

Az előállított álvéletlen-számok tehát 0.0-nál nem kisebb és 1.0-nél kisebb valós számok; `mapq`-val alakíthatjuk át őket 0 és 1 közötti egészekké:

```
- mapq (floor o secl 10.0 op*) (randseq 1);
> val it = Cons(0, fn): int seq
- takeq(5, it);
> val it = [0, 0, 1, 7, 4]: int list
```

[3]Ezt az algoritmust Park és Miller dolgozta ki 1988-ban. Helyes működéséhez a mantisszának 46 bitesnek (!) kell lennie. A MS-DOS alatti SML-megvalósítások mantisszája ennél rendszerint jóval rövidebb. Kevésbé jó statisztikai tulajdonságú álvéletlen-számokat kaphatunk, ha `a`-nak és `m`-nek kisebb relatív prímeket választunk.

15.3.2. Prímszámok

Következő programunk az *eratosztenészi szita* egy megvalósítása. Idézzük föl az ismert algoritmust:

1. Vegyük az egészek 2-vel kezdődő sorozatát: [2, 3, 4, 5, 6, 7, ...]
2. Töröljük az összes 2-vel osztható számot: [3, 5, 7, 9, 11, ...]
3. Töröljük az összes 3-mal osztható számot: [5, 7, 11, 13, 17, 19, ...]
4. Töröljük az összes 5-tel osztható számot: [7, 11, 13, 17, 19, ...]
5. Töröljük az összes ...

Látható, hogy a sorozat első eleme mindig a következő prím. A sorozatban azok a számok maradnak benne, amelyek az eddig előállított prímekkel nem oszthatók.

```
- fun sift p = filterq (fn n => n mod p <> 0);
> val sift = fn : int -> int seq -> int seq
```

A `sift` (szitál, rostál) segédfüggvény a `p` argumentum többszöröseit törli egy sorozatból. A `sieve` (szita, rosta) függvénynek már csak ismételten alkalmaznia kell `sift`-et a megfelelő sorozatra. Feltehetjük, hogy ez a sorozat sohasem üres, ezért *nem kell* az üres sorozatra illeszkedő változatot írnunk.

```
- fun sieve (Cons (p, nf)) = Cons(p, fn () => sieve(sift p (nf())))
    | sieve Nil = Nil;
> val sieve = fn : int seq -> int seq
- val primes = sieve (fromq 2);
> val primes = Cons(2, fn): int seq
- takeq(25, primes);
> val it = [2, 3, 5, 7, ..., 83, 89, 97]: int list
```

15.3.3. Numerikus számítások

A példa legyen a négyzetgyökvonás Newton-Raphson módszerrel.

```
- fun nextapprox a x = (a/x + x)/2.0;
> val nextapprox = fn : real -> real -> real
```

`nextapprox` x_k-ból x_{k+1}-et számítja ki az $x_{k+1}=\frac{\frac{a}{x_k}+x_k}{2}$ képlet alapján. A befejeződés megállapítására egyszerű tesztet írunk:

```
- fun within (eps: real) (Cons (x, xf)) =
        let val Cons (y, yf) = xf()
        in if abs (x-y) <= eps then y else within eps (Cons (y, yf))
        end;
> val within = fn : real -> real seq -> real
```

A `(Cons (y, yf))` és az `xf()` sorozat ugyanaz: az `else`-ágban azért használjuk mégis az előbbit, hogy elkerüljük `xf()` költségesebb meghívását.

```
- fun qroot a = within 1E~6 (iterateq (nextapprox a) 1.0);
> val qroot = fn : real -> real
- qroot 5.0;
> val it = 2.236067977 : real
- it * it;
> val it = 5.0 : real
```

A fenti példa világosan különválasztja a *leállásvizsgálatot* (termination test) *a következő jelölt előállításától.*

A példában az *abszolút különbséget* ($\mid x - y \mid < \varepsilon$) teszteljük, de vizsgálhatnánk pl. a *relatív különbséget* ($\mid x/y - 1 \mid < \varepsilon$) vagy az $(\mid x - y \mid)/((\mid x \mid + \mid y \mid)/2 + 1) < \varepsilon$ feltételt. A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást. Minden egyes leállásvizsgálat tehát egy-egy függvény legyen, olyan, amely egy sorozatból egy valós számot állít elő. Egy másik függvény állítja majd elő `real seq -> real seq` típusú függvényként a következő jelöltet. Ha később integrálni, differenciálni stb. akarunk, csak a megfelelő integráló, differenciáló stb. függvényeket kell megírni, a leállásvizsgálatot végző függvényeket felhasználhatjuk.

Most tehát a következő jelölt előállítására írunk függvényt, és ezzel elrejtjük a részleteket:

```
fun approxq a =
      let (* nextapprox : real -> real
          *)
      fun nextapprox x = (a/x + x) / 2.0
      in iterateq nextapprox 1.0
      end;
> val approxq = fn : real -> real seq
```

Most már könnyű megírni a `qroot` függvény egy tisztább változatát:

```
val qroot = within 1E~6 o approxq;
> val qroot = fn : real -> real
```

15.4. Sorozatok sorozata és egymásba ékelése

Legyen xq és yq egy-egy sorozat. Képezzünk új sorozatot az (x_i,y_j) párokból, ahol $x_i \in xq$ és $y_j \in yq$! A feladat megoldása során látni fogjuk, hogy a végtelen listák kezelése milyen különleges problémákat vet fel. Megkönnyíti a megoldás kidolgozását, ha először véges listákra oldjuk meg ugyanezt a feladatot `map` és `pair` alkalmazásával.

15.4.1. Keresztszorzatokból álló lista

Legyen tehát xs és ys egy-egy lista. Képezzünk listát az (x_i,y_j) párokból, ahol $x_i \in xs$ és $y_j \in ys$! `map`-et, `pair`-t és `flat`-et alkalmazva juthatunk el a keresett függvényhez. `pair` két értékből képez párt, `flat` listák listáját simítja listává. Idézzük föl a definíciójukat:

```
- fun pair x y = (x, y);
> val pair = fn : ´a -> ´b -> (´a * ´b)

- fun flat xss = foldl op@ [] xss;
> val flat = fn : ´a list list -> ´a list
```

Most már felírhatjuk a feladat megoldását véges listák esetére. A részleges `(pair x)` függvény a rögzített `x` értékből és az argumentumából képez párt. Ha ezt a függvényt `map` az `ys` lista elemeire alkalmazza:

```
map (pair x) ys
```

akkor olyan párokból álló listát kapunk eredményül, amelyben a párok első tagja a rögzített `x` érték, második tagja pedig az `ys` egy-egy eleme. Hogyan érhetjük el, hogy `x` végigfusson az `xs` lista összes elemén? Először is az eddig szabad `x`-et kössük le egy függvény argumentumaként:

```
fn x => map (pair x) ys
```

majd pedig alkalmazzuk újból `map`-et erre a függvényre és `xs`-re:

```
map (fn x => map (pair x) ys) xs
```

Igen ám, de most listák listáját kaptuk eredményül, hiszen a belső `map` már listát adott vissza, amelynek minden eleméből újabb listát képeztünk a külső `map`-pel. Nevezzük ezt a függvényt `pairss`-nek (a nevet záró két `s` utal arra, hogy az eredmény listák listája):

```
- fun pairss xs ys = map (fn x => map (pair x) ys) xs;
> val pairss = fn : ´a list -> ´b list -> (´a * ´b) list list
```

`flat` elvégzi a szükséges simítást, és ezzel a feladatot megoldó `pairs` függvény definíciója:

```
- fun pairs xs ys = flat (map (fn x => map (pair x) ys) xs);
> val pairs = fn : ´a list -> ´b list -> (´a * ´b) list
```

15.4.2. Keresztszorzatokból álló sorozat

Térjünk vissza az eredeti feladathoz, vagyis legyen xq és yq egy-egy sorozat! Képezzünk új sorozatot az (x_i,y_j) párokból, ahol $x_i \in xq$ és $y_j \in yq$!

A `pairss`-hez hasonlóan állíthatjuk elő párok sorozatának sorozatát (a nevet záró két `q` utal arra, hogy az eredmény *sorozatok sorozata):*

```
- fun pairqq xq yq = mapq (fn x => mapq (pair x) yq) xq;
> val pairqq = fn : ´a seq -> ´b seq -> (´a * ´b) seq seq
```

Az eredmény *véges része* kiíratható `takeqq`-val, amely a bal felső saroktól számított első `m` sorból és `n` oszlopból álló *téglalapot* jeleníti meg az `xqq` sorozatból:

```
- fun takeqq ((m, n), xqq) = map (secl n takeq) (takeq(m, xqq));
> val ´a takeqq = fn : (int * int) * ´a seq seq -> ´a list list
```

Állítsunk elő például párok sorozatából álló olyan sorozatot, amelyben a párok első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:[4]

```
- pairqq (fromq 30) primes;
> val it = Cons (Cons ((30, 2), fn), fn): (int * int) seq seq
- takeqq((3, 5), it);
> val it = [[(30, 2), ··· (30, 11)],
            [(31, 2), ··· (31, 11)],
            [(32, 2), ··· (32, 11)]] : (int * int) list list
```

Az előző szakaszban alkalmazott `flat` listákból álló lista elemeit fűzi egyetlen listába. Mi a helyzet végtelen sorozatok esetén? Ha `xq` végtelen, `appendq (xq, yq) = xq`, `flat`-hez hasonló függvénnyel tehát nem megyünk semmire! Ellenben két sorozat elemei *páronként egymásba ékelhetők* (interleave):

```
fun interleaveq (Nil, yq) = yq
  | interleaveq (Cons (x, xf), yq) =
                       Cons(x, fn () => interleaveq(yq, xf()));
> val interleaveq = fn : ´a seq * ´a seq -> ´a seq
```

Vegyük észre, hogy `interleaveq` *a rekurzív hívásban* váltogatja a két sorozatot, egyik sorozat sem zárja ki a másikat! Pl.

```
- takeq(10, interleaveq(fromq 0, fromq 50));
> val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
```

[4]Sajnos, az SML-értelmező nem az itt látható, jól olvasható alakra tördeli a kiírt szöveget, hanem csak a képernyőről kilógó sorokat vágja egy vagy több darabba.

Most `enumerate` néven olyan függvényt definiálunk, amely sorozatok sorozatából egyetlen sorozatot állít elő. Legyen a kétszeres mélységű sorozat feje `xq` és a farka `xqf`; alkalmazzuk `enumerate`-et rekurzívan `xqf`-re, majd az eredményt ékeljük `xq`-ba:

```
- fun enumerate Nil = Nil
    | enumerate (Cons (xq, xqf)) = interleaveq (xq, enumerate(xqf()));
> val enumerate = fn : ´a seq seq -> ´a seq
```

Csakhogy ez a „megoldás” nem jó, mert ha a sorozat, amelyre alkalmazzuk, végtelen, a rekurzió is végtelen lesz! Nem lenne az lusta nyelv esetén, az SML-ben azonban, amint van eredmény, a rekurzív hívást késleltetni kell, ezért több esetet kell megkülönböztetnünk:

```
fun enumerate Nil = Nil
  | enumerate (Cons (Nil, xqf)) = enumerate (xqf())
  | enumerate (Cons (Cons (x, xf), xqf)) =
                Cons(x, fn () => interleaveq(enumerate(xqf()),xf()));
> val ´a enumerate = fn : ´a seq seq -> ´a seq
```

Vagyis ha a bemeneti sorozat üres, készen vagyunk. Ha nem üres, meg kell vizsgálni a sorozat fejét: ha ez üres, akkor folytatni kell a rekurzív hívást, de ha nem üres, akkor az explicit `fn () =>` $\cdots$ függvénydefinícióval *késleltetni kell* a rekurziót.

Állítsuk elő például a pozitív egészekből álló párok egy sorozatát!

```
- val posintqq = pairqq (fromq 1) (fromq 1);
> val posintqq = Cons (Cons ((1, 1), fn), fn):(int * int) seq seq
- takeq(15, enumerate posintqq);
> val it = [(1,1), (2,1), (1,2), (3,1), (1,3), (2,2),
            (1,4), (4,1), (1,5), (2,3), (1,6), (3,2),
            (1,7), (2,4), (1,8)] : (int * int) list
```

A. Függelék

Az SML alapnyelv szintaxisa

Ez a szintaxisleírás a *Moscow ML Language Overview – V1.44* c. kézikönyv alapján készült. (Az előadásokon bemutatott szintaxis már a *V2.0* változat kézikönyvében szereplő jelöléseket követi: a leírt alapnyelv lényegében ugyanaz, de a jelölésekben kisebb eltérések vannak.)

A.1. Fogalmak és jelölések

A.1.1. Nevek

Egy név lehet

- alfanumerikus, azaz betűk, számjegyek, percjelek és aláhúzás-jelek olyan sorozata, amely betűvel vagy percjellel kezdődik;
- szimbolikus, azaz az alábbi jelek tetszőleges, nem üres sorozata:

```
! % & $ # + - / : < = > ? @ \ ~ ` ^ | *
```

Nem használhatók a következő, ún. fenntartott szavak vagy kulcsszók:

```
abstype and andalso as case do datatype else end
exception fn fun handle if in infix infixr let local
nonfix of op open orelse raise rec sig signature
struct structure then type val with withtype while
( ) [ ] { } , : :> ; ... _ | = => -> #
```

A neveket különféle osztályokba soroljuk:

var	értékváltozó	value variable	long
con	adatkonstruktor	value constructor	long
excon	kivételkonstruktor	exception constructor	long
tyvar	típusváltozó	type variable	
tycon	típuskonstruktor	type constructor	long
lab	mezőnév	record label	
unitid	modulnév	unit identifier	

- A típusváltozó olyan alfanumerikus név, amely percjellel kezdődik, pl. `´a`.
- A mezőnév tetszőleges név lehet, vagy olyan pozitív egész , amely nem 0-val kezdődik.

- Minden ´long´ jelzésű X osztálynak van egy *longX* párja. A *longX* osztályba tartozó nevek rövid és hosszú alakban (ún. *minősített névként)* is felírhatók. A rövid alak csak egy névből, a hosszú alak egy modulnévből, egy pontból és egy névből áll:

longx	::=	x	név	identifier
		unitid. x	minősített név	qualified identifier

A.1.2. Infix operátorok

Egy név az `infix` vagy az `infixr` direktívával *infix* helyzetűnek deklarálható. Ha az *id* név infix helyzetű, akkor az exp_1 *id* exp_2 kifejezés, szükség esetén zárójelek között, minden olyan helyen használható, ahol az *id(*exp_1*,* exp_2*)* vagy az *id{1=*exp_1*, 2=*exp_2*}* kifejezések egyébként használhatók. Infix helyzetű nevek mintában szintén használhatók.

Egy minősített nevet, vagy egy olyan nevet, amelyet az `op` szócska előz meg, sohasem lehet infix helyzetben alkalmazni. Az `infix`, `infixr` és `nonfix` direktívák szintaxisa a következő ($n \geq 1$):

`infix`	<`d`>	id_1 ... id_n	balra köt	left associative
`infixr`	<`d`>	id_1 ... id_n	jobbra köt	right associative
`nonfix`		id_1 ... id_n	nem köt	non-associative

A `d` decimális számjegy opcionális, és a nevek precedenciáját adja meg; alapértelmezés szerinti értéke 0. Nagyobb szám magasabb precedenciát jelent. Az `infix`, `infixr` és `nonfix` direktívák érvényességi tartománya a szokásos, azaz a `let` kifejezésen és a `local` deklaráción belül lokális érvényűek.

Azonos precedenciájú, de különféle balra kötő operátorok balra, azonos precedenciájú, de különféle jobbra kötő operátorok pedig jobbra kötnek. Tilos azonos precendeciájú, de különböző kötésű operátorokat használni ugyanabban a kifejezésben.

A.1.3. Jelölések

- Minden szintaktikai osztályt változatok listájaként adunk meg, mégpedig soronként egy változatot. Üres sor üres kifejezést jelent.

- A < és a > csúcsos zárójelpárok opcionális kifejezést fognak közre.

- Tetszőleges X szintaktikai osztályra (amelyre az x értelmezve van) az alábbiak szerint definiáljuk az *Xseq* szintaktikai osztályt (amelyre az *xseq* értelmezve van):

xseq	::=	x	egyelemű sorozat	singleton sequence
			üres sorozat	empty sequence
		x_1, ..., x_n	sorozat	sequence, $n \geq 1$

- A kifejezésváltozatokat precendenciájuk csökkenő sorrendjében adjuk meg.

- A típusok szintaxisa erősebben köt, mint a kifejezéseké.

- A megjegyzés oszlopban az L betű balra kötő műveletet jelöl.

- Minden ismétlődő konstrukció (pl. a *klózsorozat)* a lehető legtovább terjeszkedik jobbra. Ezért egy `case`-kifejezést egy másik `case-`, `fn-` vagy `fun`-kifejezésen belül esetleg zárójelbe kell rakni.

- A különféle típusú állandók (vö. *scon)* jelölését a 2 fejezetben ismertettük.

A.2. Az SML alapnyelv szintaxisa

A.2.1. Kifejezések és klózsorozatok

exp	::=	*infexp*		
		exp : *ty*	típusmegkötés (L)	type constraint (L)
		exp1 `andalso` *exp2*	feltételes konjunkció	sequential conjunction
		exp1 `orelse` *exp2*	feltételes alternáció	sequential disjunction
		exp `handle` *match*	kivétel kezelése	handle exception
		`raise` *exp*	kivétel jelzése	raise exception
		`if` *exp1* `then` *exp2* `else` *exp3*	feltételes kifejezés	conditional expression
		`while` *exp1* `do` *exp2*	iteráció	iteration
		`case` *exp* `of` *match*	esetszétválasztás	case analysis
		`fn` *match*	lambda-kifejezés	function expression
infexp	::=	*appexp*		
		infexp1 *id* *infexp2*	infix alkalmazás	infixed application
appexp	::=	*atexp*		
		appexp *atexp*	(prefix) alkalmazás	application
atexp	::=	*scon*	állandó	special constant
		<`op`> *longvar*	értékváltozó	value variable
		<`op`> *longcon*	adatkonstruktor	value constructor
		<`op`> *longexcon*	kivételkonstruktor	exception constructor
		`{` < *exprow* > `}`	rekord	record
		# *lab*	rekordszelektor	record selector
		`()`	nullas	0-tuple
		`(`exp_1 `,` …`,` exp_n`)`	ennes, $n \geq 2$	n-tuple, $n \geq 2$
		`[`exp_1 `,` …`,` exp_n`]`	lista, $n \geq 0$	list, $n \geq 0$
		`#[`exp_1 `,` …`,` exp_n`]`	vektor, $n \geq 0$	vector, $n \geq 0$
		`(`exp_1 `;` …`;` exp_n`)`	kifejezéssorozat, $n \geq 2$	sequence, $n \geq 2$
		`let` *dec* `in` exp_1 `;` …`;` exp_n `end`	lokális kifejezés, $n \geq 1$	local expression, $n \geq 1$
		`(` *exp* `)`		
exprow	::=	*lab* `=` *exp* < `,` *exprow* >	kifejezéssor	expression row
match	::=	*mrule* < `|` *match* >	klózsorozat, változatsorozat	match
mrule	::=	*pat* `=>` *exp*	klóz, változat	match rule

A.2.2. Deklarációk és kötések

dec	::=	`val` *tyvarseq valbind*	értékdeklaráció	value declaration
		`fun` *tyvarseq fvalbind*	függvénydeklaráció	function declaration
		`type` *typbind*	típusdeklaráció	type declaration
		`datatype` *datbind* $<$ `withtype` *typbind* $>$	datatype-deklaráció	datatype declaration
		`abstype` *datbind* $<$ `withtype` *typbind* $>$ `with` *dec* `end`	abstype-deklaráció	abstype declaration
		`exception` *exbind*	kivételdeklaráció	exception declaration
		`local` dec_1 `in` dec_2 `end`	lokális deklaráció	local declaration
		`open` $unitid_1$ ... $unitid_n$	open-deklaráció, $n \geq 1$	open declaration, $n \geq 1$
			üres deklaráció	empty declaration
		dec_1 $<$`;`$>$ dec_2	deklaráció-sorozat	sequential declaration
		`infix` $<d>$ id_1 id_n	infix-direktíva, $n \geq 1$	infix (left) directive, $n \geq 1$
		`infixr` $<d>$ id_1 id_n	infixr-direktíva, $n \geq 1$	infix (right) directive, $n \geq 1$
		`nonfix` id_1 ... id_n	nonfix-direktíva, $n \geq 1$	nonfix directive, $n \geq 1$

valbind	::=	*pat* = *exp* $<$ `and` *valbind* $>$	értékkötés	value binding
		`rec` *valbind*	rekurzív kötés	recursive binding

fvalbind ::= $<$`op`$>$ *var* $atpat_{11}$ ... $atpat_{1n}$ $<$`:`$ty>$ = exp_1 $\quad m, n \geq 1$

| $<$`op`$>$ *var* $atpat_{21}$ ... $atpat_{2n}$ $<$`:`$ty>$ = exp_2

| ...

| $<$`op`$>$ *var* $atpat_{m1}$ ... $atpat_{mn}$ $<$`:`$ty>$ = exp_m

$<$ `and` *fvalbind* $>$

typbind ::= *tyvarseq tycon* `=` *ty* $<$ `and` *typbind* $>$

datbind ::= *tyvarseq tycon* `=` *conbind* $<$ `and` *datbind* $>$

conbind ::= $<$`op`$>$ *con* $<$`of` *ty*$>$ $<$ `|` *conbind* $>$

exbind ::= $<$`op`$>$ *excon* $<$`of` *ty*$>$ $<$ `and` *exbind* $>$

$<$`op`$>$ *excon* = `op` *longexcon* $<$ `and` *exbind* $>$

Megjegyzés. *fvalbind* fenti definíciójában, ha *var* infix helyzetűnek van deklarálva, akkor vagy meg kell előznie az `op` szócskának, vagy infix helyzetben kell használni. Ez azt jelenti, hogy a klózok elején `op` *var* `(`*atpat*`,` *atpat´*`)` helyett `(`*atpat var atpat´*`)` írható. A zárójelek elhagyhatók, ha *atpat´* után közvetlenül `:`*ty* vagy `=` áll.

A.2.3. Típuskifejezések

ty	::=	*tyvar*	típusváltozó	type variable
		`{` $<$ *tyrow* $>$ `}`	rekordtípus	record type expression
		tyseq longtycon	típuskonstrukció	type construction
		ty_1 `*` … `*` ty_n	ennes típus, $n \geq 2$	tuple type, $n \geq 2$
		ty_1 `->` ty_2	függvénytípus	function type expression
		`(` *ty* `)`		

tyrow	::=	*lab* `:` *ty* $<$ `,` *tyrow* $>$	típuskifejezés-sor	type-expression row

A.2.4. Minták

atpat	::=	`_`	mindenesjel	wildcard
		scon	állandó	special constant
		$<$`op`$>$ *var*	változó	variable
		$<$`op`$>$ *longcon*	adatkonstruktor	value constructor
		$<$`op`$>$ *longexcon*	kivételkonstruktor	exception constructor
		`{` $<$ *patrow* $>$ `}`	rekord	record
		`()`	nullas	0-tuple
		`(`pat_1`,` …`,` pat_n`)`	ennes, $n \geq 2$	n-tuple, $n \geq 2$
		`[`pat_1`,` …`,` pat_n`]`	lista, $n \geq 0$	list, $n \geq 0$
		`#[`pat_1`,` …`,` pat_n`]`	vektor, $n \geq 0$	vector, $n \geq 0$
		`(` *pat* `)`		

patrow	::=	`...`	mindenesjel	wildcard
		lab `=` *pat* $<$ `,` *patrow* $>$	mintasor	pattern row
		lab $<$`:`*ty*$>$ $<$ `as` *pat* $>$ $<$ `,` *patrow* $>$	címke mint változó	label as variable

pat	::=	*atpat*	atomi minta	atomic pattern
		<`op`> *longcon atpat*	adatkonstrukció	value construction
		<`op`> *longexcon atpat*	kivételkonstrukció	exception construction
		pat_1 *con* pat_2	infix adatkonstrukció	infixed value construction
		pat_1 *excon* pat_2	infix kivételkonstrukció	infixed exception construction
		pat `:` *ty*	minta típusmegkötéssel	typed pattern
		<`op`> *var* <`:`*ty*> `as` *pat*	réteges minta	layered pattern

A.2.5. Szintaktikai korlátozások

- Egy *var* osztálybeli névre egynél többször nem illeszthető minta. Egy *lab* osztálybeli mezőnévre egynél többször nem illeszthető kifejezéssor, mintasor vagy típuskifejezés-sor.
- Egy név csak egyféleképpen köthető le egy *valbind, typbind, datbind* vagy *exbind* deklarációban. Ugyanez érvényes az adatkonstruktorokra egy *datbind* deklarációban.
- Egy *tyvar* osztálybeli típusváltozó nem fordulhat elő egynél többször egy *tyvarseq* sorozatban egy *typbind* vagy *datbind* deklaráció bal oldali *tyvarseq tycon* részében. Minden olyan *tyvar* osztálybeli típusváltozónak, amelyik előfordul a jobb oldalon, szerepelnie kell *tyvarseq*-ben.
- A `rec`-et követő minden *pat* `=` *exp* értékkötésben az *exp*-nek, szükség esetén zárójelek között, `fn` *match* alakúnak kell lennie, ahol a *match*-ekhez egy vagy több típusmegkötés is társítható.
- `true`, `false`, `nil`, `::` és `ref`nem kaphat új értéket egy *valbind, datbind* vagy *exbind* deklarációban. Az `it` név nem kaphat új értéket egy *datbind* vagy *exbind* deklarációban.

B. Függelék

Példaprogramok: fák kezelése

B.1. Fa adott tulajdonságának ellenőrzése (ugyanannyi)

Tekintsük az alábbi adattípus-deklarációt:

```
datatype ´a fa = A | B of ´a fa * ´a fa | C of ´a fa * ´a fa * ´a fa;
```

Írjon `ugyanannyi` néven olyan SML-függvényt, amely egy `´a fa` típusú fáról eldönti, hogy `B` és `C` csomópontjainak ugyanannyi `A` gyermeke (saját levele) van-e! A függvény specifikációja:

```
ugyanannyi f = igaz, ha f B és C csomópontjainak ugyanannyi A gyermeke van
ugyanannyi : ´a fa -> bool
```

Példák:

```
ugyanannyi A = true;
ugyanannyi(B(B(A,A), C(A,A,A))) = false;
ugyanannyi(B(C(B(A,A), B(A,A),B(A,A)), B(C(A,A,A), C(A,A,A)))) = true;
```

1. megoldás

Összeszámoljuk, hogy a `B` és a `C` csomópontoknak hány `A` gyermeke van külön-külön, majd megnézzük, hogy a két szám egyenlő-e.

```
(* ua : ´a fa -> int * int * int
   ua f = hármas: 1., ill. 2. tagja a B, ill. a C csomópontok A leveleinek a
          száma; 3. tagja 1, ha az aktuális csomópont A, egyébként 0
*)
fun ua (B(b1, b2)) =
      let val (b11, c11, a11) = ua b1
          val (b21, c21, a21) = ua b2
      in (b11 + b21 + a11 + a21, c11 + c21, 0)
      end
  | ua (C(c1, c2, c3)) =
      let val (b11, c11, a11) = ua c1
          val (b21, c21, a21) = ua c2
          val (b31, c31, a31) = ua c3
      in (b11 + b21 + b31, c11 + c21 + c31 + a11 + a21 + a31, 0)
      end
  | ua A = (0, 0, 1);
fun ugyanannyi f = let val (b, c, _) = ua f in b = c end;
```

A rekurzió befejeződését korábban *teljes indukcióval* igazoltuk, rekurzív adattípusok, pl. listák és fák esetén *strukturális indukcióval* bizonyítjuk.

A teljes indukciót az egész számok halmazán értelmezzük, és azon alapul, hogy minden egész után egy nála eggyel nagyobb egész *következik*. Az `INT` típust így is lehetne deklarálni: `datatype INT = 0 | Succ of INT`. A strukturális indukció a teljes indukció általánosítása rekurzív adattípusokra; szembetűnő a hasonlóság az `INT` típus deklarációja és például a `datatype ´a List = Nil | Cons of ´a List` deklaráció között.

Az adott esetben a kiértékelés biztosan véget ér, mert `ua`-t vagy rekurzív módon alkalmazzuk az aktuális `B` vagy `C` fa egy részfájára, amely biztosan rövidebb az aktuális fánál, vagy befejeződik a hívás, mert az aktuális fa `A`.

2. megoldás

A második paraméterként átadott számlálót eggyel növeljük, ha `B`-nek van `A` gyermeke, és csökkentjük, ha `C` nek van `A` gyermeke. (`ua` és `ba`, ill. `ua` és `ca` kölcsönösen rekurzív függvények.)

```
(* ua : ´a fa * int -> int
   ua(f, num) = num + az f-beli B-k A gyermekeinek száma -
                az f-beli C-k A gyermekeinek száma
*)
fun ua(A, num) = num
  | ua(B(x, y), num) = ba(x, ba(y, num))
  | ua(C(x, y, z), num) = ca(x, ca(y, ca(z, num)))
(* ba : ´a fa * int -> int
   ba(f, num) = num + a B-k A leveleinek száma
*)
and ba(A, num) = num + 1
  | ba(x, num) = ua(x, num)
(* ca : ´a fa * int -> int
   ca(f, num) = num - a C-k A leveleinek száma
*)
and ca(A, num) = num - 1
  | ca(x, num) = ua(x, num);
fun ugyanannyi f = ua(f, 0) = 0;
```

3. megoldás

A 2. megoldás egyszerűsített változata egy újabb paramétert használ a *növekmény* átadására. Ennek értéke `B` csomópont esetén $+1$, `C` csomópont esetén pedig -1. (Szeredi Péter megoldása.)

```
local
  (* ua : ´a fa * int * int -> int
     ua (f, num, incr) = num + incr + az f-beli B-k A gyermekeinek száma -
                             az f-beli C-k A gyermekeinek száma
  *)
  fun ua (C(c1, c2, c3), num, incr) =
               ua(c1, ua(c2, ua(c3, num, ~1), ~1), ~1)
    | ua (B(b1, b2), num, incr) = ua(b1, ua(b2, num, 1), 1)
    | ua (A, num, incr) = num + incr
in
   fun ugyanannyi f = ua(f, 0, 0) = 0
end;
```

B.2. Fa adott tulajdonságú részfáinak száma (bea)

Tekintsük az alábbi adattípus-deklarációt:

```
datatype fa = A | B of fa * fa | C of fa * fa * fa;
```

Írjon `bea` néven olyan SML-függvényt, amely megszámlálja egy fa típusú fában azokat a `B` csomópontokat, amelyeknek minden részfája `B` vagy `A` (de nem `C`), és ezeknek a számát adja eredményül! Segédfüggvényt definiálhat.

```
bea f = azoknak az f-beli B-knek a száma, amelyeknek csak B vagy A
        részfájuk van
bea : fa -> int
```

Példák:

```
bea A = 0;
bea(B(B(A,A),C(A,A,A))) = 1;
bea(B(C(B(A,A),C(A,A,A),B(A,A)),B(B(A,A),C(A,B(A,A),A)))) = 4;
```

1. megoldás

A `ba` segédfüggvény az olyan `B`-ket számlálja meg , amelyeknek egyetlen utódja sem `C`.

```
(* ba f = olyan (b, c) pár, ahol b a jó B-k száma f-ben, és c = true,
          ha f-ben van C
   ba : fa -> int * bool
*)
fun ba A = (0, false)
  | ba (C(bf, kf, jf)) =
       let val (bb, _) = ba bf
           val (kb, _) = ba kf
           val (jb, _) = ba jf
       in
          (bb+kb+jb, true)
       end
  | ba (B(bf, jf)) =
       let val (bb, bc) = ba bf
           val (jb, jc) = ba jf
           val b = bc orelse jc
       in
          (bb + jb + (if b then 0 else 1), b)
       end;
fun bea f = #1 (ba f);
```

Ha az aktuális fa `A`, a jó `B`-k száma nem változik, az ősei között pedig lehetnek jó `B`-k (ezért `false` az eredménypár második tagja). Ha az aktuális fa `C`, a részfái és ezek utódai között lehetnek jó `B`-k, de az ősei között egyetlen `B` sem lehet jó (ezért `true` az eredménypár második tagja). Ha az aktuális fa `B` és az utódai között nincs `C`, akkor 1-gyel megnöveljük a jó `B`-k számát, egyébként nem módosítjuk; az utódokra vonatkozó információt minden esetben változatlanul adjuk tovább.

A `bea` függvény a `ba` segédfüggvény által előállított eredménypár első tagját adja eredményül.

2. megoldás

Ez a megoldás rosszabb hatékonyságú, mert a részfákat többször is bejárja, a már megszerzett információt nem használja fel újra.

```
fun bea f =
      let (* csupaAvB f = igaz, ha f-nek nincs C részfája
             csupaAvB : fa -> bool
          *)
          fun csupaAvB (B(A, A)) = true
            | csupaAvB (B(b1, A)) = csupaAvB b1
            | csupaAvB (B(A, b2)) = csupaAvB b2
            | csupaAvB (B(b1, b2)) = csupaAvB b1 andalso csupaAvB b2
            | csupaAvB _ = false

          (* szamol f = f jó B csomópontjainak száma
             szamol : fa -> int
          *)
          fun szamol A = 0
            | szamol (B(A, A)) = 1
            | szamol (b as B(f1, f2)) =
                    szamol f1 + szamol f2 + (if csupaAvB b then 1 else 0)
            | szamol (c as C(f1, f2, f3)) =
                    szamol f1 + szamol f2 + szamol f3
      in
         szamol f
      end;
```

B.3. Fa adott tulajdonságú részfáinak száma (testverE)

Írjon `testverE` néven olyan SML-függvényt, amely a

```
datatype ´a fa = E | N of ´a fa * ´a fa * ´a fa;
```

deklarációval megadott fában meghatározza azoknak az `E` leveleknek a számát, amelyeknek legalább egy testvérük van! Egy `E` levél testvérének az ugyanahhoz az `N` csomóponthoz tartozó másik `E` levelet nevezzük.

A függvény specifikációja:

```
testverE f = az E testvérek száma az f fában
testverE : ´a fa -> int
```

Példák:

```
testverE E = 0;
testverE (N(E,E,E)) = 3;
testverE (N(E,N(E,E,E),N(N(E,E,E),E,E))) = 8;
testverE (N(E,N(E,E,E),N(E,E,E))) = 6;
```

Megoldás

A feladat és a megoldása nagyon egyszerű. Ügyeltünk arra, hogy csak a valóban megkülönböztetendő esetekre írjunk fel változatokat. Figyelje meg, hogy az `N(E,f2,E)` és az `N(f1,E,E)` eseteket visszavezettük az `N(E,E,f3)` esetre. Ezzel ugyan egy lépéssel mélyítettük a rekurziót, de ha később a program adott ágát javítani, módosítani kell, csökkent a hibák elkövetésének lehetősége.

```
fun testverE (N(E,E,E)) = 3
  | testverE (N(E,E,f3)) = 2 + testverE f3
  | testverE (N(E,f2,E)) = testverE(N(E,E,f2))
  | testverE (N(f1,E,E)) = testverE(N(E,E,f1))
  | testverE (N(f1,f2,f3)) = testverE f1 + testverE f2 + testverE f3
  |testverE E = 0;
```

B.4. Fa adott elemeinek összegzése (szintOssz)

Írjon `szintOssz` néven olyan SML-függvényt, amely egy bináris fában tárolt értékek szintenkénti összegéből alkotott listát ad eredményül! A lista első eleme az első szinten lévő gyökérelem értéke, második eleme a második szinten tárolt, legfeljebb két elem összege s.í.t. A fa típusa:

```
datatype itree = L of int | N of itree * int * itree;
```

A függvény specifikációja:

```
szintOssz t = a t-beli elemek szintenkénti összegének listája
szintOssz : itree -> int list
```

Példák:

```
szintOssz(L 1999) = [1999];
szintOssz(N(N(N(L 8, 4, L 9), 2, L 5), 1, N(L 6, 3, L 7))) = [1, 5, 22, 17];
```

A második példában használt fa és a szintenkénti összegek:

```
          1           1
        /   \
       2     3        5
      / \   / \
     4   5 6   7     22
    / \
   8   9             17
```

1. megoldás

`listaOsszeg` két, esetleg különböző hosszúságú lista elemeinek páronkénti összegéből álló listát ad eredményül. (A rövidebb listából hiányzó elemeket "pótolja".) Jobbrekurzív változata az elemek sorrendjét megfordítaná, ezért az eredeti sorrendet `rev`-vel helyre kellene állítani.

```
local
  (* listaOsszeg (xs, ys) = az xs és ys elemeiből páronként képzett
                            összegek listája
     listaOsszeg : int list * int list -> int list
  *)
  fun listaOsszeg (x::xs, y::ys) = x+y::listaOsszeg(xs, ys)
    | listaOsszeg ([], ys) = ys
    | listaOsszeg (xs, []) = xs
```

```
in
   fun szintOssz(N(left, x, right)) =
                x :: listaOsszeg(szintOssz left, szintOssz right)
     | szintOssz (L x) = [x]
end;
```

A `szintOssz` függvény az `N` csomópontban előállítja a bal, ill. a jobb részfa szintenkénti összegeinek listáját, majd a két lista elemeit páronként összeadja. Az `L` levél egyetlen eleméből egyelemű listát képez.

2. megoldás

Az `sO` segédfüggvény a `t` fa azonos szintjein lévő elemeket hozzáadja az `xs` lista megfelelő eleméhez, és ezt a listát adja eredményül. A fa gyökere a lista jobb szélső elemének felel meg: ahogy egyre mélyebbre haladunk a fában, úgy építjük a listát, ill. haladunk jobbról balra a már felépült listában.

```
local
  (* sO(t, xs) = az egyes szinteken lévő t-beli és a megfelelő xs-beli
                 elemek összegének a listája
     sO : itree * int list -> int list
  *)
  fun sO (L v, []) = [v]
    | sO (L v, x::xs) = x+v::xs
    | sO (N(l, v, r), []) = v::sO(l, sO(r, []))
    | sO (N(l, v, r), x::xs) = x+v::sO(l, sO(r, xs))
in
   fun szintOssz t = sO(t, [])
end;
```

3. megoldás

Vegyük észre, hogy a 2. megoldásban az `sO` segédfüggvény két-két klóza alig különbözik egymástól. A hasonlóságot még jobban kiemelhetjük:

```
  fun sO (L v, xs as []) = 0+v::xs
    | sO (L v, x::xs) = x+v::xs
    | sO (N(l, v, r), xs as []) = 0+v::sO(l, sO(r, xs))
    | sO (N(l, v, r), x::xs) = x+v::sO(l, sO(r, xs));
```

Az egymáshoz hasonló klózokat összevonhatjuk (Szeredi Péter megoldása):

```
local
(* feje : int list -> int
   feje xs = hd xs vagy 0, ha xs = []
*)
  fun feje [] = 0
    | feje (x::_) = x

(* farka : ´a list -> ´a list
   farka xs = tl xs vagy [], ha xs = []
*)
  fun farka [] = []
    | farka (_::xs) = xs
```

```
  (* s0(t, xs) = az egyes szinteken lévő t-beli és a megfelelő
                 xs-beli elemek összegének a listája
     s0 : itree * int list -> int list
  *)
  fun s0 (L v, xs) = feje xs + v :: farka xs
    | s0 (N(l, v, r), xs) = feje xs + v :: s0(l, s0(r, farka xs))
in
   fun szintOssz t = s0(t, [])
end;
```

B.5. Kifejezésfa egyszerűsítése (egyszerusit)

Az alábbi adattípus-definíciók olyan kifejezésfát írnak le, amelynek a levelei egész számok, a gyökérelemei pedig a ++, --, ** és // műveleti jelek:

```
datatype oper = ++ | -- | ** | //;
datatype ExTr = Lf of int | Br of oper * ExTr * ExTr;
```

Írjon olyan SML-függvényt `egyszerusit` néven, amely egy kifejezésfában az `m++n` alakú részkifejezések összes előfordulását az összegükre, az `m**n` alakú részkifejezések összes előfordulását a szorzatukra cseréli, a többi részkifejezést pedig változatlanul hagyja (`m` és `n` egész számok)!

Gondoljon a redukció során keletkező, hasonló alakú részkifejezések helyettesítésére, de kerülje el a végtelen rekurziót!

A függvény specifikációja:

```
egyszerusit kf = kf egyszerűsített változata, amelyben m++n,
                 ill. m**n összes előfordulása helyén m és n összege,
                 ill. szorzata van
egyszerusit : ExTr -> ExTr
```

Példák:

```
egyszerusit(Br(++,Lf 1,Lf 2)) = Lf 3;
egyszerusit(Br(//, Br(++,Br( **,Lf 3,Lf 4), Br(++,Lf 5,Lf 6)),Lf 7)) =
                             Br(//,Lf 23,Lf 7);
egyszerusit(Br(//,Br(//,Lf 3,Lf 4),Lf 5)) =
                             Br(//,Br(//,Lf 3,Lf 4),Lf 5);
egyszerusit(Br(//,Br(--,Br( **,Lf 3,Lf 4), Br(++,Lf 5,Lf 6)),Lf 7)) =
                             Br(//, Br(--, Lf 12, Lf 11), Lf 7);
```

1. megoldás

A ++ és a `**` műveleti jeleket tartalmazó részkifejezések kezelésére két-két változatot kell írni: egyet-egyet a ++, ill. `**` műveleti jelből és pontosan két levélből (jelöljük `Lf b` -vel és `Lf j`-vel) álló, és egyet-egyet a ++, ill. `**` műveleti jelből és egyéb részfákból álló csomópontok kezelésére.

Az első két esetben a kijelölt művelet elvégezhető, az eredmény az `Lf(b+j)`, ill. az `Lf(b*j)` levél. A két utóbbi esetben `egyszerusit` rekurzív hívásával először is a bal és a jobb részfát egyszerűsítjük. Előfordulhat, hogy mindkettő levéllé válik , ezért meg kell próbálni, hátha további rekurzív hívással lehet egyszerűsíteni az adott műveleti jelből és a redukált részfákból összerakott fát is.

Ha az aktuális fa gyökerében más műveleti jel van, `egyszerusit` rekurzív hívásával csak a bal és a jobb részfát egyszerűsítjük, további redukcióra nincs lehetőség. Nem lehet egyszerűsíteni a kifejezést akkor sem, ha az aktuális fa levél.

```
fun egyszerusit (Br( ++, Lf b, Lf j)) = Lf(b+j)
  | egyszerusit (Br( **, Lf b, Lf j)) = Lf(b*j)
  | egyszerusit (Br( ++, bf, jf)) =
                egyszerusit(Br( ++, egyszerusit bf, egyszerusit jf))
  | egyszerusit (Br( **, bf, jf)) =
                egyszerusit(Br( **, egyszerusit bf, egyszerusit jf))
  | egyszerusit (Br(mj, bf, jf)) =
                Br(mj, egyszerusit bf, egyszerusit jf)
  | egyszerusit (kf as Lf v) = kf;
```

2. megoldás

Három változat (klóz) összevonásával, valamint a közös részek kiemelésével a megoldás rövidebbé tehető.

```
fun egyszerusit (Br( ++, Lf b, Lf j)) = Lf(b+j)
  | egyszerusit (Br( **, Lf b, Lf j)) = Lf(b*j)
  | egyszerusit (Br(mj, bf, jf)) =
      let val f = Br(mj, egyszerusit bf, egyszerusit jf)
      in
        if mj = ++ orelse mj = ** then egyszerusit f else f
      end
  | egyszerusit (kf as Lf v) = kf;
```

B.6. Kifejezésfa egyszerűsítése (`coeff`)

Tekintse az alábbi típust és adattípust:

```
type term = int * char;
datatype expr = ++ of expr * term | Z;
infix 6 ++;
```

Egy `term` típusú párt egy egész együttható és egy `char` típusú változónév szorzatának, egy `expr` típusú kifejezést `term` típusú tagok és `Z` (zérus) állandók összegének tekintünk.

Írjon SML-függvényt `coeff` néven, amelynek `expr` típusú kifejezésből és `char` típusú változónévből álló pár az argumentuma, és az eredménye az adott változó együtthatóinak az összege az adott kifejezésben! Hatékony, jobbrekurzív programot írjon! Segédfüggvényt definiálhat.

A függvény specifikációja:

```
coeff (e, v) = v együtthatóinak az összege e-ben
coeff : expr * char -> int
```

Példák:

```
coeff(Z ++ (2,#"a") ++ (3,#"b") ++ (~5,#"a") ++ (4,#"c"), #"a") = ~3;
coeff(Z ++ (2,#"a") ++ (3,#"b") ++ (~5,#"a") ++ (4,#"c"), #"x") = 0;
```

Megoldás

Figyeljük meg, hogy a `Z` az `expr` típusú kifejezések *baloldali egységeleme;* `Z` maga is `expr` típusú kifejezés, az `expr` típusú kifejezésekben pedig csak a bal oldalon állhat `expr` típusú kifejezés.

A `cf` segédfüggvény az `n` argumentumban gyűjti az `e`-beli `v`-k együtthatóinak az összegét. `v` `coeff`-ben lokális, a `cf` szempontjából azonban globális név.

```
fun coeff (e, v) =
        (* cf e n = n + a v együtthatóinak az összege e-ben
           cf : expr -> int -> int
        *)
     let fun cf (e ++ (c, v0)) n = cf e (n + (if v = v0 then c else 0))
           | cf Z n = n
     in
        cf e 0
     end;
```

B.7. Szövegfeldolgozás (parPairs)

Írjon SML-függvényt `parPairs` néven, amely az argumentumként kapott füzérben található, egymáshoz tartozó kerek nyitó- és csukózárójelek pozícióiból alkotott párok listáját adja eredményül, tetszőleges sorrendben! A füzér karaktereit 1-től számozzuk. Segédfüggvényt definiálhat.

A függvény specifikációja:

```
parPairs s = az s füzérbeli, egymáshoz tartozó, kerek nyitó- és
             csukózárójelek pozícióiból alkotott párok listája
parPairs : string -> (int * int) list
```

Példák:

```
parPairs "Zárójelmentes." = [];
parPairs ")" = [];
parPairs "(" = [];
parPairs "real(3*4) + (sin(0.5) - (11.4+3.4)) * 1.2" =
                 [(5, 11), (19, 23), (27, 38), (15, 39)];
```

Megoldás

Az alábbi megoldásban kihasználjuk, hogy a lista veremként, azaz LIFO-tárként használható: amit legutoljára rakunk bele, azt vesszük ki belőle legközelebb.

A füzért az `explode` függvény karakterlistává alakítja. A `pp` segédfüggvény, ha kerek nyitózárójelet talál, az indexét (azaz helyének sorszámát az eredeti füzérben), berakja a `bs` verembe. Ha csukózárójelet talál, és a `bs` verem nem üres, kiveszi a `bs`-ből a megfelelő nyitózárójel indexét, és a `(b, i)` párt berakja `ps`-be. Ha egyéb karaktert talál, egyszerűen továbblép a listában. `i` értékét minden egyes lépésben 1-gyel megnöveli. Ha a lista elfogy, a indexpárok `ps`-ben összegyűjtött listáját az eredeti sorrendbe rakva (azaz `rev`-et alkalmazva) adja eredményül.

Megjegyzés:. A `pp` segédfüggvényt nem lenne könnyű deklaratív módon specifikálni, ezért megelégszünk a műveleti szemléletű specifikációval.

```
fun parPairs s =
      let (* pp (cs, i, bs, ps) =
                     cs = a feldolgozandó karakterek listája;
                     i  = a cs első karakterének az indexe
                          (= helye az eredeti füzérben);
                     bs = a még le nem zárt nyitózárójelek indexének
                          fordított sorrendű listája;
                     ps = az egymáshoz tartozó nyitó- és csukózárójelek
                          indexeiből álló párok listája
```

```
       pp : char list * int * int list * (int * int) list ->
                                                  (int * int) list
    *)
   fun pp (#"("::cs, i, bs, ps) = pp(cs, i+1, i::bs, ps)
     | pp (#")"::cs, i, b::bs, ps) = pp(cs, i+1, bs, (b,i)::ps)
     | pp (_::cs, i, bs, ps) = pp(cs, i+1, bs, ps)
     | pp ([], _, _, ps) = rev ps
in
   pp(explode s, 1, [], [])
end;
```

C. Függelék

Példaprogramok: füzérek és listák kezelése

C.1. Füzér adott tulajdonságú elemei (mezok)

Írjon mezok néven olyan SML-függvényt, amelynek egy füzér maximális hosszúságú, nemüres részeiből álló lista az eredménye! E lista elemei az eredeti sorrendben felsorolt olyan füzérek, amelyek vagy csak a megadott, zárt intervallumba tartozó, vagy csak az ezen kívül eső karakterekből állnak. Használhatja a String.tokens és a String.isPrefix magasabb rendű függvényeket. Segédfüggvényt definiálhat (pl. két lista összefuttatására).

A függvény specifikációja:

```
mezok(s, c1, c2) = az s füzér olyan maximális hosszúságú, nemüres,
      folytonos részeinek az eredeti sorrendet megőrző listája, ahol
      a listaelemekben minden karakter kódja vagy a [c1, c2] zárt
      intervallumba, vagy azon kívül esik
    mezok : string * char * char -> string list
```

Példa:

```
mezok ("Ali baba + a 40 rablo", #"a", #"n") =
  ["A", "li", " ", "baba", " + ", "a", " 40 r", "abl", "o"];
```

1. megoldás

Egy-egy listába szétválogatjuk a füzér adott intervallumba eső, ill. azon kívüli karakterekből álló szakaszait, majd a két lista elemeit a megfelelő sorrendben összefuttatjuk. A megfelelő sorrend meghatározására megvizsgáljuk, hogy melyik lista első elemével kezdődik az eredeti füzér.

```
fun mezok(s, c1, c2) =
      let (* jokar c = igaz, ha c a [c1, c2] zárt intervallumba esik
               jokar : char -> bool
          *)
          fun joKar c = c >= c1 andalso c <= c2
          (* osszefuttat(xs, ys, zs) = az xs és az ys elemei váltakozva
                                       a zs elé fűzve
             PRE : |length xs - length ys| <= 1
             osszefuttat : ´a list * ´a list * ´a list -> ´a list
          *)
```

```
        fun osszefuttat (x::xs, y::ys, zs) = osszefuttat(xs, ys, y::x::zs)
          | osszefuttat ([], [y], zs) = rev(y :: zs)
          | osszefuttat ([x], [], zs) = rev(x :: zs)
          | osszefuttat (_, _, zs) = rev zs  (* lehetetlen eset *)

        val is = String.tokens (not o joKar) s
        val os = String.tokens joKar s
    in
        if String.isPrefix (hd is) s
        then osszefuttat(is, os, [])
        else osszefuttat(os, is, [])
    end;
```

2. megoldás

Ez a megoldás nem használja sem a `String.tokens`, sem a `String.isPrefix` függvényt.

```
exception Mezok;
fun mezok ("", _, _) = []
  | mezok (s, c1, c2) =
    let (* jokar c = igaz, ha c a [c1, c2] zárt intervallumba esik
              jokar : char -> bool
        *)
        fun joKar c = c >= c1 andalso c <= c2
        (* mezok0(cs, js, rs, ts) =
              mezok0 : char list * char list * char list * char list list ->
                                                                char list list
        *)
        fun mezok0 (c::cs, js, [], ts) =
                if joKar c
                then mezok0(cs, c::js, [], ts)
               else mezok0(cs, [], [c], rev js::ts)
          | mezok0 (c::cs, [], rs, ts) =
                if joKar c
                then mezok0(cs, [c], [], rev rs::ts)
                else mezok0(cs, [], c::rs, ts)
          | mezok0 ([], jjs as j::js, [], ts) = rev jjs :: ts
          | mezok0 ([], [], rrs as r::rs, ts) = rev rrs :: ts
          | mezok0 _ = raise Mezok (* lehetetlen eset *)
        val (c::cs) = explode s
        val (js, rs) = if joKar c then ([c],[]) else ([],[c])
    in
        map implode (rev(mezok0(cs, js, rs, [])))
    end;
```

C.2. Füzér adott tulajdonságú elemei (`basename`)

Írjon olyan SML-függvényt `basename` néven, amely egy füzérként megadott állománynév utolsó nemüres komponensét adja eredményül! A névben egymás után többször szereplő `#"/"` karaktereket egyetlen `#"/"` karakternek vegye! Legalább egyet alkalmazzon a `String.fields`, `String.tokens`, `List.nth`, `List.length`, `List.last`, `List.take` függvények közül!

A függvény specifikációja:

```
basename s = az s állománynév utolsó nemüres komponense
PRE: s <> ""
basename : string -> string
```

Példák:

```
basename "dr-1/dr-2/file.ext" = "file.ext";
basename "dr-1//dr-2///file.ext" = "file.ext";
basename "/dr-1/file.ext/" = "file.ext";
basename "/dr-1/file.ext///" = "file.ext";
basename "///" = "";
basename "/" = "";
```

Megoldás

A megoldás nagyon egyszerű, ha a megfelelő könyvtári függvényeket használjuk.

```
fun basename s = let val ts = String.tokens (fn c => c = #"/") s
                 in
                    if null ts then "" else List.last ts
                 end;
```

C.3. Füzér adott tulajdonságú elemei (rootname)

írjon olyan SML-függvényt `rootname` néven, amely egy füzérként megadott állománynév első nemüres, esetleg `#"/"` jellel kezdődő komponensét adja eredményül! A névben egymás után többször szereplő `#"/"` karaktereket egyetlen `#"/"` karakternek vegye! Legalább egyet alkalmazzon a `String.fields`, `String.tokens`, `List.nth`, `List.length`, `List.last`, `List.take`, `List.drop` függvények közül!

A függvény specifikációja:

```
rootname s = az s állománynév első nemüres komponense
PRE: s <> ""
rootname : string -> string
```

Példák:

```
rootname "dr-1/dr-2/file.ext" = "dr-1";
rootname "/dr-1/file.ext///" = "/dr-1";
rootname "///file.ext///" = "/file.ext";
rootname "/" = "/";
rootname "///" = "/";
```

Megoldás

A megoldás most is nagyon egyszerű, ha a megfelelő könyvtári függvényeket használjuk.

```
fun rootname s = let val ts = rev(String.tokens (fn c => c = #"/") s)
                 in
                    if null ts then "/"
                    else (if String.sub(s,0) = #"/"
                          then "/"
                          else "") ^ List.last ts
                 end;
```

C.4. Lista adott tulajdonságú részlistái (`szomsor`)

Írjon olyan SML-függvényt `szomsor` néven, amely előállítja egy adott egészszám-lista olyan (általában nem folytonos, de a sorrendet megőrző) rész-listáinak listáját, amelyben a rész-listák legalább kételemű, maximális (egyik irányban sem kiterjeszthető), egyesével növekvő számtani sorozatot alkotnak! A listáról felteheti, hogy csupa különböző egészből áll.

```
(* szomsor ns = ns legalább kételemű, maximális (egyik irányban sem
        kiterjeszthető), egyesével növekvő számtani sorozatot alkotó
        rész-listáinak listája (feltehető, hogy ns minden eleme különböző)
    szomsor : int list -> int list list
*)
```

Segítség: Írjon segédfüggvényt, amely az egészlista első elemével kezdődő számtani sorozatot alkotó listát és a számtani sorozatban fel nem használt elemek listáját adja vissza párként, majd vizsgálja meg, hogy van-e még mit feldolgozni. Csak legalább két elemű sorozatokat fűzzön az eredménylistához!

Példa:

```
szomsor [3,2,4,9,7,1,8,5,6] = [[3,4,5,6], [7,8]];
```

Megoldás

Kezdjük a segédfüggvénnyel!

```
(* szsor ns ss ms = pár, amelynek első tagja az ns első elemével
          kezdődő, a feltételeknek megfelelő számtani sorozat,
          második tagja ns fel nem használt elemeinek a listája,
          mindkettő az eredeti sorrendben
  szsor ns ss ms : int list * int list * int list -> (int list * int list)
*)
fun szsor [] ss ms = (rev ss, rev ms)
  | szsor (n::ns) [] ms = szsor ns [n] ms
  | szsor (n::ns) (sss as s::ss) ms =
          if n=s+1
          then szsor ns (n::sss) ms
          else szsor ns sss (n::ms);

fun szomsor (nns as n1::n2::ns) =
      let val (ss, ms) = szsor nns [] []
      in if length ss >= 2
         then ss :: szomsor ms
         else szomsor ms
      end
  | szomsor _ = [];
```

C.5. Bináris számok inkrementálása (`binc`)

Írjon `binc` néven SML-függvényt egy listaként ábrázolt bináris szám inkrementálására! A bináris számjegyeket az `I` és `O` adatkonstruktorokkal jelöljük, típusuk:

```
datatype bin = I | O;
```

A listák feje a legnagyobb helyiértékű bináris számjegy, amely sohasem 0. A függvény specifikációja:

```
binc ns = az ns-ben tárolt bináris számot inkrementálja
binc : bin list -> bin list
```

Példák:

```
binc [0] = [I];
binc [I] = [I,0];
binc [I,I,I,I,I,I,I,I] = [I,0,0,0,0,0,0,0,0];
```

Megoldás

Először olyan segédfüggvényt írunk, amely fordított sorrendű bináris számokat inkrementál, és a bináris számjegyeket egy gyűjtőargumentumban gyűjti.

```
local
    (* bi (bs, c, rs) = rs a bs fordított sorrendű bináris szám normál
                        sorrendű inkrementáltja (a bs lista feje
                        a legkisebb helyiértékű bit, c a növekmény)
       bi : (bin list * bin * bin list) -> bin list
    *)
    fun bi (b::bs, 0, rs) = bi(bs, 0, b::rs)
      | bi (0::bs, I, rs) = bi(bs, 0, I::rs)
      | bi (I::bs, I, rs) = bi(bs, I, 0::rs)
      | bi ([], 0, rs) = rs
      | bi ([], I, rs) = I::rs
in
    (* binc ns = az ns bináris szám inkrementáltja
       binc : bin list -> bin list
    *)
    fun binc ns = bi(rev ns, I, [])
end;
```

Kiegészítésképpen írjunk egy-egy segédfüggvényt `bin list` típusú bináris számok és `int` típusú egészek egymásba alakítására `b2i`, ill. `i2b` néven!

```
(* b2i bs = a bs bináris szám egész számként, a vezető 0-k nélkül
   b2i : bin list -> int
*)
fun b2i bs =
     case Int.fromString(implode(map (fn 0 => #"0" | I => #"1") bs)) of
          SOME i => i
        | NONE   => 0;

app load ["Int"];
(* i2b i = az i egész bináris megfelelője (0 -> 0, nem 0 -> I),
           a vezető 0-k nélkül
   i2b : int -> bin list
*)
fun i2b i = map (fn #"0" => 0 | _ => I) (explode(Int.toString i));
```

C.6. Mátrix transzponáltja (`trans`)

Írjon `trans` néven olyan SML-függvényt, amely előállítja egy mátrix transzponáltját! Egy mátrixot sorok listájaként adunk meg, ahol a sorok a mátrixelemek listái. (Egy `a[n,m]` `n*m`-es mátrix transzponáltja az a `b[m,n]` `m*n`-es mátrix, ahol `b[k,l]` = `a[l,k]`.) Könyvtári függvényeket használhat, de saját segédfüggvényt ne definiáljon!

A függvény specifikációja:

```
trans mss = az mss mátrix transzponáltja
trans : ´a list list -> ´a list list
```

Példa:

```
trans [[1,2,3], [4,5,6], [7,8,9], [0,0,0]] =
                         [[1,4,7,0], [2,5,8,0], [3,6,9,0]];
```

1. megoldás

A mátrix sorainak – a részlistáknak – az első elemét, ill. a többi elemét rekurzív módon egy-egy listába (`ms1`, `mss1`) gyűjtjük, amíg csak vannak feldolgozatlan elemek. Nem elég a teljes lista nem üres voltát vizsgálni, azt is meg kell nézni, hogy az egyes részlisták nem üresek-e: az utóbbit ellenőrzi `List.exists`.

```
fun trans mss =
      if (not o null) mss andalso List.exists (not o null) mss
      then
          let val (ms1, mss1) = (map hd mss, map tl mss)
          in
              ms1 :: trans mss1
          end
      else
          [];
```

2. megoldás

Valamivel hatékonyabb az alábbi megoldás, mert a teljes lista üres voltát csak egyetlen egyszer ellenőrzi. Ha nem volt üres kezdetben, nem is válhat azzá menet közben: csak a részlistái válnak üressé a feldolgozás végére.

```
fun trans [] = []
  | trans mss = if List.exists null mss then []
                else let val (ns, nss) = (map hd mss, map tl mss)
                     in ns :: trans nss
                     end;
```

3. megoldás

Talán ez a lehető legtömörebb megoldás: az egyik `map` kigyűjti a listák fejét, a másik pedig a farkát, amire azután rekurzívan alkalmazzuk a `trans` függvényt.

```
fun trans [] = []
  | trans ([]::_) = []
  | trans mss = map hd mss :: trans(map tl mss);
```

4. megoldás

Kevésbé hatékony a `tabulate` függvényt használó, két egymásba ágyazott ciklusra épülő megoldás:

```
fun trans [] = []
  | trans mss = List.tabulate(length(hd mss),
                  fn r => List.tabulate(length mss,
                  fn c => List.nth(List.nth(mss, c), r)));
```

D. Függelék

Válogatás az SML´97 könyvtáraiból

A zárthelyin és a vizsgán a *-gal megjelölt szakaszokban felsorolt típusok, értékek, jelölések, konstruktorok, kivételek és függvények ismeretét várjuk el.

Belső típusok*. bool, char, exn, int, ´a list, ´a option, order, real, string, unit, word, word8.

Belső kivételek*. Bind, Chr, Domain, Div, Fail, Interrupt, Io, Match, Option, Ord, Overflow, Size, Subscript.

Belső függvények a kezdeti környezetben*. ~, +, -, *, /, ^, ::, @, =, <>, <, <=, >=, >, abs, app, ceil, chr, concat, div, explode, false, floor, foldl, foldr, hd, implode, length, map, mod, not, null, o, ord, print, real, rev, round, size, str, tl, true, trunc.

Csak interaktív módban használható belső függvények*. compile, load, loadOne, printVal, printDepth, printLength, quit, system, use.

A Bool könyvtárból*. bool, not, toString, fromString.

A Char könyvtárból*. char, minChar, maxChar, maxOrd,chr, ord, succ, pred, isLower, isUpper, isDigit, isAlpha, isHexDigit, isAlphaNum, isPrint, isSpace, isGraph, isPunct, isCntrl, isAscii, toLower, toUpper, contains, notContains, fromString, toString, <, <=, >, >=, compare.

A General könyvtárból*. A General könyvtár definiálja a belső típusokról, a belső kivételekről, valamint a belső függvényekről szóló szakaszokban felsorolt neveket .

Az Int könyvtárból*. int, precision, minInt, maxInt, ~, *, div, mod, quot, rem, +, -, <, <=, >, >=, abs, min, max, sign, compare, toString, fromString.

A List könyvtárból*. ´a list, null, hd, tl, last, nth, take, drop, length, rev, @, concat, revAppend, app, map, mapPartial, find, filter, partition, foldr, foldl, exists, all, tabulate.

A Listsort könyvtárból*. sort, sorted.

A Math könyvtárból*. real, pi, e, sqrt, sin, cos, tan, atan, asin, acos, atan2, exp, pow, ln, log10, sinh, cosh, tanh.

Az Option könyvtárból*. Option, getOpt, isSome, valOf, filter, map, app, join, compose, mapPartial, composePartial.

A Real könyvtárból*. ~, +, -, *, /, abs, min, max, sign, compare, fromInt, floor, ceil, trunc, round, <, <=, >, >=, toString, fromString.

A String könyvtárból*. string, maxSize, size, sub, substring, extract, concat, ^, str, implode, explode, map, translate, tokens, fields, isPrefix, compare, collate, <, <=, >, >=.

A TextIO könyvtárból. instream, outstream, openIn, closeIn, input, inputAll, inputLine, endOfStream, lookahead, stdIn, openOut, openAppend, closeOut, output, output1, flushOut, stdOut, stdErr, print.

A Time könyvtárból. time, Time, zeroTime, now, toSeconds, toMilliseconds, toMicroseconds, fromSeconds, fromMilliseconds, fromMicroseconds, fromReal, toReal, toString, fromString, +, -, <, <=, >, >=, compare.

A Timer könyvtárból. cpu_timer, real_timer, startCPUTimer, totalCPUTimer, checkCPUTimer, startRealTimer, totalRealTimer, checkRealTimer.

A Word könyvtárból. word, wordSize, orb, andb, xorb, notb, <<, >>, ~>>, +, -, *, div, mod, >, <, >=, <=, compare, min, max, toString, fromString, toInt, toIntX, fromInt.

A Word8 könyvtárból. word, word8, wordSize, orb, andb, xorb, notb, <<, >>, ~>>, +, -, *, div, mod, >, <, >=, <=, compare, min, max, toString, fromString, toInt, toIntX, fromInt.

Az összefoglalót a Moscow ML 1.44 szignatúrái alapján készítettük.

D.1. Structure Bool

```
type bool = bool

val not : bool -> bool

val toString   : bool -> string
val fromString : string -> bool option
val scan       : (char, 'a) StringCvt.reader -> (bool, 'a) StringCvt.reader

(*
   [not b] is the logical negation of b.

   [toString b] returns the string "false" or "true" according as b is
   false or true.

   [fromString s] scans a boolean b from the string s, after possible
   initial whitespace (blanks, tabs, newlines).  Returns (SOME b) if s
   has a prefix which is either "false" or "true"; the value b is the
   corresponding truth value; otherwise NONE is returned.

   [scan getc src] scans a boolean b from the stream src, using the
   stream accessor getc.  In case of success, returns SOME(b, rst)
   where b is the scanned boolean value and rst is the remainder of
   the stream; otherwise returns NONE.
*)
```

D.2. Structure Char

```
type char = char

val minChar : char
val maxChar : char
val maxOrd  : int

val chr     : int  -> char       (* may raise Chr *)
val ord     : char -> int
val succ    : char -> char       (* may raise Chr *)
val pred    : char -> char       (* may raise Chr *)

val isLower     : char -> bool   (* contains "abcdefghijklmnopqrstuvwxyz"  *)
val isUpper     : char -> bool   (* contains "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  *)
val isDigit     : char -> bool   (* contains "0123456789"                  *)
val isAlpha     : char -> bool   (* isUpper orelse isLower                 *)
val isHexDigit  : char -> bool   (* isDigit orelse contains "abcdefABCDEF" *)
val isAlphaNum  : char -> bool   (* isAlpha orelse isDigit                 *)
val isPrint     : char -> bool   (* any printable character (incl. #" ")   *)
val isSpace     : char -> bool   (* contains " \t\r\n\v\f"                 *)
val isPunct     : char -> bool   (* printable, not space or alphanumeric   *)
val isGraph     : char -> bool   (* (not isSpace) andalso isPrint          *)
val isAscii     : char -> bool   (* ord c < 128                            *)
val isCntrl     : char -> bool   (* control character                      *)

val toLower     : char -> char
val toUpper     : char -> char

val fromString  : string -> char option      (* ML escape sequences *)
val toString    : char -> string             (* ML escape sequences *)

val fromCString : string -> char option      (* C escape sequences  *)
val toCString   : char -> string             (* C escape sequences  *)

val contains    : string -> char -> bool
val notContains : string -> char -> bool

val <  : char * char -> bool
val <= : char * char -> bool
val >  : char * char -> bool
val >= : char * char -> bool
val compare : char * char -> order

(*
   [char] is the type of characters.

   [minChar] is the least character in the ordering <.

   [maxChar] is the greatest character in the ordering <.

   [maxOrd] is the greatest character code; equals ord(maxChar).
```

[chr i] returns the character whose code is i. Raises Chr if i<0 or i>maxOrd.

[ord c] returns the code of character c.

[succ c] returns the character immediately following c, or raises Chr if c = maxChar.

[pred c] returns the character immediately preceding c, or raises Chr if c = minChar.

[isLower c] returns true if c is a lowercase letter (a to z).

[isUpper c] returns true if c is a uppercase letter (A to Z).

[isDigit c] returns true if c is a decimal digit (0 to 9).

[isAlpha c] returns true if c is a letter (lowercase or uppercase).

[isHexDigit c] returns true if c is a hexadecimal digit (0 to 9 or a to f or A to F).

[isAlphaNum c] returns true if c is alphanumeric (a letter or a decimal digit).

[isPrint c] returns true if c is a printable character (space or visible).

[isSpace c] returns true if c is a whitespace character (blank, newline, tab, vertical tab, new page).

[isGraph c] returns true if c is a graphical character, that is, it is printable and not a whitespace character.

[isPunct c] returns true if c is a punctuation character, that is, graphical but not alphanumeric.

[isCntrl c] returns true if c is a control character, that is, if not (isPrint c).

[isAscii c] returns true if 0 <= ord c <= 127.

[toLower c] returns the lowercase letter corresponding to c, if c is a letter (a to z or A to Z); otherwise returns c.

[toUpper c] returns the uppercase letter corresponding to c, if c is a letter (a to z or A to Z); otherwise returns c.

[contains s c] returns true if character c occurs in the string s; false otherwise. The function, when applied to s, builds a table and returns a function which uses table lookup to decide whether a given character is in the string or not. Hence it is relatively

```
expensive to compute  val p = contains s  but very fast to compute
p(c) for any given character.
```

[notContains s c] returns true if character c does not occur in the
string s; false otherwise. Works by construction of a lookup table
in the same way as the above function.

[fromString s] attempts to scan a character or ML escape sequence
from the string s. Does not skip leading whitespace. For
instance, fromString "\\065" equals #"A".

[toString c] returns a string consisting of the character c, if c
is printable, else an ML escape sequence corresponding to c. A
printable character is mapped to a one-character string; bell,
backspace, tab, newline, vertical tab, form feed, and carriage
return are mapped to the two-character strings "\\a", "\\b", "\\t",
"\\n", "\\v", "\\f", and "\\r"; other characters with code less
than 32 are mapped to three-character strings of the form "\\^Z",
and characters with codes 127 through 255 are mapped to
four-character strings of the form "\\ddd", where ddd are three
decimal digits representing the character code. For instance,

```
   toString #"A"      equals "A"
   toString #"\\"     equals "\\\\"
   toString #"\""     equals "\\\""
   toString (chr   0) equals "\\^@"
   toString (chr   1) equals "\\^A"
   toString (chr   6) equals "\\^F"
   toString (chr   7) equals "\\a"
   toString (chr   8) equals "\\b"
   toString (chr   9) equals "\\t"
   toString (chr  10) equals "\\n"
   toString (chr  11) equals "\\v"
   toString (chr  12) equals "\\f"
   toString (chr  13) equals "\\r"
   toString (chr  14) equals "\\^N"
   toString (chr 127) equals "\\127"
   toString (chr 128) equals "\\128"
```

[fromCString s] attempts to scan a character or C escape sequence
from the string s. Does not skip leading whitespace. For
instance, fromString "\\065" equals #"A".

[toCString c] returns a string consisting of the character c, if c
is printable, else an C escape sequence corresponding to c. A
printable character is mapped to a one-character string; bell,
backspace, tab, newline, vertical tab, form feed, and carriage
return are mapped to the two-character strings "\\a", "\\b", "\\t",
"\\n", "\\v", "\\f", and "\\r"; other characters are mapped to
four-character strings of the form "\\ooo", where ooo are three
octal digits representing the character code. For instance,

```
   toString #"A"      equals "A"
   toString #"A"      equals "A"
```

```
      toString #"\\"     equals "\\\\"
      toString #"\""     equals "\\\""
      toString (chr   0) equals "\\000"
      toString (chr   1) equals "\\001"
      toString (chr   6) equals "\\006"
      toString (chr   7) equals "\\a"
      toString (chr   8) equals "\\b"
      toString (chr   9) equals "\\t"
      toString (chr  10) equals "\\n"
      toString (chr  11) equals "\\v"
      toString (chr  12) equals "\\f"
      toString (chr  13) equals "\\r"
      toString (chr  14) equals "\\016"
      toString (chr 127) equals "\\177"
      toString (chr 128) equals "\\200"

   [<] compares character codes.  That is, c1 < c2 returns true
   if ord(c1) < ord(c2), and similarly for <=, >, >=.

   [compare(c1, c2)] returns LESS, EQUAL, or GREATER, according as c1 is
   precedes, equals, or follows c2 in the ordering Char.< .
*)
```

D.3. Structure General

```
eqtype char
type   exn
eqtype int
eqtype real
eqtype string
eqtype unit

datatype bool      = false | true
datatype 'a list   = nil | op :: of 'a * 'a list
datatype order     = LESS | EQUAL | GREATER
datatype 'a ref    = ref of 'a

exception Bind
exception Match
exception Interrupt

exception Subscript
exception Size
exception Fail of string

exception Overflow
exception Div
exception Domain

val =   : ''a * ''a -> bool
val <>  : ''a * ''a -> bool

(* Below, numtxt is int, Word.word, Word8.word, real, char, string: *)
val <   : numtxt * numtxt -> bool
val <=  : numtxt * numtxt -> bool
val >   : numtxt * numtxt -> bool
val >=  : numtxt * numtxt -> bool

(* Below, realint is int or real:                                    *)
val ~   : realint -> realint           (* raises Overflow            *)
val abs : realint -> realint           (* raises Overflow            *)

(* Below, num is int, Word.word, Word8.word, or real:                *)
val +   : num * num -> num             (* raises Overflow            *)
val -   : num * num -> num             (* raises Overflow            *)
val *   : num * num -> num             (* raises Overflow            *)
val /   : real * real -> real          (* raises Div, Overflow       *)

(* Below, wordint is int, Word.word or Word8.word:                   *)
val div : wordint * wordint -> wordint  (* raises Div, Overflow       *)
val mod : wordint * wordint -> wordint  (* raises Div                 *)

val real   : int -> real               (* equals Real.fromInt        *)
val floor  : real -> int               (* round towards minus infinity *)
val ceil   : real -> int               (* round towards plus infinity  *)
```

```
val trunc  : real -> int              (* round towards zero          *)
val round  : real -> int              (* round to nearest even       *)

val o      : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
val ignore : 'a -> unit
val before : 'a * 'b -> 'a

val !      : 'a ref -> 'a
val :=     : 'a ref * 'a -> unit

val vector : 'a list -> 'a vector

(* Non-standard types and exceptions *)

datatype 'a frag = QUOTE of string | ANTIQUOTE of 'a

exception Io of function : string, name : string, cause : exn
exception Graphic_failure of string
exception Out_of_memory
```

D.4. Structure Int

```
type int = int

val precision : int option
val minInt    : int option
val maxInt    : int option

val ~    : int -> int                (* Overflow      *)
val *    : int * int -> int          (* Overflow      *)
val div  : int * int -> int          (* Div, Overflow *)
val mod  : int * int -> int          (* Div           *)
val quot : int * int -> int          (* Div, Overflow *)
val rem  : int * int -> int          (* Div           *)
val +    : int * int -> int          (* Overflow      *)
val -    : int * int -> int          (* Overflow      *)
val >    : int * int -> bool
val >=   : int * int -> bool
val <    : int * int -> bool
val <=   : int * int -> bool
val abs  : int -> int                (* Overflow      *)
val min  : int * int -> int
val max  : int * int -> int

val sign     : int -> int
val sameSign : int * int -> bool
val compare  : int * int -> order

val toInt     : int -> int
val fromInt   : int -> int
val toLarge   : int -> int
val fromLarge : int -> int

val toString   : int -> string
val fromString : string -> int option   (* Overflow      *)

val scan : StringCvt.radix
           -> (char, 'a) StringCvt.reader -> (int, 'a) StringCvt.reader
val fmt  : StringCvt.radix -> int -> string

(*
   [precision] is SOME n, where n is the number of significant bits in an
   integer.  In Moscow ML n is 31 in 32-bit architectures and 63 in 64-bit
   architectures.

   [minInt] is SOME n, where n is the most negative integer.

   [maxInt] is SOME n, where n is the most positive integer.

   [~, *, div, mod, +, -, abs] are the usual operations on integers.
   They raise Overflow if the result is not representable.  If q = i
   div d and r = i mod d then it holds that qd + r = i, where either 0
```

<= r < d or d < r <= 0. Evaluating i div 0 or i mod 0 reaises Oevrflow. In other words, div rounds towards minus infinity. Note that i div ~1 raises Overflow if i is the most negative integer, whereas i mod ~1 is 0.

[quot(i, d)] is the quotient of i by d, rounding towards zero (instead of rounding towards minus infinity, as done by div). Evaluating quot(i, 0) raises Div. Evaluating quot(i, ~1) raises Overflow if i is the most negative integer.

[rem(i, d)] is the remainder for quot. That is, if q = quot(i, d) and r = rem(i, d) then d * q + r = i where either 0 <= r < d or d < r <= 0. If made infix, the recommended fixity for quot and rem is

```
     infix 7 quot rem
```

[min(x, y)] is the smaller of x and y.

[max(x, y)] is the larger of x and y.

[sign x] is ~1, 0, or 1, according as x is negative, zero, or positive.

[>, >=, <, <=] are the usual comparisons on integers.

[compare(x, y)] returns LESS, EQUAL, or GREATER, according as x is less than, equal to, or greater than y.

[sameSign(x, y)] is true iff sign x = sign y.

[toInt x] is x (because this is the default int type in Moscow ML).

[fromInt x] is x (because this is the default int type in Moscow ML).

[toLarge x] is x (because this is the largest int type in Moscow ML).

[fromLarge x] is x (because this is the largest int type in Moscow ML).

[fmt radix i] returns a string representing i, in the radix (base) specified by radix.

```
  radix    description                      output format
  ------------------------------------------------------
   BIN     signed binary      (base  2)    ~?[01]+
   OCT     signed octal       (base  8)    ~?[0-7]+
   DEC     signed decimal     (base 10)    ~?[0-9]+
   HEX     signed hexadecimal (base 16)    ~?[0-9A-F]+
```

[toString i] returns a string representing i in signed decimal format. Equivalent to (fmt DEC i).

[fromString s] returns SOME(i) if a decimal integer numeral can be scanned from a prefix of string s, ignoring any initial whitespace;

```
   returns NONE otherwise.  A decimal integer numeral must have form,
   after possible initial whitespace:
        [+~-]?[0-9]+

   [scan radix getc charsrc] attempts to scan an integer numeral
   from the character source charsrc, using the accessor getc, and
   ignoring any initial whitespace.  The radix argument specifies the base
   of the numeral (BIN, OCT, DEC, HEX).  If successful, it returns
   SOME(i, rest) where i is the value of the number scanned, and rest
   is the unused part of the character source.  A numeral must have
   form, after possible initial whitespace:

     radix    input format
     ---------------------------
      BIN     [+~-]?[0-1]+
      OCT     [+~-]?[0-7]+
      DEC     [+~-]?[0-9]+
      HEX     [+~-]?[0-9a-fA-F]+
*)
```

D.5. Structure List

```
type 'a list = 'a list

exception Empty  (* Subscript and Size *)

val null       : 'a list -> bool
val hd         : 'a list -> 'a                          (* Empty     *)
val tl         : 'a list -> 'a list                     (* Empty     *)
val last       : 'a list -> 'a                          (* Empty     *)

val nth        : 'a list * int -> 'a                    (* Subscript *)
val take       : 'a list * int -> 'a list               (* Subscript *)
val drop       : 'a list * int -> 'a list               (* Subscript *)

val length     : 'a list -> int

val rev        : 'a list -> 'a list

val @          : 'a list * 'a list -> 'a list
val concat     : 'a list list -> 'a list
val revAppend  : 'a list * 'a list -> 'a list

val app        : ('a -> unit) -> 'a list -> unit
val map        : ('a -> 'b) -> 'a list -> 'b list
val mapPartial : ('a -> 'b option) -> 'a list -> 'b list

val find       : ('a -> bool) -> 'a list -> 'a option
val filter     : ('a -> bool) -> 'a list -> 'a list
val partition  : ('a -> bool ) -> 'a list -> ('a list * 'a list)

val foldr      : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
val foldl      : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

val exists     : ('a -> bool) -> 'a list -> bool
val all        : ('a -> bool) -> 'a list -> bool

val tabulate   : int * (int -> 'a) -> 'a list           (* Size      *)

val getItem    : 'a list -> ('a * 'a list) option

(*
   [null xs] is true iff xs is nil.

   [hd xs] returns the first element of xs.  Raises Empty if xs is nil.

   [tl xs] returns all but the first element of xs.
   Raises Empty if xs is nil.

   [last xs] returns the last element of xs.  Raises Empty if xs is nil.

   [nth(xs, i)] returns the i'th element of xs, counting from 0.
```

Raises Subscript if i<0 or i>=length xs.

[take(xs, i)] returns the first i elements of xs. Raises Subscript if i<0 or i>length xs.

[drop(xs, i)] returns what is left after dropping the first i elements of xs. Raises Subscript if i<0 or i>length xs.
It holds that take(xs, i) @ drop(xs, i) = xs when 0 <= i <= length xs.

[length xs] returns the number of elements in xs.

[rev xs] returns the list of xs's elements, reversed.

[xs @ ys] returns the list which is the concatenation of xs and ys.

[concat xss] returns the list which is the concatenation of all the lists in xss.

[revAppend(xs, ys)] is equivalent to rev xs @ ys, but more efficient.

[app f xs] applies f to the elements of xs, from left to right.

[map f xs] applies f to each element x of xs, from left to right, and returns the list of f's results.

[mapPartial f xs] applies f to each element x of xs, from left to right, and returns the list of those y's for which f(x) evaluated to SOME y.

[find p xs] applies f to each element x of xs, from left to right until p(x) evaluates to true; returns SOME x if such an x exists otherwise NONE.

[filter p xs] applies p to each element x of xs, from left to right, and returns the sublist of those x for which p(x) evaluated to true.

[partition p xs] applies p to each element x of xs, from left to right, and returns a pair (pos, neg) where pos is the sublist of those x for which p(x) evaluated to true, and neg is the sublist of those for which p(x) evaluated to false.

[foldr op% e xs] evaluates x1 % (x2 % (... % (x(n-1) % (xn % e)) ...)) where xs = [x1, x2, ..., x(n-1), xn], and % is taken to be infixed.

[foldl op% e xs] evaluates xn % (x(n-1) % (... % (x2 % (x1 % e)))) where xs = [x1, x2, ..., x(n-1), xn], and % is taken to be infixed.

[exists p xs] applies p to each element x of xs, from left to right until p(x) evaluates to true; returns true if such an x exists, otherwise false.

```
   [all p xs] applies p to each element x of xs, from left to
   right until p(x) evaluates to false; returns false if such an x
   exists, otherwise true.

   [tabulate(n, f)] returns a list of length n whose elements are
   f(0), f(1), ..., f(n-1), created from left to right.  Raises Size
   if n<0.

   [getItem xs] attempts to extract an element from the list xs.  It
   returns NONE if xs is empty, and returns SOME (x, xr) if xs=x::xr.
   This can be used for scanning booleans, integers, reals, and so on
   from a list of characters.  For instance, to scan a decimal integer
   from a list cs of characters, compute
        Int.scan StringCvt.DEC List.getItem cs
*)
```

D.6. Structure Listsort

```
val sort   : ('a * 'a -> order) -> 'a list -> 'a list
val sorted : ('a * 'a -> order) -> 'a list -> bool

(*
   [sort ordr xs] sorts the list xs in nondecreasing order, using the
   given ordering.  Uses Richard O'Keefe's smooth applicative merge
   sort.

   [sorted ordr xs] checks that the list xs is sorted in nondecreasing
   order, in the given ordering.
*)
```

D.7. Structure Math

```
type real = real

val pi : real
val e  : real

val sqrt  : real -> real
val sin   : real -> real
val cos   : real -> real
val tan   : real -> real
val atan  : real -> real
val asin  : real -> real
val acos  : real -> real
val atan2 : real * real -> real
val exp   : real -> real
val pow   : real * real -> real
val ln    : real -> real
val log10 : real -> real
val sinh  : real -> real
val cosh  : real -> real
val tanh  : real -> real

(*
   [pi] is the circumference of the circle with diameter 1:
   3.14159265358979323846.

   [e] is the base of the natural logarithm: 2.7182818284590452354.

   [sqrt x] is the square root of x.  Raises Domain is x < 0.0.

   [sin r] is the sine of r, where r is in radians.

   [cos r] is the cosine of r, where r is in radians.

   [tan r] is the tangent of r, where r is in radians.  Raises Domain if
   r is a multiple of pi/2.

   [atan t] is the arc tangent of t, in the open interval ] ~pi/2, pi/2 [.

   [asin t] is the arc sine of t, in the closed interval [ ~pi/2, pi/2 ].
   Raises Domain if abs x > 1.

   [acos t] is the arc cosine of t, in the closed interval [ 0, pi ].
   Raises Domain if abs x > 1.

   [atan2(y, x)] is the arc tangent of y/x, in the interval ] ~pi, pi ],
   except that atan2(y, 0) = sign y * pi/2.  The quadrant of the result
   is the same as the quadrant of the point (x, y).
   Hence sign(cos(atan2(y, x))) = sign x
   and   sign(sin(atan2(y, x))) = sign y.
```

```
   [exp x] is e to the x'th power.

   [pow (x, y)] is x it the y'th power, defined when
      y >= 0 and (y integral or x >= 0)
   or y < 0 and ((y integral and x <> 0.0) or x > 0).

   We define pow(0, 0) = 1.

   [ln x] is the natural logarithm of x (that is, with base e).
   Raises Domain if x <= 0.0.

   [log10 x] is the base-10 logarithm of x.  Raises Domain if x <= 0.0.

   [sinh x] returns the hyperbolic sine of x, mathematically defined as
   (exp x - exp (~x)) / 2.  Raises Overflow if x is too large.

   [cosh x] returns the hyperbolic cosine of x, mathematically defined as
   (exp x + exp (~x)) / 2.  Raises Overflow if x is too large.

   [tanh x] returns the hyperbolic tangent of x, mathematically defined
   as (sinh x) / (cosh x).  Raises Domain if x is too large.
*)
```

D.8. Structure Meta

```
(* Functions for use in an interactive Moscow ML session *)

val printVal    : 'a -> 'a
val printDepth  : int ref
val printLength : int ref
val installPP   : (ppstream -> 'a -> unit) -> unit

val use         : string -> unit
val compile     : string -> unit
val load        : string -> unit
val loadOne     : string -> unit
val loaded      : unit -> string list
val loadPath    : string list ref

val quietdec    : bool ref
val verbose     : bool ref

val exnName     : exn -> string
val exnMessage  : exn -> string

val quotation   : bool ref
val valuepoly   : bool ref

val quit        : unit -> 'a
val system      : string -> int

(*
   [printVal e] prints the value of expression e to standard output
   exactly as it would be printed at top-level, and returns the value
   of e.  Output is flushed immediately.  This function is provided as
   a simple debugging aid.  The effect of printVal is similar to that
   of `print' in Edinburgh ML or Umeaa ML.  For string arguments, the
   effect of SML/NJ print can be achieved by the function
   TextIO.print : string -> unit.

   [printDepth] determines the depth (in terms of nested constructors,
   records, tuples, lists, and vectors) to which values are printed by
   the top-level value printer and the function printVal.  The components
   of the value whose depth is greater than printDepth are printed as
   `#'.  The initial value of printDepth is 20. This value can be
   changed at any moment, by evaluating, for example,
        printDepth := 17;

   [printLength] determines the way in which list values are printed
   by the top-level value printer and the function printVal.  If the
   length of a list is greater than printLength, then only the first
   printLength elements are printed, and the remaining elements are
   printed as `...'.  The initial value of printLength is 200.  This
   value can be changed at any moment, by evaluating, for example,
        printLength := 500;
```

[quit ()] quits Moscow ML immediately.

[installPP pp] installs the prettyprinter pp at type ty, provided pp has type ppstream -> ty -> unit. The type ty must be a nullary (parameter-less) type constructor representing a datatype, either built-in (such as bool) or user-defined. Whenever a value of type ty is about to be printed by the interactive system, or function printVal is invoked on an argument of type ty, the pretty-printer pp will be invoked to print it. See library unit PP for more information.

[use "f"] causes ML declarations to be read from file f as if they were entered from the console. A file loaded by use may, in turn, evaluate calls to use. For best results, use ‘use’ only at top level, or at top level within a use’d file.

[compile "U.sig"] will compile and elaborate the unit signature in file U.sig, producing a compiled signature file U.ui. During compilation, the compiled signatures of other units will be accessed if they are mentioned in U.sig.

[compile "U.sml"] will elaborate and compile the unit body in file U.sml, producing a bytecode file U.uo. If there is an explicit signature U.sig, then file U.ui must exist, and the unit body must match the signature. If there is no U.sig, then an inferred signature file U.ui will be produced also. No evaluation takes place. During compilation, the compiled signatures of other units will be accessed if they are mentioned in U.sml.

The declared identifiers will be reported if verbose is true (see below); otherwise compilation will be silent. In any case, compilation warnings are reported, and compilation errors abort the compilation and raise the exception Fail with a string argument.

[load "U"] will load and evaluate the compiled unit body from file U.uo. The resulting values are not reported, but exceptions are reported, and cause evaluation and loading to stop. If U is already loaded, then load "U" has no effect. If any other unit is mentioned by U but not yet loaded, then it will be loaded automatically before U.

After loading a unit, it can be opened with ‘open U’. Opening it at top-level will list the identifiers declared in the unit.

When loading U, it is checked that the signatures of units mentioned by U agree with the signatures used when compiling U, and it is checked that the signature of U has not been modified since U was compiled; these checks are necessary for type safety. The exception Fail is raised if these signature checks fail, or if the file containing U or a unit mentioned by U does not exist.

[loadOne "U"] is similar to 'load "U"', but raises exception Fail if U is already loaded or if some unit mentioned by U is not yet loaded. That is, it does not automatically load any units mentioned by U. It performs the same signature checks as 'load'.

[loaded ()] returns a list of the names of all compiled units that have been loaded so far. The names appear in some random order.

[loadPath] determines the load path: which directories will be searched for interface files (.ui files), bytecode files (.uo files), and source files (.sml files). This variable affects the load, loadOne, and use functions. The current directory is always searched first, followed by the directories in loadPath, in order. By default, only the standard library directory is in the list, but if additional directories are specified using option -I, then these directories are prepended to loadPath.

[quietdec] when \tt true, turns off the interactive system's prompt and responses, except warnings and error messages. Useful for writing scripts in SML. The default value is false; can be set to true with the -quietdec command line option.

[verbose] determines whether the signature inferred by a call to compile will be printed. The printed signature follows the syntax of Moscow ML signatures, so the output of compile "U.sml" can be edited to subsequently create file U.sig. The default value is ref false.

[exnName exn] returns a name for the exception constructor in exn. Never raises an exception itself. The name returned may be that of any exception constructor aliasing with exn. For instance,

```
     let exception E1; exception E2 = E1 in exnName E2 end
```

may evaluate to "E1" or "E2".

[exnMessage exn] formats and returns a message corresponding to exception exn. For the exceptions defined in the SML Basis Library, the message will include the argument carried by the exception.

[quotation] determines whether quotations and antiquotations are permitted in declarations entered at top-level and in files compiled with compile. A quotation is a piece of text surrounded by backquote characters 'a b c' and is used to embed object language phrases in ML programs; see the Moscow ML Owner's Manual for a brief explanation of quotations. When quotation is false, the backquote character is an ordinary symbol which can be used in ML symbolic identifiers. When quotation is \tt true, the backquote character is illegal in symbolic identifiers, and a quotation 'a b c' will be recognized by the parser and evaluated to an object of type 'a General.frag list. The default value is ref false.

[valuepoly] determines whether the type checker should use 'value

```
   polymorphism', making no distinction between imperative ('_a) and
   applicative ('a) type variables, and generalizing type variables
   only in non-expansive expressions.  An expression is non-expansive
   if it is a variable, a special constant, a function, a tuple or
   record of non-expansive expressions, a parenthesized or typed
   non-expansive expression, or the application of an exception or value
   constructor (other than ref) to a non-expansive expression.
        If valuepoly is false, then the type checker will distinguish
   imperative and applicative type variables, generalize all applicative
   type variables, and generalize imperative type variables only in
   non-expansive expressions.  This is the default, required by the
   1990 Definition of Standard ML, Section 4.8.

   [system "com"] causes the command com to be executed by the
   operating system.  If a non-zero integer is returned, this must
   indicate that the operating system has failed to execute the
   command.  Under MS DOS, the integer returned tends to always equal
   zero, even when the command fails.
*)
```

D.9. Structure Option

```
exception Option

val getOpt         : 'a option * 'a -> 'a
val isSome         : 'a option -> bool
val valOf          : 'a option -> 'a
val filter         : ('a -> bool) -> 'a -> 'a option
val map            : ('a -> 'b) -> 'a option -> 'b option
val app            : ('a -> unit) -> 'a option -> unit
val join           : 'a option option -> 'a option
val compose        : ('a -> 'b) * ('c -> 'a option) -> ('c -> 'b option)
val mapPartial     : ('a -> 'b option) -> ('a option -> 'b option)
val composePartial : ('a -> 'b option) * ('c -> 'a option) -> ('c -> 'b option)

(*
   [getOpt (xopt, d)] returns x if xopt is SOME x; returns d otherwise.

   [isSome vopt] returns true if xopt is SOME x; returns false otherwise.

   [valOf vopt] returns x if xopt is SOME x; raises Option otherwise.

   [filter p x] returns SOME x if p x is true; returns NONE otherwise.

   [map f xopt] returns SOME (f x) if xopt is SOME x; returns NONE otherwise.

   [app f xopt] applies f to x if xopt is SOME x; does nothing otherwise.

   [join xopt] returns x if xopt is SOME x; returns NONE otherwise.

   [compose (f, g) x] returns SOME (f y) if g x is SOME y; returns NONE
   otherwise.  It holds that compose (f, g) = map f o g.

   [mapPartial f xopt] returns f x if xopt is SOME x; returns NONE otherwise.
   It holds that mapPartial f = join o map f.

   [composePartial (f, g) x] returns f y if g x is SOME y; returns NONE
   otherwise.  It holds that composePartial (f, g) = mapPartial f o g.

   The operators (map, join, SOME) form a monad.
*)
```

D.10. Structure Real

```
type real = real

exception Div
and Overflow

val ~    : real -> real
val +    : real * real -> real
val -    : real * real -> real
val *    : real * real -> real
val /    : real * real -> real
val abs  : real -> real
val min  : real * real -> real
val max  : real * real -> real
val sign : real -> int
val compare : real * real -> order

val sameSign    : real * real -> bool
val toDefault   : real -> real
val fromDefault : real -> real
val fromInt     : int -> real

val floor : real -> int
val ceil  : real -> int
val trunc : real -> int
val round : real -> int

val >     : real * real -> bool
val >=    : real * real -> bool
val <     : real * real -> bool
val <=    : real * real -> bool
val ==    : real * real -> bool
val !=    : real * real -> bool
val ?=    : real * real -> bool

val toString   : real -> string
val fromString : string -> real option
val scan       : (char, 'a) StringCvt.reader -> (real, 'a) StringCvt.reader
val fmt        : StringCvt.realfmt -> real -> string

(*
   [~, *, /, +, -, >, >=, <, <=, abs] are the usual operations on reals.

   [min(x, y)] is the smaller of x and y.

   [max(x, y)] is the larger of x and y.

   [sign x] is ~1, 0, or 1, according as x is negative, zero, or positive.

   [compare(x, y)] returns LESS, EQUAL, or GREATER, according
   as x is less than, equal to, or greater than y.
```

[sameSign(x, y)] is true iff sign x = sign y.

[toDefault x] is x.

[fromDefault x] is x.

[fromInt i] is the floating-point number representing integer i.

[floor r] is the largest integer <= r (rounds towards minus infinity). May raise Overflow.

[ceil r] is the smallest integer >= r (rounds towards plus infinity). May raise Overflow.

[trunc r] is numerically largest integer between r and zero (rounds towards zero). May raise Overflow.

[round r] is the integer nearest to r, using the default rounding mode. NOTE: This isn't the required behaviour: it should round to nearest even integer in case of a tie. May raise Overflow.

[==(x, y)] is equivalent to x=y in Moscow ML (because of the absence of NaNs and Infs).

[!=(x, y)] is equivalent to x<>y in Moscow ML (because of the absence of NaNs and Infs).

[?=(x, y)] is false in Moscow ML (because of the absence of NaNs and Infs).

[fmt spec r] returns a string representing r, in the format specified by spec (see below). The requested number of digits must be >= 0 in the SCI and FIX formats and > 0 in the GEN format; otherwise Size is raised, even in a partial application fmt(spec).

```
spec          description                          C printf
-------------------------------------------------------------
SCI NONE      scientific,   6 digits after point      %e
SCI (SOME n)  scientific,   n digits after point      %.ne
FIX NONE      fixed-point,  6 digits after point      %f
FIX (SOME n)  fixed-point,  n digits after point      %.nf
GEN NONE      auto choice, 12 significant digits      %.12g
GEN (SOME n)  auto choice,  n significant digits      %.ng
```

[toString r] returns a string representing r, with automatic choice of format according to the magnitude of r.
Equivalent to (fmt (GEN NONE) r).

[fromString s] returns SOME(r) if a floating-point numeral can be scanned from a prefix of string s, ignoring any initial whitespace; returns NONE otherwise. The valid forms of floating-point numerals

```
   are described by:
        [+~-]?(([0-9]+(\.[0-9]+)?)|(\.[0-9]+))([eE][+~-]?[0-9]+)?

   [scan getc charsrc] attempts to scan a floating-point number from
   the character source charsrc, using the accessor getc, and ignoring
   any initial whitespace.  If successful, it returns SOME(r, rest)
   where r is the number scanned, and rest is the unused part of the
   character source.  The valid forms of floating-point numerals
   are described by:
        [+~-]?(([0-9]+(\.[0-9]+)?)|(\.[0-9]+))([eE][+~-]?[0-9]+)?
*)
```

D.11. Structure String

```
local

type char = Char.char

in

type string = string
val maxSize   : int
val size      : string -> int
val sub       : string * int -> char
val substring : string * int * int -> string
val extract   : string * int * int option -> string
val concat    : string list -> string
val ^         : string * string -> string
val str       : char -> string
val implode   : char list -> string
val explode   : string -> char list

val map       : (char -> char) -> string -> string
val translate : (char -> string) -> string -> string
val tokens    : (char -> bool) -> string -> string list
val fields    : (char -> bool) -> string -> string list
val isPrefix  : string -> string -> bool

val compare   : string * string -> order
val collate   : (char * char -> order) -> string * string -> order

val fromString  : string -> string option     (* ML escape sequences *)
val toString    : string -> string            (* ML escape sequences *)
val fromCString : string -> string option     (* C escape sequences *)
val toCString   : string -> string            (* C escape sequences *)

val <  : string * string -> bool
val <= : string * string -> bool
val >  : string * string -> bool
val >= : string * string -> bool

end

(*
   [string] is the type of strings of characters.

   [maxSize] is the maximal number of characters in a string.

   [size s] is the number of characters in string s.

   [sub(s, i)] is the i'th character of s, counting from zero.
   Raises Subscript if i<0 or i>=size s.

   [substring(s, i, n)] is the string s[i..i+n-1].  Raises Subscript
```

```
if i<0 or n<0 or i+n>size s.  Equivalent to extract(s, i, SOME n).

[extract (s, i, NONE)] is the string s[i..size s-1].
Raises Subscript if i<0 or i>size s.

[extract (s, i, SOME n)] is the string s[i..i+n-1].
Raises Subscript if i<0 or n<0 or i+n>size s.

[concat ss] is the concatenation of all the strings in ss.
Raises Size if the sum of their sizes is greater than maxSize.

[s1 ^ s2] is the concatenation of strings s1 and s2.

[str c] is the string of size one which contains the character c.

[implode cs] is the string containing the characters in the list cs.
Equivalent to  concat (List.map str cs).

[explode s] is the list of characters in the string s.

[map f s] applies f to every character of s, from left to right,
and returns the string consisting of the resulting characters.
Equivalent to  CharVector.map f s
       and to  implode (List.map f (explode s)).

[translate f s] applies f to every character of s, from left to
right, and returns the concatenation of the resulting strings.
Raises Size if the sum of their sizes is greater than maxSize.
Equivalent to concat (List.map f (explode s)).

[tokens p s] returns the list of tokens in s, from left to right,
where a token is a non-empty maximal substring of s not containing
any delimiter, and a delimiter is a character satisfying p.

[fields p s] returns the list of fields in s, from left to right,
where a field is a (possibly empty) maximal substring of s not
containing any delimiter, and a delimiter is a character satisfying p.

Two tokens may be separated by more than one delimiter, whereas two
fields are separated by exactly one delimiter.  If the only delimiter
is the character #"|", then
     "abc||def" contains two tokens:   "abc" and "def"
     "abc||def" contains three fields: "abc" and "" and "def"

[isPrefix s1 s2] is true if s1 is a prefix of s2.
That is, if there exists a string t such that s1 ^ t = s2.

[compare (s1, s2)] does lexicographic comparison, using the
standard ordering Char.compare on the characters.  Returns LESS,
EQUAL, or GREATER, according as s1 is less than, equal to, or
greater than s2.
```

```
   [collate cmp (s1, s2)] performs lexicographic comparison, using the
   given ordering cmp on characters.

   [fromString s] scans the string s as an ML source program string,
   converting escape sequences into the appropriate characters.  Does
   not skip leading whitespace.

   [toString s] returns a string corresponding to s, with
   non-printable characters replaced by ML escape sequences.
   Equivalent to String.translate Char.toString.

   [fromCString s] scans the string s as a C source program string,
   converting escape sequences into the appropriate characters.  Does
   not skip leading whitespace.

   [toCString s] returns a string corresponding to s, with
   non-printable characters replaced by C escape sequences.
   Equivalent to String.translate Char.toCString.

   [<], [<=], [>], and [>=] compare strings lexicographically.
*)
```

D.12. Structure TextIO

```
type elem   = Char.char
type vector = string

(* Text input: *)

type instream

val openIn       : string -> instream
val closeIn      : instream -> unit
val input        : instream -> vector
val inputAll     : instream -> vector
val inputNoBlock : instream -> vector option
val input1       : instream -> elem option
val inputN       : instream * int -> vector
val inputLine    : instream -> string
val endOfStream  : instream -> bool
val lookahead    : instream -> elem option

type cs (* character source state *)

val scanStream   : ((char, cs) StringCvt.reader -> ('a, cs) StringCvt.reader)
                   -> instream -> 'a option

val stdIn        : instream

(* Text output: *)

type outstream

val openOut      : string -> outstream
val openAppend   : string -> outstream
val closeOut     : outstream -> unit
val output       : outstream * vector -> unit
val output1      : outstream * elem -> unit
val outputSubstr : outstream * substring -> unit
val flushOut     : outstream -> unit

val stdOut       : outstream
val stdErr       : outstream

val print        : string -> unit

(* This structure provides input/output functions on text streams.
   The functions are state-based: reading from or writing to a stream
   changes the state of the stream.  The streams are buffered: output
   to a stream may not immediately affect the underlying file or
   device.

   Note that under DOS, Windows, OS/2, and MacOS, text streams will be
   `translated' by converting (e.g.) the double newline CRLF to a
```

single newline character \n.

[instream] is the type of state-based characters input streams, and type outstream is the type of state-based character output streams.

[elem] is the type char of characters, and type vector is the type of character vectors (strings).

TEXT INPUT:

[openIn s] creates a new instream associated with the file named s. Raises Io.Io is file s does not exist or is not accessible.

[closeIn istr] closes stream istr. Has no effect if istr is closed already. Further operations on istr will behave as if istr is at end of stream (that is, will return "" or NONE or true).

[input istr] reads some elements from istr, returning a vector v of those elements. The vector will be empty (size v = 0) if and only if istr is at end of stream or is closed. May block (not return until data are available in the external world).

[inputAll istr] reads and returns the string v of all characters remaining in istr up to end of stream.

[inputNoBlock istr] returns SOME(v) if some elements v can be read without blocking; returns SOME("") if it can be determined without blocking that istr is at end of stream; returns NONE otherwise. If istr does not support non-blocking input, raises Io.NonblockingNotSupported.

[input1 istr] returns SOME(e) if at least one element e of istr is available; returns NONE if istr is at end of stream or is closed; blocks if necessary until one of these conditions holds.

[inputN(istr, n)] returns the next n characters from istr as a string, if that many are available; returns all remaining characters if end of stream is reached before n characters are available; blocks if necessary until one of these conditions holds. (This is the behaviour of the 'input' function prescribed in the 1990 Definition of Standard ML).

[inputLine istr] returns one line of text, including the terminating newline character. If end of stream is reached before a newline character, then the remaining part of the stream is returned, with a newline character added. If istr is at end of stream or is closed, then the empty string "" is returned.

[endOfStream istr] returns false if any elements are available in istr; returns true if istr is at end of stream or closed; blocks if necessary until one of these conditions holds.

[lookahead istr] returns SOME(e) where e is the next element in the stream; returns NONE if istr is at end of stream or is closed; blocks if necessary until one of these conditions holds. Does not advance the stream.

[stdIn] is the buffered state-based standard input stream.

[scanStream scan istr] turns the instream istr into a character source and applies the scanner 'scan' to that source. See StringCvt for more on character sources and scanners. The Moscow ML implementation currently can backtrack only 512 characters, and raises Fail if the scanner backtracks further than that.

TEXT OUTPUT:

[openOut s] creates a new outstream associated with the file named s. If file s does not exist, and the directory exists and is writable, then a new file is created. If file s exists, it is truncated (any existing contents are lost).

[openAppend s] creates a new outstream associated with the file named s. If file s does not exist, and the directory exists and is writable, then a new file is created. If file s exists, any existing contents are retained, and output goes at the end of the file.

[closeOut ostr] closes stream ostr; further operations on ostr (except for additional close operations) will raise exception Io.Io.

[output(ostr, v)] writes the string v on outstream ostr.

[output1(ostr, e)] writes the character e on outstream ostr.

[flushOut ostr] flushes the outstream ostr, so that all data written to ostr becomes available to the underlying file or device.

[stdOut] is the buffered state-based standard output stream.

[stdErr] is the unbuffered state-based standard error stream. That is, it is always kept flushed, so flushOut(stdErr) is redundant.

The functions below are not yet implemented:

[setPosIn(istr, i)] sets istr to the (untranslated) position i. Raises Io.Io if not supported on istr.

[getPosIn istr] returns the (untranslated) current position of istr. Raises Io.Io if not supported on istr.

[endPosIn istr] returns the (untranslated) last position of istr. Because of translation, one cannot expect to read

```
     endPosIn istr - getPosIn istr
   from the current position.

   [getPosOut ostr] returns the current position in stream ostr.
   Raises Io.Io if not supported on ostr.

   [endPosOut ostr] returns the ending position in stream ostr.
   Raises Io.Io if not supported on ostr.

   [setPosOut(ostr, i)] sets the current position in stream to ostr to
   i.  Raises Io.Io if not supported on ostr.

   [mkInstream sistr] creates a state-based instream from the
   functional instream sistr.

   [getInstream istr] returns the functional instream underlying the
   state-based instream istr.

   [setInstream(istr, sistr)] redirects istr, so that subsequent input
   is taken from the functional instream sistr.

   [mkOutstream sostr] creates a state-based outstream from the
   outstream sostr.

   [getOutstream ostr] returns the outstream underlying the
   state-based outstream ostr.

   [setOutstream(ostr, sostr)] redirects the outstream ostr so that
   subsequent output goes to sostr.

*)
```

D.13. Structure Time

```
eqtype time

exception Time

val zeroTime : time
val now      : unit -> time

val toSeconds        : time -> int
val toMilliseconds   : time -> int
val toMicroseconds   : time -> int
val fromSeconds      : int -> time
val fromMilliseconds : int -> time
val fromMicroseconds : int -> time

val fromReal : real -> time
val toReal   : time -> real

val toString   : time -> string (* rounded to millisecond precision *)
val fmt        : int -> time -> string
val fromString : string -> time option
val scan       : (char, 'a) StringCvt.reader -> (time, 'a) StringCvt.reader

val +  : time * time -> time
val -  : time * time -> time
val <  : time * time -> bool
val <= : time * time -> bool
val >  : time * time -> bool
val >= : time * time -> bool

val compare : time * time -> order

(*
   [time] is the type of values representing durations as well as absolute
   points in time (which can be thought of as durations since some time zero).

   [zeroTime] represents the 0-second duration, and the origin of time,
   so zeroTime + t = t + zeroTime = t for all t.

   [now ()] returns the point in time at which the application occurs.

   [fromSeconds s] returns the time value corresponding to s seconds.
   Raises Time if s < 0.

   [fromMilliseconds ms] returns the time value corresponding to ms
   milliseconds.  Raises Time if ms < 0.

   [fromMicroseconds us] returns the time value corresponding to us
   microseconds.  Raises Time if us < 0.

   [toSeconds t] returns the number of seconds represented by t,
```

```
   truncated.  Raises Overflow if that number is not representable as
   an int.

   [toMilliseconds t] returns the number of milliseconds
   represented by t, truncated.  Raises Overflow if that number is not
   representable as an int.

   [toMicroseconds t] returns the number of microseconds
   represented by t, truncated.  Raises Overflow if t that number is
   not representable as an int.

   [realToTime r] converts a real to a time value representing that
   many seconds.  Raises Time if r < 0 or if r is not representable
   as a time value.  It holds that realToTime 0.0 = zeroTime.

   [timeToReal t] converts a time the number of seconds it represents;
   hence realToTime and timeToReal are inverses of each other when
   defined.  Raises Overflow if t is not representable as a real.

   [fmt n t] returns as a string the number of seconds represented by
   t, rounded to n decimal digits.  If n <= 0, then no decimal digits
   are reported.

   [toString t] returns as a string the number of seconds represented
   by t, rounded to 3 decimal digits.  Equivalent to (fmt 3 t).

   [fromString s] returns SOME t where t is the time value represented
   by the string s of form [\n\t ]*([0-9]+(\.[0-9]+)?)|(\.[0-9]+);
   or returns NONE if s cannot be parsed as a time value.

   [scan getc src], where getc is a character accessor, returns SOME
   (t, rest) where t is a time and rest is rest of the input, or NONE
   if s cannot be parsed as a time value.

   [t1 + t2] is the sum of the times t1 and t2.  For reals r1, r2 >= 0.0,
   realToTime r1 + realToTime r2 = realToTime(Real.+(r1,r2)).  Raises
   Overflow if the result is not representable as a time value.

   [t1 - t2] is the t1 minus t2, that is, the duration from t2 to t1.
   Raises Time if t1 < t2 or if the result is not representable as a
   time value.  It holds that t - zeroTime = t.

   [t1 < t2] asserts that t1 is strictly before t2.  Similarly for
   <=, >, >=.  It holds for reals r1, r2 >= 0.0 that
       realToTime r1 < realToTime r2  iff  Real.<(r1, r2)

   [compare(t1, t2)] returns LESS, EQUAL, or GREATER, according
   as t1 precedes, equals, or follows t2 in time.
*)
```

D.14. Structure Timer

```
local

type time = Time.time

in

type cpu_timer
type real_timer

val startCPUTimer : unit -> cpu_timer
val totalCPUTimer : unit -> cpu_timer
val checkCPUTimer : cpu_timer -> usr : time, sys : time, gc : time

val startRealTimer : unit -> real_timer
val totalRealTimer : unit -> real_timer
val checkRealTimer : real_timer -> time

end

(*
   [cpu_timer] is the type of values measuring the CPU time consumed.

   [real_timer] is the type of values measuring the real time that has passed.

   [startCPUTimer ()] returns a cpu_timer started at the moment of
   the call.

   [totalCPUTimer ()] returns a cpu_timer started at the moment the
   library was loaded.

   [checkCPUTimer tmr] returns usr, sys, gc where usr is the amount
   of user CPU time consumed since tmr was started, gc is the amount
   of user CPU time spent on garbage collection, and sys is the
   amount of system CPU time consumed since tmr was started.  Note
   that gc time is included in the usr time.  Under MS DOS, usr time
   and gc time are measured in real time.

   [startRealTimer ()] returns a real_timer started at the moment of
   the call.

   [totalRealTimer ()] returns a real_timer started at the moment the
   library was loaded.

   [checkRealTimer tmr] returns the amount of real time that has passed
   since tmr was started.
*)
```

D.15. Structure Word

```
type word = word
val wordSize : int

val orb  : word * word -> word
val andb : word * word -> word
val xorb : word * word -> word
val notb : word -> word

val <<  : word * word -> word
val >>  : word * word -> word
val ~>> : word * word -> word

val +   : word * word -> word
val -   : word * word -> word
val *   : word * word -> word
val div : word * word -> word
val mod : word * word -> word

val >   : word * word -> bool
val <   : word * word -> bool
val >=  : word * word -> bool
val <=  : word * word -> bool
val compare : word * word -> order

val min : word * word -> word
val max : word * word -> word

val toString   : word -> string
val fromString : string -> word option
val scan : StringCvt.radix
           -> (char, 'a) StringCvt.reader -> (word, 'a) StringCvt.reader
val fmt  : StringCvt.radix -> word -> string

val toLargeWord   : word -> word
val toLargeWordX  : word -> word        (* with sign extension *)
val fromLargeWord : word -> word

val toLargeInt   : word -> int
val toLargeIntX  : word -> int          (* with sign extension *)
val fromLargeInt : int -> word

val toInt   : word -> int
val toIntX  : word -> int               (* with sign extension *)
val fromInt : int -> word

(*
   [word] is the type of n-bit words, or n-bit unsigned integers.

   [wordSize] is the value of n above.  In Moscow ML, n=31 on 32-bit
   machines and n=63 on 64-bit machines.
```

[orb(w1, w2)] returns the bitwise ‘or’ of w1 and w2.

[andb(w1, w2)] returns the bitwise ‘and’ of w1 and w2.

[xorb(w1, w2)] returns the bitwise ‘exclusive or’ or w1 and w2.

[notb w] returns the bitwise negation of w.

[<<(w, k)] returns the word resulting from shifting w left by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

[>>(w, k)] returns the word resulting from shifting w right by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

[~>>(w, k)] returns the word resulting from shifting w right by k bits. The bits shifted in are replications of the left-most bit: the ‘sign bit’, so this is an arithmetical shift. Consequently, for k >= wordSize and wordToInt w >= 0 the result is all 0-bits, and for k >= wordSize and wordToInt w < 0 the result is all 1-bits.

To make <<, >>, and ~>> infix, use the declaration

```
                       infix 5 << >> ~>>
```

*[+, -, *, div, mod]* represent unsigned integer addition, subtraction, multiplication, division, and remainder, modulus two to wordSize. The operations (i div j) and (i mod j) raise Div when j=0. Otherwise no exceptions are raised.

[w1 > w2] returns true if the unsigned integer represented by w1 is larger than that of w2, and similarly for <, >=, <=.

[compare(w1, w2)] returns LESS, EQUAL, or GREATER, according as w1 is less than, equal to, or greater than w2 (as unsigned integers).

[min(w1, w2)] returns the smaller of w1 and w2 (as unsigned integers).

[max(w1, w2)] returns the larger of w1 and w2 (as unsigned integers).

[fmt radix w] returns a string representing w, in the radix (base) specified by radix.

radix	description		output format
BIN	unsigned binary	(base 2)	[01]+
OCT	unsigned octal	(base 8)	[0-7]+
DEC	unsigned decimal	(base 10)	[0-9]+
HEX	unsigned hexadecimal	(base 16)	[0-9A-F]+

[toString w] returns a string representing w in unsigned

```
   hexadecimal format.  Equivalent to (fmt HEX w).

   [fromString s] returns SOME(w) if a hexadecimal unsigned numeral
   can be scanned from a prefix of string s, ignoring any initial
   whitespace; returns NONE otherwise.  Raises Overflow if the scanned
   number cannot be represented as a word.  An unsigned hexadecimal
   numeral must have form, after possible initial whitespace:
        [0-9a-fA-F]+

   [scan radix getc charsrc] attempts to scan an unsigned numeral from
   the character source charsrc, using the accessor getc, and ignoring
   any initial whitespace.  The radix argument specifies the base of
   the numeral (BIN, OCT, DEC, HEX).  If successful, it returns
   SOME(w, rest) where w is the value of the numeral scanned, and rest
   is the unused part of the character source.  Raises Overflow if the
   scanned number cannot be represented as a word.  A numeral must
   have form, after possible initial whitespace:

     radix    input format
     -------------------------------------
      BIN     (0w)?[0-1]+
      OCT     (0w)?[0-7]+
      DEC     (0w)?[0-9]+
      HEX     (0wx|0wX|0x|0X)?[0-9a-fA-F]+

   [toInt w] returns the (signed) integer represented by bit-pattern w.
   [toIntX w] returns the (signed) integer represented by bit-pattern w.
   [fromInt i] returns the word representing integer i.

   [toLargeInt w] returns the (signed) integer represented by bit-pattern w.
   [toLargeIntX w] returns the (signed) integer represented by bit-pattern w.
   [fromLargeInt i] returns the word representing integer i.

   [toLargeWord w] returns w.
   [toLargeWordX w] returns w.
   [fromLargeWord w] returns w.
*)
```

D.16. Structure Word8

```
type word = word8
val wordSize : int

val orb  : word * word -> word
val andb : word * word -> word
val xorb : word * word -> word
val notb : word -> word

val <<  : word * Word.word -> word
val >>  : word * Word.word -> word
val ~>> : word * Word.word -> word

val +   : word * word -> word
val -   : word * word -> word
val *   : word * word -> word
val div : word * word -> word
val mod : word * word -> word

val >   : word * word -> bool
val <   : word * word -> bool
val >=  : word * word -> bool
val <=  : word * word -> bool
val compare : word * word -> order

val min : word * word -> word
val max : word * word -> word

val toString   : word -> string
val fromString : string -> word option
val scan : StringCvt.radix
           -> (char, 'a) StringCvt.reader -> (word, 'a) StringCvt.reader
val fmt  : StringCvt.radix -> word -> string

val toInt   : word -> int
val toIntX  : word -> int              (* with sign extension *)
val fromInt : int -> word

val toLargeInt   : word -> int
val toLargeIntX  : word -> int         (* with sign extension *)
val fromLargeInt : int -> word

val toLargeWord   : word -> Word.word
val toLargeWordX  : word -> Word.word   (* with sign extension *)
val fromLargeWord : Word.word -> word

(*
   [word] is the type of 8-bit words, or 8-bit unsigned integers in
   the range 0..255.

   [wordSize] is 8.
```

[orb(w1, w2)] returns the bitwise ‘or’ of w1 and w2.

[andb(w1, w2)] returns the bitwise ‘and’ of w1 and w2.

[xorb(w1, w2)] returns the bitwise ‘exclusive or’ or w1 and w2.

[notb w] returns the bitwise negation of w.

[<<(w, k)] returns the word resulting from shifting w left by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

[>>(w, k)] returns the word resulting from shifting w right by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

[~>>(w, k)] returns the word resulting from shifting w right by k bits. The bits shifted in are replications of the left-most bit: the ‘sign bit’, so this is an arithmetical shift. Consequently,
for k >= wordSize and wordToInt w >= 0 the result is all 0-bits, and
for k >= wordSize and wordToInt w < 0 the result is all 1-bits.

To make <<, >>, and ~>> infix, use the declaration:

```
infix 5 << >> ~>>
```

[+, -, *, div, mod] represent unsigned integer addition, subtraction, multiplication, division, and remainder, modulus 256. The operations (i div j) and (i mod j) raise Div when j = 0. Otherwise no exceptions are raised.

[w1 > w2] returns true if the unsigned integer represented by w1 is larger than that of w2, and similarly for <, >=, <=.

[compare(w1, w2)] returns LESS, EQUAL, or GREATER, according as w1 is less than, equal to, or greater than w2 (as unsigned integers).

[min(w1, w2)] returns the smaller of w1 and w2 (as unsigned integers).

[max(w1, w2)] returns the larger of w1 and w2 (as unsigned integers).

[fmt radix w] returns a string representing w, in the radix (base) specified by radix.

radix	description		output format
BIN	unsigned binary	(base 2)	[01]+
OCT	unsigned octal	(base 8)	[0-7]+
DEC	unsigned decimal	(base 10)	[0-9]+
HEX	unsigned hexadecimal	(base 16)	[0-9A-F]+

[toString w] returns a string representing w in unsigned

```
   hexadecimal format.  Equivalent to (fmt HEX w).

   [fromString s] returns SOME(w) if a hexadecimal unsigned numeral
   can be scanned from a prefix of string s, ignoring any initial
   whitespace; returns NONE otherwise.  Raises Overflow if the scanned
   number cannot be represented as a word.  An unsigned hexadecimal
   numeral must have form, after possible initial whitespace:
       [0-9a-fA-F]+

   [scan radix getc charsrc] attempts to scan an unsigned numeral
   from the character source charsrc, using the accessor getc, and
   ignoring any initial whitespace.  The radix argument specifies the
   base of the numeral (BIN, OCT, DEC, HEX).  If successful, it
   returns SOME(w, rest) where w is the value of the numeral scanned,
   and rest is the unused part of the character source.  Raises
   Overflow if the scanned number cannot be represented as a word.  A
   numeral must have form, after possible initial whitespace:

     radix    input format
     -------------------------------------
      BIN     (0w)?[0-1]+
      OCT     (0w)?[0-7]+
      DEC     (0w)?[0-9]+
      HEX     (0wx|0wX|0x|0X)?[0-9a-fA-F]+

   [toInt w] returns the integer in the range 0..255 represented by w.

   [toIntX w] returns the signed integer (in the range ~128..127)
   represented by bit-pattern w.

   [fromInt i] returns the word holding the 8 least significant bits of i.

   [toLargeInt w] returns the integer in the range 0..255 represented by w.

   [toLargeIntX w] returns the signed integer (in the range ~128..127)
   represented by bit-pattern w.

   [fromLargeInt i] returns the word holding the 8 least significant bits of i.

   [toLargeWord w] returns the Word.word value corresponding to w.

   [toLargeWordX w] returns the Word.word value corresponding to w,
   with sign extension.  That is, the 8 least significant bits of the
   result are those of w, and the remaining bits are all equal to the
   most significant bit of w: its `sign bit'.

   [fromLargeWord w] returns w modulo 256.
*)
```