DEKLARATÍV PROGRAMOZÁS

OKTATÁSI SEGÉDLET

Bevezetés a funkcionális programozásba

Ötödik, bővített kiadás

Hanák D. Péter

Irányítástechnika és Informatika Tanszék

Budapest, 2005. március

1.2. A monolitikus és a strukturált programozásról . . . 7
1.3. Az imperatív programozási paradigma . . . 8
1.4. A deklaratív programozási paradigma . . . 9
1.4.1. A logikai programozási paradigma . . . 9
1.4.2. A funkcionális programozási paradigma . . . 9
1.4.2.1. A *read-eval-print* ciklus . . . 10
1.4.2.2. Hivatkozási átlátszóság . . . 10
1.4.2.3. A funkcionális program . . . 10
1.5. SML-értelmezők és fordítók . . . 10
1.6. Információforrások . . . 12
1.7. Változások az előző kiadáshoz képest . . . 13
1.8. Köszönetnyilvánítás . . . 13
1.9. Hibajelentés . . . 13

2. Egyszerű példák SML-ben **14**
2.1. Egész szám négyzete . . . 14
2.2. Legnagyobb közös osztó . . . 14
2.3. Intervallumösszeg . . . 15
2.4. Pénzváltás . . . 17

3. Nevek, függvények, egyszerű típusok **19**
3.1. Értékdeklaráció . . . 19
3.1.1. Névadás állandónak . . . 19
3.1.2. Névadás függvénynek . . . 20
3.1.3. Nevek újradefiniálása . . . 21
3.1.3.1. Nevek képzése . . . 21
3.2. Egész, valós, füzér, karakter és más egyszerű típusok . . . 22
3.2.1. Egészek és valósak . . . 22
3.2.1.1. A `real`, `floor`, `ceil`, `abs`, `round` és `trunc` függvény . . . 22
3.2.1.2. Alapműveletek előjeles egész számokkal . . . 23
3.2.1.3. Alapműveletek valós számokkal . . . 24
3.2.1.4. Alapműveletek előjel nélküli egészekkel . . . 25
3.2.2. Típusmegkötés . . . 25
3.2.3. Füzérek . . . 26
3.2.3.1. Escape-szekvenciák . . . 26
3.2.3.2. Gyakori műveletek füzérekkel . . . 27
3.2.4. Karakterek . . . 27
3.2.4.1. Gyakori műveletek karakterekkel . . . 27

4.1.4. Ennes elemeinek kiválasztása mintaillesztéssel . . . 34
4.1.5. A nullas és a `unit` típus . . . 35
4.1.5.1. A `print`, a `use` és a `load` üggvény . . . 35
4.2. Rekord . . . 35
4.2.1. Rekordminta . . . 36
4.2.2. Gyakorló feladatok . . . 37
4.3. Polimorfizmus . . . 37
4.3.1. Polimorf típusellenőrzés . . . 37
4.3.2. Egyenlőségvizsgálat polimorf függvényekben . . . 38

5. Kiértékelés, deklaráció **39**
5.1. Kifejezések kiértékelése az SML-ben . . . 39
5.1.1. Mohó kiértékelés . . . 40
5.1.1.1. Mohó kiértékelés rekurzív függvények esetén . . . 40
5.1.1.2. Iteratív függvények . . . 40
5.1.1.3. Feltételes kifejezések speciális kiértékelése . . . 41
5.1.2. Lusta kiértékelés . . . 42
5.1.3. A mohó és a lusta kiértékelés összevetése . . . 42
5.2. Lokális érvényű és egyidejű deklaráció . . . 43
5.2.1. Kifejezés lokális érvényű deklarációval . . . 43
5.2.2. Deklaráció lokális érvényű deklarációval . . . 44
5.2.3. Egyidejű deklaráció . . . 44
5.3. *Gyakorló feladat* . . . 44

6. Számítások rekurzív függvényekkel **45**
6.1. Egész kitevőjű hatványozás . . . 45
6.2. Fibonacci-számok . . . 46
6.3. Egész négyzetgyök közelítéssel . . . 48
6.4. Valós szám négyzetgyöke Newton-Raphson módszerrel . . . 49
6.5. $\pi/4$ közelítő értéke kölcsönös rekurzióval . . . 50

7. Listák **52**
7.1. Listajelölések . . . 52
7.1.1. Típuskifejezés . . . 52
7.2. Lista létrehozása . . . 53
7.3. Egyszerű műveletek listákkal . . . 53
7.3.1. Egyesével növekvő számtani sorozat . . . 53
7.3.2. Lista elemeinek szorzata és összege . . . 53

---

9. Magasabbrendű függvények **67**
9.1. Az `fn` jelölés . . . 67
9.1.1. Függvény definiálása `fun`, `val` és `val rec` kulcsszóval . . . 68
9.2. Részlegesen alkalmazható függvények . . . 68
9.3. Magasabbrendű függvények . . . 69
9.3.1. `secl` és `secr` . . . 70
9.3.2. Két függvény kompozíciója . . . 70
9.3.3. `curry` és `uncurry` . . . 71
9.3.4. `map` és `filter` . . . 71
9.3.4.1. Gyakorló feladat . . . 72
9.3.5. `takewhile` és `dropwhile` . . . 72
9.3.6. `exists` és `forall` . . . 73
9.3.7. `foldl` és `foldr` . . . 74
9.3.8. `repeat` . . . 75
9.3.9. `map` újradefiniálása `foldr`-rel . . . 76

10. Kivételkezelés **77**
10.1. Kivétel deklarálása az `exception` kulcsszóval . . . 77
10.2. Kivétel jelzése a `raise` kulcsszóval . . . 77
10.2.1. Belső kivételek . . . 78
10.3. Kivétel feldolgozása a `handle` kulcsszóval . . . 78
10.4. Néhány példa a kivételkezelésre . . . 78

11. Bináris fák **81**
11.1. Egyszerű műveletek bináris fákon . . . 82
11.2. Lista előállítása bináris fa elemeiből . . . 84
11.3. Bináris fa előállítása lista elemeiből . . . 85
11.4. Elem törlése bináris fából . . . 86
11.5. Bináris keresőfák . . . 87

12. Listák rendezése **89**
12.1. Beszúró rendezés . . . 89
12.1.1. Generikus megoldások . . . 89
12.1.2. Beszúró rendezés `foldr`-rel és `foldl`-lel . . . 91
12.1.3. A futási idők összehasonlítása . . . 91
12.2. Gyorsrendezés . . . 93
12.3. Összefésülő rendezés . . . 95
12.3.1. Fölülről lefelé haladó összefésülő rendezés . . . 95
12.3.2. Alulról fölfelé haladó összefésülő rendezés . . . 96

13.6.1. Keresztszorzatokból álló lista . . . 107
13.6.2. Keresztszorzatokból álló lusta lista . . . 108

14. Példaprogramok: füzérek és listák **110**
14.1. Füzér adott tulajdonságú elemei (`mezok`) . . . 110
14.2. Füzér adott tulajdonságú elemei (`basename`) . . . 112
14.3. Füzér adott tulajdonságú elemei (`rootname`) . . . 112
14.4. Füzér egyes elemeinek azonosítása (`parPairs`) . . . 113
14.5. Lista adott tulajdonságú részlistái (`szomsor`) . . . 114
14.6. Bináris számok inkrementálása (`binc`) . . . 115
14.7. Mátrix transzponáltja (`trans`) . . . 116

15. Példaprogramok: fák **118**
15.1. Fa adott tulajdonságának ellenőrzése (`ugyanannyi`) . . . 118
15.2. Fa adott tulajdonságú részfáinak száma (`bea`) . . . 120
15.3. Fa adott tulajdonságú részfáinak száma (`testverE`) . . . 121
15.4. Fa adott elemeinek összegzése (`szintOssz`) . . . 122
15.5. Kifejezésfa egyszerűsítése (`egyszerusit`) . . . 124
15.6. Kifejezésfa egyszerűsítése (`coeff`) . . . 125

16. Egy egyszerű fordítóprogram SML-ben **127**
16.1. A forrásnyelv . . . 127
16.2. A forrásnyelv konkrét szintaxisa . . . 127
16.3. A célnyelv . . . 128
16.4. A fordítás folyamata . . . 130
16.5. A forrásnyelv absztrakt szintaxisa . . . 130
16.6. A fordítóprogram építőkockái . . . 130
16.7. A fordító forráskódja SML-nyelven . . . 132
16.7.1. Symtab szignatúrája és struktúrája . . . 133
16.7.2. Lexical szignatúrája és struktúrája . . . 137
16.7.3. Parsefun szignatúrája és struktúrája . . . 139
16.7.4. Parse szignatúrája és struktúrája . . . 145
16.7.5. Encode szignatúrája és struktúrája . . . 148
16.7.6. Assemble szignatúrája és struktúrája . . . 152
16.7.7. Compile szignatúrája és struktúrája . . . 155

B.2. Structure Binaryset . . . 167
B.3. Structure Bool . . . 169
B.4. Structure Char . . . 169
B.5. Structure General . . . 173
B.6. Structure Int . . . 178
B.7. Structure List . . . 180
B.8. Structure ListPair . . . 182
B.9. Structure Listsort . . . 184
B.10. Structure Math . . . 184
B.11. Structure Meta . . . 185
B.12. Structure Option . . . 189
B.13. Structure Random . . . 190
B.14. Structure Real . . . 191
B.15. Structure Regex . . . 193
B.16. Structure Splaymap . . . 198
B.17. Structure Splayset . . . 199
B.18. Structure String . . . 201
B.19. Structure StringCvt . . . 203
B.20. Structure TextIO . . . 204
B.21. Structure Time . . . 208
B.22. Structure Timer . . . 210
B.23. Structure Word . . . 211
B.24. Structure Word8 . . . 214

Ez a jegyzet oktatási segédletként a *Deklaratív programozás* 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* (vagy *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

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 körülményes modellezni az időt, a valós világ jelenségeit, a ki- és beviteli műveleteket.[5]

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 és a Borland Pascal, a Delphi 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).
[5] Egyes „tiszta” funkcionális programozási nyelvek, pl. a haskell és a clean speciális nyelvi elemekkel oldják meg az állapotváltozás nélküli programozást.

- 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 vagy 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 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[8] 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*[9] 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 Robin Milner megvalósította típuselméleti eredményeit. Eredetileg *logikai állítások igazolására, tételbizonyításra* tervezték, erre utal a nem túl ötletes *Meta Language* elnevezés is. A HOPE-pal

[6] Ugyancsak latin szó, jelentése: kijelentő, kinyilatkoztató (vö. deklaráció).
[7] Vannak, akik a deklaratív programozást a logikaival azonosítják, és a funkcionális programozást nem tekintik deklaratívnak.
[8] A függvényérték olyan érték, amely függvényként más értékre alkalmazható.
[9] Szintén latin szó, jelentése: alkalmazó (vö. applikáció).

záshoz 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. 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 1997-ben 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 négyet ajánlunk az olvasó figyelmébe. Mind a négy használható egyebek mellett linux, WinNT, Win2k és WinXP alatt is.

ugyancsak szabványos *modulnyelvel,* valamint az 1997-es definíciónak megfelelő *alapkönyvtárat* megvalósítja, és ezeken felül számtalan új elemmel egészíti ki mindhármat. Többek között lehetővé teszi a lusta kiértékelést, a párhuzamos és elosztott programozást, a korlát-alapú programozást, valamint a dinamikus kötést.

Az SML-nyelvvel most ismerkedők igényeinek a kis erőforrásigényű *mosml* mindenben megfelel.

Az SML-értelmezőknek 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-mód. (A Prolog-értelmezők kényelmes használatához ugyancsak az *emacs*-ra, mégpedig az *emacs* Prolog-módjára van szükség.) Windows alatt vannak más olyan programok is, amelyek az értelmezők kényelmesebb kezelését teszik lehetővé. A tárgy előadói az emacs használatát preferálják.

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*, az *smlnj*, a *Poly/ML* és az *Alice* válaszai és főleg hibaüzenetei különböznek egymástól. Ebben a jegyzetben rendszerint az *mosml* 2.01 verziójának válaszait és hibaüzeneteit adjuk meg, és általában utalunk rá, ha ettől valamilyen ok miatt eltérünk.

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

- a `quit()` függvényhívással,
- Windows alatt a ***ctrl-z,*** majd az ***enter*** leütésével,
- unix (linux) alatt a ***ctrl-d*** leütésével,
- a `Process.exit` *`arg`* függvényhívással, ahol *`arg`*-nak `Process.status` típusú értéknek kell lennie.

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

Az angol nyelvű könyvek közül különösen a következőket javasoljuk:

1. Jeffrey D. Ullman: *Elements of ML Programming* (2nd Edition, ML97). MIT Press 1997.
   `<http://www-db.stanford.edu/~ullman/emlp.html>`
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. Hal Abelson, Jerry Sussman, Julie Sussman: *Structure and Interpretation of Computer Programs* (MIT Press, 1984; ISBN 0-262-01077-1)
   `<http://mitpress.mit.edu/sicp>` Letölthető a teljes könyv elektronikus változata!
4. Richard Bosworth: *A Practical Course in Functional Programming Using Standard ML.* McGraw-Hill 1995. ISBN: 0-07-707625-7.

Az on-line információforrások közül javasoljuk a következőket:

1. Andrew Cumming: *A Gentle Introduction to ML.*
   `<http://www.dcs.napier.ac.uk/course-notes/sml/manual.html>`
2. Stephen Gilmore: *Programming in Standard ML ´97: An On-line Tutorial.*
   `<http://www.dcs.ed.ac.uk/home/stg/NOTES>`
3. Hal Abelson, Jerry Sussman, Julie Sussman: *Structure and Interpretation of Computer Programs*
   `<http://mitpress.mit.edu/sicp>`
4. Robert Harper: *Programming in Standard ML*
   `<http://www-2.cs.cmu.edu/ rwh/smlbook/offline.pdf>`
5. Gerd Smolka: *Programmierung. Eine Einführung in die Informatik.*
   `<http://www.ps.uni-sb.de/courses/prog-ws04/script/index.html>`
6. *COMP.LANG.ML Frequently Asked Questions and Answers.*
   `<http://www.faqs.org/faqs/meta-lang-faq/>`

Az *mosml*, az *smlnj*, a *Poly/ML* és az *Alice* honlapjának címe:

1. Moscow ML: `<http://www.dina.kvl.dk/~sestoft/mosml.html>`
2. Standard ML of New Jersey: `<http://www.smlnj.org/>`
3. Poly/ML: `<http://www.polyml.org/>`
4. Alice: `<http://www.ps.uni-sb.de/alice>`

[illegible] harmadik szakasza a *Listák* c. fejezetbe került át.

- A korábbi *Részlegesen alkalmazható függvények* c. fejezet anyaga bekerült a *Magasabbrendű függvények* c. fejezetbe.
- Új az *Egy egyszerű fordítóprogram SML-ben* c. fejezet.
- A *Válogatás az SML Alapkönytárából* c. fejezet további struktúrák – *Binarymap*, *Binaryset*, *ListPair*, *Random*, *Regex*, *Splaymap*, *Splaytree* és *StringCvt* – rövid leírását tartalmazza.
- A többi fejezet szövegében és szerkezetében is vannak kisebb-nagyobb változások.

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.

Ebben a fejezetben, ízelítőként, néhány egyszerű programot mutatunk be SML-ben.

2.1. Egész szám négyzete

```
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.

2.2. Legnagyobb közös osztó

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

$$\gcd(0, n) = n$$
$$\gcd(m, n) = \gcd(n \bmod m, m), \text{ ha } m > 0$$

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:

2.3. Intervallumösszeg

Adott az $s \geq 1$ egész szám. Határozzuk meg azt a lehető leghosszabb $[i, j]$ zárt intervallumot, amelyre $1 \leq i \leq j$ és az s az $[i, j]$ intervallumba eső számok összegével egyenlő.

Kézenfekvőnek látszik a következő algoritmus: az intervallum alsó és felső határát is 1-ről indítjuk. Egy ciklusban addig növeljük a felső határt, amíg az intervallumba eső számok összege kisebb s-nél, ill. addig növeljük az alsó határt, amíg a számok összege nagyobb s-nél.

Ciklus helyett rekurzív segédfüggvényt használunk az SML-ben. Ahelyett, hogy az intervallumba eső számokat minden lépésben újból és újból összeadnánk, akkumulátort (másnéven gyűjtőargumentumot) használunk az összeg képzésére. Valahányszor az alsó határt növeljük meg, az értékét kivonjuk akkumulátorból, és valahányszor a felső határt növeljük meg, az értékét hozzáadjuk az akkumulátorhoz.

`intvalsum` két számpárt adjon eredményül: az első számpár a zárt intervallum alsó és felső határa, a második számpár első tagja az intervallum hossza, második tagja pedig a rekurzív hívások – a szükséges lépések – száma legyen.

Ha az $s < 1$ argumentumra alkalmazzuk az `intvalsum` függvényt, `((0, 0), (0, 0))` legyen a visszaadott érték.

```
(* ivs (s, i, j, t, n) = ((0, 0), (0, 0)) s < 1-re, egyébként
              ((a, b), (c, d)), ahol [a,b] az a lehető leghoszabb
              zárt intervallum, amely elemeinek összege s, hossza c, a
              meghatározásához szükséges rekurzív hívások száma pedig d
   ivs : int * int * int * int * int -> (int * int) * (int * int)
   PRE : (i, j, t, n) = (1, 1, 1, 1,)
*)
fun ivs (s, i, j, t, n) =
      if s < 1
      then ((0, 0), (0, 0))
      else if t < s
      then ivs (s, i, j+1, t+j+1, n+1)
      else if t > s
      then ivs (s, i+1, j, t-i, n+1)
      else ((i, j), (j-i+1, n+1));
```

A `PRE` szócska az algoritmus helyes működésének előfeltételét *(precondition)* adja meg.

```
(* intvalsum s = ((0, 0), (0, 0)) s < 1-re, egyébként ((a, b), (c, d)),
              ahol [a,b] az a lehető leghoszabb zárt intervallum, amely
              elemeinek összege s, hossza c, a meghatározásához
              szükséges rekurzív hívások száma pedig d
```

```
intvalsum 67108864;
> val it = ((67108864, 67108864), (1, 134217728))
                                    : (int * int) * (int * int)

intvalsum 268435456;
> val it = ((268435456, 268435456), (1, 536870912))
                                    : (int * int) * (int * int)
```

Az 536 870 912 lépés megtételéhez még egy 1.3 GHz-es CPU-n is kb. egy percre volt szüksége az mosml-nek. Az smlnj egy nagyságrenddel gyorsabban, „mindössze” 6 s alatt állította elő az eredményt. Van mit javítani az algoritmus hatékonyságán! Nézzük, mit tehetünk.

A lépések számát radikálisan csökkenthetjük, ha legfeljebb annyi lépést teszünk meg, amennyi a keresett intervallum hossza. Az azonos összegű intervallumok közül az lesz a leghosszabb, amelynek az alsó határa a lehető legkisebb, mert ilyenkor összegezzük a lehető legkisebb számokat.

A keresett intervallum hosszának tehát van felső korlátja: annak az intervallumnak a hossza, amely 1-től kezdődik, m-ig tart, és az elemeinek összege nem kisebb s-nél. Képlettel: $(1 + m) \cdot m/2 \geq s$, azaz $m^2 + m \geq 2s$, azaz $m \geq \sqrt{2s - m}$. Ha felső korlátnak az $f = \lfloor \sqrt{2s} \rfloor$ számot választjuk, legfeljebb néhány lépéssel kell többet megtennünk, cserébe egyszerűbb lesz az r kiszámítása.

Az egész számok körében maradva az f-et az alábbi `lenLimit` függvénnyel számíthatjuk ki:

```
(* lenLimit (s, h) = felső korlát az s összegű zárt intervallum
                     hosszára
   lenLimit : int * int -> int
   PRE : f = 1
*)
fun lenLimit (s, h) = if h*h div 2 < s
                      then lenLimit(s, h+1)
                      else h-1;
```

A `h*h < 2*s` helyett a `h*h div 2 < s` kifejezést számítjuk ki a függvényben, mert így valamivel később ütközünk az egész számok ábrázolásának felső korlátjába. `div` az egészosztás hányadosát adja eredményül.

Ezek után a feltételt kielégítő intervallumot a lehető leghosszabbtól kezdve kereshetjük. Amíg nem találjuk meg, minden lépésben eggyel csökkentsük a jelölt h hosszát, és vizsgáljuk meg a $k = s - (1 + h) \cdot h/2$ különbséget! Akkor van meg a megoldás, amikor a k a h egész számú többszörösévé válik, mert ilyenkor az intervallumot k/h-val felfelé eltolva valóban a lehető leghosszabb s összegű intervallumot kapjuk.

```
            if k mod h <> 0
            then ivs(s, h-1, n+1)
            else ((k div h + 1, k div h + h), (h, n))
        end;
```

Az `s - h * (h+1) div 2` kifejezés értékére többször is szükség van a függvény törzsében, ezért előre kiszámítjuk és elnevezzük `k`-nak. `mod` az egészosztás maradékát adja eredményül. `k mod h <> 0` helyett vizsgálhatnánk a `k - k div h * h > 0` feltételt is.

```
fun intvalsum s = ivs(s, lenLimit(s, 1), 1);
```

Nézzük, mit kapunk most eredményül a korábban már vizsgált esetekben!

```
intvalsum 8192;
> val it = ((8192, 8192), (1, 127)) : (int * int) * (int * int)

intvalsum 67108864;
> val it = ((67108864, 67108864), (1, 11585)) :
                                    (int * int) * (int * int)

intvalsum 268435456;
> val it = ((268435456, 268435456), (1, 23170)) :
                                    (int * int) * (int * int)
```

A megfelelő algoritmus a megoldást több nagyságrenddel kevesebb lépésben, egy szemvillanásnyi idő alatt állítja elő. A tanulság az lehet, hogy a programjaink végrehajtási idejét nem a rekurzió használata, hanem az ügyetlenül megválasztott algoritmusok viselkedése növeli meg tetemesen.

2.4. 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:

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) *klózokat* 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 klózt 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 klózok 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 klózra ne kerüljön sor.

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

3.1. 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.)

3.1.1. 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.

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 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 argumentumként megadott kifejezést csak akkor kell zárójelbe tenni, ha erre precedenciaokok miatt szükség van. Helyesek tehát az alábbi példák:

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

[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.

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.*

3.1.3.1. 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 (ún. tapadó 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.

[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.

3.2.1. 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`

3.2.1.1. 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 egyenlőtlenségekkel* 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
```

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

[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.

```
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
```

3.2.1.2. 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

- 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.

[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.

deklarációt (ajánlott precedenciaszintjük a 7-es, v.ö. 3.3.1. szakasz): `infix 7 quot rem`.

3.2.1.3. 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

`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 (ld. a 3.2.2. 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.

A műveletek precedenciájáról további részletek a 3.3.1. szakaszban olvashatók.

3.2.2. 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.

[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.

3.2.3.1. 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 3.2.4. 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]

[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";
```

3.2.4. Karakterek

Az SML-ben a karakterek típusa `char`, a jelölésük pedig elég körülményes: egyetlen karakterből álló füzérállandó, amely előtt `#` áll. Ez azért van így, mert eredetileg nem volt karakterjelölés az SML-ben, karakter helyett egyetlen karakterből álló füzért használtak. Példák karakterjelölésre:

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

3.2.4.1. 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
```

A relációs operátorok (<, <=, =, >=, >, <>) `char` típusú értékek összehasonlítására is használhatók.

3.2.5. Igazságértékek, logikai kifejezések, feltételes kifejezések

Az SML igazságérték-típusa a `bool` (v.ö. 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: <=, <, >, >=,

[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.

- 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.

3.2.5.2. Logikai operátorok

Az SML-ben három logikai operátort alkalmazhatunk, ezek a kétoperandusú `andalso` és `orelse`, valamint az egyoperandusú `not`. Az `andalso` és az `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 (vö. 3.3.1. szakasz). 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;
```

[13] Jegyezzük meg, hogy egyes SML-megvalósításokban y valós számok egyenlőségét nem lehet vizsgálni az SML-ben (más nyelv használata esetén sem célszerű!), mert az ábrázolás pontatlansága, ill. 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. Az *smlnj* a `real` típust nem is tekinti egyenlőségi típusnak.)

[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 `||`.

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)`

3.3. Infix operátorok

A fejezet hátralévő részében összefoglaljuk, amit az infix operátorokról tudni kell.

3.3.1. Infix operátorok precedenciája

A következő táblázat az *infix* pozícióban használható operátorok típusát és precedenciáját mutatja az SML-ben.

[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 alakalmazható függvényekről szóló fejezetben magyarázzuk meg.

	`>, >=`	*`numtxt * numtxt`* `-> bool`	nagyobb, nagyobb-egyenlő
3	`:=`	`'a ref * 'a -> unit`	értékadás
	`o`	`('b->'c)*('a->'b) -> ('a->'c)`	két függvény kompozíciója
0	`before`	`'a * 'b -> 'a`	a bal oldali argumentum

Megjegyzések a táblázathoz:

1. A *`numtxt`* megnevezés a *`num`*-csoportba tartozó típusok mellett a `char` és a `string` típust jelenti.
2. Az `'a` és a `''a` ún. típusváltozók. (A típusváltozókról később részletesen szólunk.) A `''a` típusváltozó az ún. *egyenlőségi típust*, azaz olyan értékek típusát jelöli, amelyeken az *egyenlőségvizsgálat* elvégezhető.

3.3.2. Felhasználói infix operátor

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 az előző szakaszban szóltunk. A programozó az `infix` deklarációban meghatározhatja egy operátor precedenciáját is (ld. a következő példát).

A logikai operátorokról szóló 3.2.5.2. 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 |||;
```

```
- 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.

3.3.3. 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
```

```
:
!  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 (ld. 3.1.3.1.):

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

4.1. Ennes

A `(firstname, lastname)` egy *pár*, azaz kételemű ennes. A `(day, month, year)` egy *hármas*, azaz háromelemű ennes. Általában: egy *ennes* (angolul: *tuple*) elemeit vessző választja el egymástól, az elemek tetszőleges és egymástól különböző típusúak lehetnek, a sorrendjük fontos.[1] 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.

4.1.1. 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.

Az infix típusoperátoroknak is van precedenciájuk és lehet kötésük. A *keresztszorzat* (Descartes-szorzat, jele a `*`) precedenciája nagyobb a *leképzés* (jele a `->`) precendenciájánál. A `->` típusoperátor *jobbra köt*. A `*` típusoperátor különlegessége az, hogy nem köt semmerre! Típuskifejezésben is használható *zárójel* a műveletek sorrendjének meghatározására.[2]

Látni fogjuk a 7.1.1. szakaszban és a 8. fejezetben, hogy az eddig megismerteken kívül vannak más típusállandók és típusoperátorok is, sőt a programozó maga is definiálhat típusállandókat, típusváltozókat és típusoperátorokat.

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 *szinonimája.*[3]

[1] Később, a 4.1.5. 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.

[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.

[3] Új adattípus létrehozására, azaz *erős absztrakcióra* kétféle lehetőség is van az SML-ben. Az egyik a sokféleképpen használható `datatype` deklaráció, amelyről a 8. fejezet szól, és amellyel az SML modulszerkezetébe (`structure` és `signature`) rejtve valósíthatunk meg erős absztrakciót. A másik lehetőség a már elavult `abstype` deklaráció.


```
> val negvec = fn : real * real -> real * real
- negvec b;
> val it = (~3.60000, ~0.90000) : real * real
```

4.1.3. 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` típuskifejés áll! Az *mosml* tervezői ezzel is tudatosítani akarják, hogy a `vec` név csak *szinonima,* 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` szinonimát 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.

4.1.4. 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.

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.[4] 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.

4.2. Rekord

A *rekord* olyan ennes, amelyben az egyes elemeket megcímkézzük: a sorrend többé nem fontos, az elemekre nem a helyük szerint, hanem a címkéjükkel hivatkozunk, ezért 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
```

[4] Az egyes értelmezők másképpen kezelik, ezért mindenütt másképpen kell betölteni a modulokat.

```
> 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
```

4.2.1. 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.

---

típusa: `scalevec : real * vec -> vec`.

4. 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 lesz az *mosml*-értelmező válasza a `harom()` kifejezésre?

4.3. 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.)

4.3.1. 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
```

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

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

4.3.2. 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.[5]

Az egyenlőség tehát csak *korlátozott értelemben* polimorf művelet. *Egyenlőségi típusnak* (equality type) az olyan típust nevezzük, 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
```

[5] A `datatype` deklarációról szól a 8. fejezet.

5.1. 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
*)
fun zero x = 0;
```

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

[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. Jegyezzük meg, hogy különbség van 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.


```
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.

5.1.1.1. Mohó kiértékelés rekurzív függvények esetén

A *faktoriális* matematikai definíciója:

$\text{fac}\,0 = 1$

$\text{fac}\,n = n{*}\text{fac}(n-1)$

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.

5.1.1.2. 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:

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.

5.1.1.3. 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 n = 2 n-edik hatványa
   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.

---

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
```

5.1.3. 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] Az SML egyik kiterjesztése, az Alice már lusta kifejezések használatát is megengedi. A 13. fejezetben több példát is bemutatunk ún. lusta listák létrehozására és használatára.

5.2. Lokális érvényű és egyidejű deklaráció

Egy korábbi fejezetben már megismerkedtünk a (globális) értékdeklaráció fogalmával. A következő két szakaszban olyan deklarációkkal foglalkozunk, amelyek *lokális érvényűek*, a harmadikban pedig a kölcsönösen rekurzív függvények deklarálásához szükséges *egyidejű deklarációt* ismertetjük.

5.2.1. Kifejezés lokális érvényű deklarációval

Az olyan kifejezést, amelyben lokális érvényű deklarációt (rövidebben: lokális deklarációt) használunk, a `let` kulcsszó vezeti be. Szintaxisa a következő:

`let` D `in` E `end`

D sokszor nem egyetlen deklaráció, hanem D_1`;` D_2`;` ...`;` D_n alakú deklarációsorozat, más néven *szekvenciális deklaráció*, amelyben a `;` (pontosvessző) opcionális. Az ilyen kifejezést lokális deklarációt használó kifejezésnek vagy `let`-kifejezésnek nevezzük.

`let`-kifejezésben *érték, függvény, típus* és *kivétel* deklarálható. A deklarált lokális érték csak magában a `let`-kifejezésben látható.

Most arra mutatunk példát, hogy mikor **ne** használjunk `let`-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!

`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
```

5.3. *Gyakorló feladat*

Írja át `isLetter` és segédfüggvényei korábbi definícióját (ld. 3.2.5.3. szakasz) `if`-ekkel, `andalso` és `orelse` nélkül!

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 jobbrekurzívvá tenni egy rekurzív függvényt. Most további, összetettebb példákat mutatunk be.

6.1. 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
   ** : real * int -> real
   PRE: k >= 0
*)
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` kifejezéssel, ugyanakkor átláthatóbb, világosabban mutatja az esetek szétválasztását – csak éppen nem mindig alkalmazható. Ha például a `**` operátort negatív `k`-kra is definiálni akarjuk, mintaillesztést nem, csak `if-then-else` kifejezést használhatunk.

A bemutatott megoldás nem elég hatékony. Hogyan javíthatunk a hatékonyságán? Felhasználhatjuk a következő azonosságokat:

$x^1 = x$
$x^{2k} = (x^2)^k, k > 0$
$x^{2k+1} = x \cdot x^{2k}, k > 0$

Egy példa: $2^{10} = 4^5 = 4 * 4^4 = 4 * 16^2 = 4 * 256 = 1024$, és egy megoldás:

1. `k<0` esetén *kivételkezelést*,
2. a `k=0` vizsgálat felesleges ismétlődését kiküszöbölő *lokális deklarációt használó deklarációval,*
3. a `pwr(x*x, k div 2)` függvényalkalmazás kétszeri kiszámítását kiküszöbölő *lokális deklarációt használó kifejezéssel.*

A `pwr` függvény definíciója pl. így módosul, ha lokális deklarációt használó 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;
```

6.2. 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-be, használhatatlan programot kapunk: pl. F_{40}-et egy 1.3 GHz-es Pentium processzoros számítógépen 5 s alatt számolja ki az smlnj és csaknem 21 s alatt az mosml! Ezért most `nextfib` néven olyan függvényt írunk, 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
   fibpair : int -> int * int
   PRE : n > 0
*)
fun fibpair n = if n = 1 then (0, 1) else nextfib(fibpair(n-1));
```

```
   iterfib : int * int * int -> int
   PRE : n > 0
*)
fun iterfib (1, prev, curr) = curr
  | iterfib (n, prev, curr) = iterfib(n-1, curr, prev + curr)
```

Az `else` ágban a rekurzió jobbrekurzió. `fib` `iterfib`-et hívja meg az `n`-edik Fibonacci-szám előállításához:

```
(* fib n = az n-edik Fibonacci-szám
   fib : int -> int
   PRE : n > 0
*)
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
      iterfib : int * int * int -> int
      PRE : n > 0
   *)
   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 deklarációt használó deklaráció helyett lokális deklarációt használó kifejezést is használhatnánk, de ez nem mindig van így.

$$i^2 \leq m < (i+1)^2$$

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 alábbi `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.

---

tása 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
```

6.5. $\pi/4$ 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
*)
fun pos d = neg(d - 2.0) + 1.0/d

(* neg d = pi/4.0 közelítő értékének negatív előjelű,
           d nevezőjű tagja (d = 3.0, 7.0, 11.0, ...)
   neg : real -> real
*)
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
   sum : int -> real
   PRE : n >= 0
*)
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:

```
            val s = if n mod 2 = 0 then 1.0 else ~1.0
        in
            4.0 * pi4(d, s)
        end
end
```

A *lista* lineáris adatszerkezet, azonos típusú elemek sorozata. A listát *rekurzív lineráris adatszerkezetnek* is tekinthetjük. A rekurzív definíció szerint a lista

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

7.1. Listajelölések

Az üres listát – a listaműveletek *egységelemét* – `[]`-lel vagy `nil`-lel jelöljük. A nemüres lista elemeit egymástól vesszővel elválasztva szögletes zárójelek – `[` és `]` – között sorolhatjuk fel, pl. `[3, 5, 9]`. A legalább egy elemből álló lista első elemét a lista *fejének,* a lista maradékát 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] Példák:

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

7.1.1. Típuskifejezés

Ha egy lista elemeinek a típusa `´a`, akkor a lista típusa `´a list`.[2] 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 4.1.1. szakaszban már szó volt arról, hogy a típusoperátoroknak is van *precedenciájuk:* a `*` precedenciája nagyobb a `->` precedenciájánál, a most megismert`list` precedenciája pedig a `*` precedenciájnál is nagyobb. A `*` és a `->` *infix,* a `list` *postfix* pozíciójú.

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] Típus nélküli funkcionális nyelvekben, pl. a LISP-ben a listának különböző típusú elemei is lehetnek.
[2] Az itt `´a`-val jelölt típusváltozóval és a polimorf típussal részletesen a 4.3.1. szakasz foglalkozik.

7.3.1. Egyesével növekvő számtani sorozat

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)
```

7.3.2. Lista elemeinek szorzata és összege

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 (a Prolog szóhasználatával:) *klózok* a jelen esetben *kölcsönösen kizárják* egymást (a minták diszjunktak), ezért a két klóz sorrendje közömbös, a klózokat fordított sorrendben is írhatjuk:

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

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

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

[3] `[]` 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.

[4] 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.

```
*)
fun maxl [m] = m
  | maxl (m::ms) = let
                       val n = maxl ms
                   in
                       if m > n then m else n
                   end;
```

Elegánsabb `maxl` következő, a `max` függvényt alkalmazó változata:

```
fun max (m,n) = if m > n then m else n;
fun maxl [m] = m
  | maxl (m::ms) = max(m, maxl ms);
```

2. vesszük az első két elem közül a nagyobbat, a maradéklista elé fűzzük, majd rekurzív módon meghatározzuk az így kapott – az eredetinél eggyel rövidebb – lista legnagyobb elemét:

```
fun maxl [m] = m
  | maxl (m::n::ns) = maxl(max(m,n)::ns);
```

Vegyük észre, hogy `maxl`-nek ez a változata jobbrekurzív.

Az SML-értelmező mindhárom fenti 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 többszörösen terhelhető módon polimorf (ld. 4.3.1 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). A 12. fejezetben megmutatjuk, hogyan kell ún. *generikus* algoritmusokat írni.

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

A következő szakaszokban néhány fontos listakezelő függvényt definiálunk.

7.4. 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.[5]

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

A minták felírásának sorrendje közömbös ebben a függvényben is.

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

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

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

Ez a `hd` és `tl` csak nemüres listára alkalmazható, amire az SML-értelmező figyelmeztet. Belső változatuk üres lista esetén kivételt jelez.

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

[5] `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.

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

Egy példa a függvény alkalmazására:

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

`nlength` rossz hatékonyságú, mert nem jobbrekurzív: az 1-esek a veremben csak gyűlnek, gyűlnek, amíg a maradéklista ki nem ürül. A függvény javított, jobbrekurzív 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 *rosztussága*[7] 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 ([], _) = []
```

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

[7] 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.

```
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]
```

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 elővesszi őket az eredménylista előállításához.

Következő kérdésünk az, hogy érdemes-e megírni `take` jobbrekurzív 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, csak annyit nyerünk vele, hogy a veremhasználat korlátos marad, mert az eredménylistát még meg kell fordítani.

```
(* 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 jobbrekurzív is.

Az `else` ágban lévő `x::xs` lista ugyanaz, mint a `drop` második argumentuma: ha az *as* kulcsszó alkalmazásával ún. *réteges mintát* (layered pattern) használunk, megtakarítjuk az `else` ágban a lista újraépítésének költségét az `x` elemből és az `xs` listából:

```
fun drop (_, [])= []
  | drop (i, xxs as x::xs) = if i>0 then drop (i-1, xs) else xxs;
```

[8]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.

Infix változatát pl. így definiálhatjuk:

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

Itt is, akárcsak a `take`-nél, az eredménylistát meg kell fordítani, ha jobbrekurzív változatot írunk.

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!

```
fun flat [] = []
  | flat (ls::lss) = ls @ flat lss;
```

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

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

```
(* 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 *kivételkezelés* 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;
```

Jobbrekurzív változata segédargumentumokban gyűjti külön-külön a két listát, de sajnos megfordítva, ezért az eredménylistákat még meg kell fordítani:

```
(* 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);
```

`flat`, `combine` és `split` `concat`, `zip`, ill. `unzip` néven megtalálható a `List` könyvtárban.

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

A `setof` függvénynek 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.[9] A következő öt halmazműveletet definiáljuk:

- 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 ([], _) = [];
```

[9] Az SML Alapkönyvtárban több struktúra is van halmazok kezelésére.

```
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:

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

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

A 4.1.1. szakaszban már találkoztunk a *type* típusdeklarációval. Egyszerűsített szintaxisa a következő:

```
type newtyp = typexp
```

ahol *`newtyp`* az – esetleg típusváltozókat is tartalmazó – új neve, szinonimája a *`typexp`* típuskifejezéssel megadott, már ismert típusnak.

Tudjuk, hogy a `type` kulcsszóval bevezetett típusdeklaráció *gyenge absztrakció*, hiszen csak *új nevet* ad egy már *létező* adattípusnak, nem hoz létre új adattípust.

Új adattípus létrehozására a sokféleképpen használható `datatype` deklaráció használható, amelyet az SML modulszerkezetébe rejtve *erős absztrakciót* valósíthatunk meg. Ebben a fejezetben a `datatype` deklarációt ismertetjük. Egyszerűsített szintaxisa a következő:

```
datatype newtyp = datconcon | ... | datconcon |
                  datconfun of typ | ... | datconfun of typ
```

ahol *`newtyp`* az – esetleg típusváltozókat is tartalmazó – neve a nulla vagy több *`datconcon`* *adatkonstruktorállandóval* és nulla vagy több *`datconfun`* *adatkonstruktorfüggvénnyel* leírt új adattípusnak, ahol *`typ`* az adatkonstruktorfüggvény paramétere.

8.1. Felsorolásos típus adatkonstruktorállandókkal

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

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

A `degree` típusnak ebben a példában csak *adatkonstruktorállandói* vannak. Az adatkonstruktoroknak is van típusuk. Ebben a példában az összes adatkonstruktorállandó *degree* típusú.

```
- datatype degree = Duke | Marquis | Earl | Viscount | Baron;
> New type names: =degree
  datatype degree =
  (degree,
   {con Baron : degree,
    con Duke : degree,
    con Earl : degree,
    con Marquis : degree,
```

```
  | 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
```

8.2. Felsorolásos típus adatkonstruktorfüggvényekkel

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

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

Király csak egy van, ezért definiálhattuk `King`-et adatkonstruktorállandóként. A főnemest nemesi címe (`string`), birtokának neve (`string`) és sorszáma (`int`), a lovagot és a parasztot csupán a neve (`string`) azonosítja, ezért definiáltuk `Peer`-t, `Knight`-ot és `Peasant`-ot – paraméteres – adatkonstruktorfüggvényként.

Ebben a példában az adatkonstruktorok típusa a következő:

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

Jól látszik, hogy `Peer`, `Knight` és `Peasant` valóban függvények – adatkonstruktorfüggvények.

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

```
  | 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:

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

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
```

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

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

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

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

8.4. A `case`-kifejezés

A `case`-kifejezés szerkezete emlékeztet a `datatype`-deklaráció szerkezetére, ezért foglalkozunk vele ebben a fejezetben.

Egyszerűsített 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.*

9.1. Az `fn` jelölés

A 3.1.2. szakaszban már találkoztunk a (gyakran *lambdának* ejtett) `fn` kulcsszóval. Névtelen függvény például az `fn x => E` *függvénykifejezés,* ahol az `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 ily 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"
```



```
*)
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 formázó 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.

```
- 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. Magasabbrendű függvények

Magasabbrendű függvénynek (angolul *higher-order function* vagy *functional)* az olyan függvényt nevezzük, amelynek egy vagy több argumentuma és/vagy az eredménye függvény.

Minden részlegesen alkalmazható függvény magasabbrendű is, hiszen legalább két argumentumuk van (mert különben nem lehetnének részlegesen alkalmazhatók), és ha nem az összes argumentumukra alkalmazzuk őket, akkor függvényt adnak eredményül.

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

(* secr f y x = f lekötött jobb oldali argumentummal
   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. Néhány példa 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
```

9.3.2. 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 $\circ$ 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 (sqrt o real) 10
```

9.3.3. curry és uncurry

Könnyen definiálható egy-egy olyan függvény, amelyik egy párra alkalmazható függvényt részlegesen alkalmazható függvénnyé, ill. egy részlegesen alkalmazható függvényt párra alkalmazható függvénnyé alakít.

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

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

Egy-egy példa `curry` és `uncurry` alkalmazására:

```
- val plus = curry op+;
> val plus = fn : int -> int -> int
- plus 4 5;
> val it = 9 : int

- val add = uncurry plus;
> val add = fn : int * int -> int
- add(4, 5);
> val it = 9 : int
```

9.3.4. map és filter

A `map` egy paraméterként átadott függvényt alkalmaz egy lista minden elemére. Eredménye az eredeti listával megegyező hosszúságú, az elemek eredeti sorrendjét megőrző lista.

A `filter` egy listából összegyűjti azokat az elemeket, amelyek a paraméterként átadott *predikátumot* kielégítik. Eredménye az eredeti listánál esetleg rövidebb, az elemek eredeti sorrendjét megőrző lista.

[1] `sqrt` gyanánt vagy a 6.4. szakaszban definiált `sqroot`, vagy a `Math` könyvtárbeli `Math.sqrt` függvény használható.

Nézzünk egy-egy 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
```

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: ∀i,j • i ≠ j, s_i ∈ ss, s_j ∈ S • s_i ≠ s_j,∀i,j • i ≠ j, t_i ∈ T, t_j ∈ T • t_i ≠ t_j

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

A magyarázatban a `PRE` szó *előfeltételt* (precondition) jelöl: az utána álló logikai állításnak *a függvény kiértékelése előtt* teljesülnie kell ahhoz, hogy a függvény helyes értéket adjon eredményül.

Az `isMem` függvényt a 4.3.2. szakaszban *infix* operátorként definiáltuk.

9.3.4.1. Gyakorló feladat

Mi a következő függvénykifejezések típusa és mi a kiértékelésük eredménye, ha az `[["abc","def"], ["mnoprq"], ["","xy"]]` listára alkalmazzuk őket?

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

9.3.5. 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 [];
```

`exists` és `forall` a matematikai 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 4.3.2. szakaszban 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.

`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.[3] 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

(* 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` nem az: a veremben tárolja a listaelemeket, 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;
```

[2] Ezért szokták `reduce` néven is definiálni a kifejezéseket jobbról balra haladva egyszerűsítő `foldr` függvényt.
[3] Mindkettő belső függvény az SML-ben.

zására:

```
- flat [[1,2,3],[4,5],[6,7,8,9]];
> val it = [1, 2, 3, 4, 5, 6, 7, 8, 9] : int list
- (foldl op@ []) [[1,2,3],[4,5],[6,7,8,9]];
> val it = [6, 7, 8, 9, 4, 5, 1, 2, 3]
```

A `length` függvény egy iteratív változata (az `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
```

9.3.8. `repeat`

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:

`map` újradefiniálásához 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)
```

Mitagadás, `uncurry(curry op:: o f)` elég ronda 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
```

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.

10.1. 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` *kivételállandóval* (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.

10.2. 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.

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`

10.3. 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, . . ., 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.

10.4. 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.

```
> 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
```

```
[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"]
```

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:

bintree1.eps

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)
   )
 )
```


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
```

11.1. 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.

```
            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
```

`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 eléggé 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))
```

```
                    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 (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)
```

11.4. 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.

```
(* 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.

11.5. 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)
```

```
(* 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)
```

Ebben a fejezetben rendezőalgoritmusok néhány megvalósítását mutatjuk be SML-ben: a beszúró rendezést (`inssortr`, `inssortl`), a gyorsrendezést (`quicksort`), a fölülről lefelé haladó és az alulról fölfelé haladó összefésülő rendezést (`tmsort` és `bmsort`), valamint a simarendezést (`smsort`).

12.1. 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]
```

A `PRE` szó után a függvény helyes működésének előfeltételét adjuk meg (vö. 9.3.4. szakasz). `inssortr`-rel rekurzívan rendezzük a lista maradékát; végrehajtási ideje $O(n^2)$:

```
(* inssortr xs = xs elemeinek a <= reláció szerint rendezett listája
   inssortr : real list -> real list
*)
fun inssortr (x::xs) = ins(x, inssortr xs)
  | inssortr [] = []
```

12.1.1. Generikus megoldások

`inssortr` fenti változata csak `real list` típusú listák rendezésére használható. Azért nem politípusú, mert az elemek beszúrására használt `ins` függvényben a <= reláció sem az. `ins` és vele együtt `inssortr` csak úgy tehető politípusúvá, ha nem építjük beléjük a <= műveletet, hanem paraméterként adjuk át nekik. Az olyan függvényt, amelyet egy monotípusú (azaz nem politípusú) függvény paraméterként való átadásával teszünk politípusúvá, *generikus* függvénynek nevezzük.

Ha tehát a következő elemet a helyére rakó függvényt (az alábbi változatban `f`-et) paraméterként adjuk át, az `inssortr` tetszőleges típusú adatok rendezésére lesz használható:

```
(* inssortr f xs = xs elemeinek az f reláció szerint rendezett listája
   inssortr : ('a * 'b list -> 'b list) -> 'a list -> 'b list
*)
fun inssortr f (x::xs) = f(x, inssortr f xs)
  | inssortr _ [] = []
```


Ezzel `inssortr` egy újabb változata:

```
(* inssortr cmp xs = xs elemeinek a cmp reláció szerint rendezett
                     listája
   inssortr : ('a * 'a -> bool) -> 'a list -> 'a list
*)
fun inssortr cmp (x::xs) = ins cmp (x, inssortr cmp xs)
  | inssortr _ [] = [];
```

Nézzünk két példát `inssortr` alkalmazására:

```
inssortr op <= [5, 3, 7, 5, 9];
> val it = [3, 5, 5, 7, 9] : int list

inssortr op >= [5, 3, 7, 5, 9];
> val it = [9, 7, 5, 5, 3] : int list
```

Mint tudjuk, az `op` szócska és a műveleti jel közötti szóköz elmaradhat. `inssortr` 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);
```

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

```
inssortr leStrPair [("Vas", "Huba"), ("Kő", "Tamás")];
> val it = [("Kő", "Tamás"), ("Vas", "Huba")] : (string * string) list

inssortr leStrPair [("Kő", "Tamás"), ("Kő", "Huba")];
> val it = [("Kő", "Huba"), ("Kő", "Tamás")] : (string * string) list

inssortr leStrPair [("Vas", "Huba"), ("Kő", "Huba")];
> val it = [("Kő", "Huba"), ("Vas", "Huba")] : (string * string) list
```

```
        in
            sort xs []
        end;
```

12.1.2. Beszúró rendezés `foldr`-rel és `foldl`-lel

A második argumentumát gyűjtőargumentumként használó `foldl` sokkal kisebb vermet használ, mint `foldr`, ezért `inssortl` hosszabb listák rendezésére alkalmas.

```
fun inssortr cmp  = foldr (ins cmp) [];
fun inssortl cmp = foldl (ins cmp) [];
```

12.1.3. A futási idők összehasonlítása

1 és −1 különbségű számtani sorozatok, valamint álvéletlenszámokból álló sorozatok rendezésének futási idejét fogjuk megmérni. Egyesével növekvő egészlistát ad eredményül 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;
```

Álvéletlen eloszlású listát állít elő a `Random` könyvtárbeli `rangelist` függvény.[1] A futási idő méréséhez 5000 elemet tartalmazó listákat készítünk:

```
load "Random";
val rs5 = Random.rangelist (1, 100000) (5000, Random.newgen());
val is5 = 1 -- 5000;
val ds5 = rev is5;
val rs150 = Random.rangelist (1, 1000000) (150000, Random.newgen());
val is150 = 1 -- 150000;
val ds150 = rev is150;
```

[1] `Random` csak mosml-ben használható, nem része az SML Alapkönyvtárnak. Álvéletlen számok előállíthatók a 13.5.1. szakaszban bemutatott algoritmusssal.

```
fun runtime2 t xs =
    let
        val t1 = runtime ("with inssortr, orig=" ^ t) inssortr op>= xs
        val t2 = runtime ("with inssortl, orig=" ^ t) inssortl op>= xs
    in
        print(t1 ^ t2)
    end;

runtime2 "incr" is5;
  Int sort with with inssortr, orig=incr, length=5000, time=6.55s
  Int sort with with inssortl, orig=incr, length=5000, time=0.00s


runtime2 "decr" ds5;
  Int sort with with inssortr, orig=decr, length=5000, time=0.00s
  Int sort with with inssortl, orig=decr, length=5000, time=5.75s


runtime2 "rand" rs5;
  Int sort with with inssortr, orig=rand, length=5000, time=2.85s
  Int sort with with inssortl, orig=rand, length=5000, time=2.48s
```

A mérési eredmények összevetéséből jól látható, hogy a két változat futási ideje között csak kis különbség van a jobbrekurzív változat javára. A különbség a *veremhasználat hatékonyságában* van: a jobbrekurzív `inssortl` által létrehozott iteratív processz nagyságrenddel hosszabb listákat képes kezelni a nemjobbrekurzív `inssortr` által létrehozott rekurzív processznél.

Jól mutatják ezt az alábbi futási eredmények, amelyekben a futási idő minimalizálása érdekében már eleve megfelelően rendezett listákat adunk át a két függvénynek.

```
print
  (runtime "with inssortr, orig=decr" inssortr op>= (rev(1--154321)));
! Uncaught exception:
! Out_of_memory


print (runtime "with inssortl, orig=incr" inssortl op>= (1--1654321));
  Int sort with with inssortl, orig=incr, length=1654321, time=1.47s
```

```
fun quicksort cmp xs =
      let (* qs ys = a rekurzívan particionált ys elemeinek cmp
                     szerint rendezett listája
             qs : 'a list -> 'a list
          *)
          fun qs (m::ys) = partition(m, ys, ([], []))
            | qs ys = ys
          (* partition (m, xs, (ls, rs)) = a cmp(hd x, m) relációt
                  kielégítő ls és a not(cmp(hd x, m)) relációt
                  kielégítő rs részlistákkal rekurzívan particionált
                  ys elemeinek cmp szerint rendezett listája
             partition : 'a * 'a list -> 'a list * ('a list * 'a list)
          *)
          and partition (m, x::xs, (ls, rs)) =
                if cmp(x, m)
                then partition(m, xs, (x::ls, rs))
                else partition(m, xs, (ls, x::rs))
            | partition (m, [], (ls, rs)) = qs ls @ (m::qs rs)
      in
          qs xs
      end;
```

`partition` iteratív függvény, amely két eredménylistát épít fel, `ls`-t és `rs`-t.

```
print(runtime "with quicksort, orig=incr" quicksort op> is5);
  Int sort with with quicksort, orig=incr, length=5000, time=7.91s


print(runtime "with quicksort, orig=decr" quicksort op> ds5);
  Int sort with with quicksort, orig=decr, length=5000, time=8.16s


print(runtime "with quicksort, orig=rand" quicksort op> rs5);
  Int sort with with quicksort, orig=rand, length=5000, time=0.05s
```

A gyorsrendezés és a beszúró rendezések mérési eredményeit összevetve látható, hogy `quicksort`-nak ez a változata elég gyors rendezetlen listákra, de – a mediánválasztás módja miatt – lassú rendezett listákra, ui. ilyenkor az n hosszú listát minden lépésben egy egyelemű és egy $n-1$ elemű részlistára bontjuk fel.

A `quicksortm` a lista közepéről választja a mediánt. A szükséges műveletek (`length`, `div`, `List.take`, `List.drop`, `@`) költsége elég nagy lehet egy lista fejre és farokra bontásának költségéhez képest.

```
                partition(hd rs, ls @ tl rs, ([], []))
            end
          | qs ys = ys
      (* partition (m, xs, (ls, rs)) = a cmp(hd x, m) relációt
              kielégítő ls és a not(cmp(hd x, m)) relációt
              kielégítő rs részlistákkal rekurzívan particionált
              ys elemeinek cmp szerint rendezett listája
         partition : 'a * 'a list -> 'a list * ('a list * 'a list)
      *)
      and partition (m, x::xs, (ls, rs)) =
            if cmp(x, m)
            then partition(m, xs, (x::ls, rs))
            else partition(m, xs, (ls, x::rs))
        | partition (m, [], (ls, rs)) = qs ls @ (m::qs rs)
  in
      qs xs
  end;


print(runtime "with quicksortm, orig=incr" quicksortm op> is5);
  Int sort with with quicksortm, orig=incr, length=5000, time=0.02s


print(runtime "with quicksortm, orig=decr" quicksortm op> ds5);
  Int sort with with quicksortm, orig=decr, length=5000, time=0.02s


print(runtime "with quicksortm, orig=rand" quicksortm op> rs5);
  Int sort with with quicksortm, orig=rand, length=5000, time=0.04s
```

A mérési eredmények azt mutatják, hogy az extra műveletek költsége megtérül azáltal, hogy közel egyforma hosszú részlistákkal dolgozunk.

A `partition` függvényben a `@` művelet is kiküszöbölhető újabb segédargumentum bevezetésével. Az algoritmus veremkezelésének hatékonysága tovább javítható pl. úgy, hogy az m értékű elemeket egy harmadik listában gyűjtjük vagy megszámláljuk, amikor a listát két részre bontjuk. Mint korábban láttuk, ezek a változtatások csak kisebb mértékben gyorsítják a programot, ugyanakkor nehezebbé teszik a megértését és helyességének belátását.

---

vagy *réteges minta* (...as...) alkalmazásával:

```
fun merge cmp (xxs as x::xs, yys as y::ys)= if cmp (x, y)
                                            then x::merge(xs, yys)
                                            else y::merge(xxs, ys)
  | merge ...
```

Hatékonyságromlást okoz, hogy a részeredményeket itt is a veremben tároljuk. Jobbrekurzió alkalmazása esetén meg kell fordítani az eredménylistát. Ezt gyakorló feladatként meghagyjuk az olvasónak.

12.3.1. Fölülről lefelé haladó összefésülő rendezés

A fölülről lefelé haladó összefésülő rendezés (*top-down* merge sort) is akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük.

```
(* tmsort cmp xs = xs elemeinek a cmp reláció szerint rendezett listája
   tmsort : ('a * 'a -> bool) -> 'a list -> 'a list
*)
fun tmsort cmp xs =
      let
          val h = length xs
          val k = h div 2
      in
          if h > 1
          then merge cmp (tmsort(List.take(xs, k)),
                          tmsort(List.drop(xs, k)))
          else xs
      end;
```

A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség egy n hosszú lista rendezéséhez. A módszer egyszerű, és elég gyors is, amint az alábbi mérési eredmények mutatják:

```
print(runtime "with tmsort, orig=incr" tmsort op> is5);
  Int sort with with tmsort, orig=incr, length=5000, time=0.03s

print(runtime "with tmsort, orig=decr" tmsort op> ds5);
  Int sort with with tmsort, orig=decr, length=5000, time=0.03s

print(runtime "with tmsort, orig=rand" tmsort op> rs5);
  Int sort with with tmsort, orig=rand, length=5000, time=0.04s
```

A B C D E F G H 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 cmp xs = xs elemeinek a cmp reláció szerint rendezett listája
   bmsort : ('a * 'a -> bool) -> 'a list -> 'a list
*)
fun bmsort cmp xs = sorting cmp (xs, [], 0)
```

bmsort a sorting segédfüggvényt használja. sorting második argumentuma egy olyan hármas, amelynek első tagja a rendezendő lista, második tagja a már rendezett részlistákat gyűjti, (kezdőértéke []), harmadik tagja 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 cmp (xs, lss, k) = az xs elemeinek a cmp reláció szerint
                              rendezett listája; az algoritmus a még rendezetlen
                              xs lista elemeit berakja a k elemet tartalmazó,
                              cmp szerint folyamatosan rendezett lss listába
   sorting : ('a * 'a -> bool) -> 'a list * 'a list list * int ->
                                                                 'a list
   PRE: k >= 0
*)
and sorting cmp (x::xs, lss, k) =
                sorting cmp (xs, mergepairs cmp ([x]::lss, k+1), k+1)
  | sorting cmp ([], lss, k) = hd(mergepairs cmp (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 cmp (llss, n)= az n elemet tartalmazó, már rendezett lss
                             lista első két részlistáját, ha egyforma
```

A függvények működését az alábbi példán követhetjük. A kezdőhívás legyen

```
bmsort op> [1,2,3,4,5,6,7,8,9] =
                                sorting op> ([1,2,3,4,5,6,7,8,9], [], 0);
```

Amíg `sorting` második argumentumának első tagja a nem üres `x::xs` lista, `sorting` saját magát hívja meg. A rekurzív hívás második argumentumának első tagja a lépésenként egyre rövidülő `xs` lista, harmadik tagja (`k+1`) a már feldolgozott listaelemek száma, második tagja pedig a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`.

A következő táblázatos elrendezés `mergepairs` második argumentumának mindkét tagját, valamint a rekurzív `sorting` hívás msodik argumentumának itt `j`-vel jelölt harmadik tagját, `k+1`-et és bináris számként `k`-t mutatja lépésről lépésre. (Ne feledjük, hogy `mergepairs` második argumentumának listák listája az első tagja.) 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		m1
[[2],[1]]	2	2		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]]				

```
print(runtime "with bmsort, orig=incr" bmsort op> is5);
  Int sort with with bmsort, orig=incr, length=5000, time=0.02s

print(runtime "with bmsort, orig=decr" bmsort op> ds5);
  Int sort with with bmsort, orig=decr, length=5000, time=0.02s

print(runtime "with bmsort, orig=rand" bmsort op> rs5);
  Int sort with with bmsort, orig=rand, length=5000, time=0.04s
```

`bmsort` egészen jól elboldogul sokkal hoszabb listákkal is:

```
print(runtime "with bmsort, orig=incr" bmsort op> is150);
  Int sort with with bmsort, orig=incr, length=150000, time=0.76s

print(runtime "with bmsort, orig=decr" bmsort op> ds150);
  Int sort with with bmsort, orig=decr, length=150000, time=1.04s

print(runtime "with bmsort, orig=rand" bmsort op> rs150);
  Int sort with with bmsort, orig=rand, length=150000, time=1.97s
```

12.4. 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 cmp (run, xs) = (r, rs), ahol r xs egy cmp szerint növekvő
                          kezdőfutama, rs pedig xs maradéka
   nextrun : ('a * 'a -> bool) -> 'a list * 'a list ->
                                                    'a list * 'a list
*)
fun nextrun cmp (run, x::xs) = if not(cmp(x, hd run))
                               then (rev run, x::xs)
                               else nextrun cmp (x::run, xs)
  | nextrun cmp (run, []) = (rev run, [])
```

```
          val (run, tail) = nextrun cmp ([x], xs)
       in
          smsorting cmp (tail, mergepairs cmp (run::lss, k+1), k+1)
       end
   | smsorting cmp ([], lss, k) = hd(mergepairs cmp(lss, 0))

(* smsort cmp xs = xs elemeinek cmp szerint rendezett listája
   smsort : 'a list -> 'a list
*)
fun smsort cmp xs = smsorting cmp (xs, [], 0);
```

Megmérjük `smsort` futási idejét is:

```
print(runtime "with smsort, orig=incr" smsort op> is150);
  Int sort with with smsort, orig=incr, length=150000, time=0.05s


print(runtime "with smsort, orig=decr" smsort op> ds150);
  Int sort with with smsort, orig=decr, length=150000, time=1.24s


print(runtime "with smsort, orig=rand" smsort op> rs150);
  Int sort with with smsort, orig=decr, length=150000, time=2.06s
```

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.

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.

Az SML egy bővítése, az Alice is alapvetően mohó kiértékelésű, de lehet vele *lusta* kiértékelésű kifejezést (rövidebben: lusta kifejezést) létrehozni. Ebben a fejezetben a példákban az Alice-nyelv bővített szintaxisát használjuk, és az Alice válaszait és üzeneteit adjuk meg.

13.1. Lusta kifejezés és függvény létrehozása

A lusta kiértékelést a `lazy` kulcsszóval írhatjuk elő egy Alice-nyelvű programban.

```
val zs = lazy [1,2,3,4,5,6,7,8,9];
val zs : int list = _lazy
```

A lusta kifejezést az Alice csak akkor értékeli ki, ha *szükség* van rá, azaz csak akkor, ha egy mohó kiértékelésű művelet argumentumaként használjuk. Mindig mohó kiértékelésűek a következők:

- mintaillesztésben a vizsgált érték,
- függvényalkalmazásban a függvényérték,
- kivétel jelzésében a kivételt alkotó érték,
- primitív műveletben (pl. `op+`, `op=`) az olyan operandus, amelyhez a műveletnek hozzá kell férnie.

A lusta kifejezés lusta marad mindaddig, amíg az Alice nem értékeli ki, de ha egyszer kiértékelte, a kifejezés lusta volta megszűnik. A következő példában a mohó kiértékelésű `hd` függvényt (rövidebben: mohó függvényt) alkalmazzuk a lusta `zs`-re.

```
zs;
val it : int list = _lazy

hd zs;
val it : int = 1

zs;
val it : int list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A példából is látható, hogy ha egyszer az Alice egy lusta kifejezést kiértékelt, az lusta voltát elveszti, a továbbiakban az Alice az egyszer kiszámított és eltárolt értéket adja eredményül. Látni fogjuk, hogy ez nemcsak a teljes lusta kifejezésre, hanem annak valamely részkifejezésére is igaz.

A `hd` és a `tl` lusta változatát így deklarálhatjuk (a függvénynevek végén álló *z* betűvel szokás a függvények lusta – lazy – voltára emlékeztetni):

```
0 + headz zs;
val it : int = 1

zs;
val it : int list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

tailz zs;
val it : int list = _lazy

[] @ tailz zs;
val it : int list = _lazy

tailz zs @ [];
val it : int list = [2, 3, 4, 5, 6, 7, 8, 9]
```

Vegyük észre, hogy a `[] @ tailz zs` kifejezésben az eredmény előállításához a `@` operátornak nincs szüksége jobb oldali operandusának a kiszámítására, ezért az eredmény is lusta kiértékelésű lesz.

13.2. Lusta lista

Az Alice-ben könnyű lusta listát létrehozni. A lusta lista olyan *nem korlátos méretű* lista, amelynek előnyős tulajdonságai mellett hátrányai, veszélyei is vannak, pl.

- egy lusta lista *bármely részét* megjeleníthetjük, de *sohasem az egészet;*
- két lusta lista elemeiből páronként képezhetünk egy harmadikat, de *nem számíthatjuk ki* egy lusta lista *elemeinek összegét,* nem kereshetjük meg benne *a legkisebbet,* nem fordíthatjuk meg az *elemek sorrendjét* stb.;
- egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény *tetszőleges véges része véges idő alatt* előáll.

Most a korábban megismert `from` függvény lusta változatát mutatjuk be. A `fromz k` függvényalkalmazás olyan lusta listát hoz létre, amelynek az elemei `fromz k`-tól kezdve egyesével növekvő számtani sorozatot alkotnak.

```
fun lazy fromz k = k :: fromz(k+1);
val fromz : int -> int list = _fn
```

```
xs;
val it : int list = 3 :: _lazy
```

Lássunk most néhány további példát `headz` és `tailz` használatára.

```
headz(fromz 3);
val it : int = _lazy

headz(fromz 3) + 0;
val it : int = 3

tailz(fromz 3);
val it : int list = _lazy

headz(tailz(fromz 3)) + 0;
val it : int = 4
```

De vigyázzunk! Veremtúlcsorduláshoz vezet a következő kifejezés kiértékelése:

```
tailz(fromz 3) @ [];
```

A `List.take` és `List.drop` függvényt lusta listára is alkalmazhatjuk a lusta lista egy részének kiértékelésére.

```
List.take(fromz 1, 5);
val it : int list = [1, 2, 3, 4, 5]

List.drop(fromz 1, 5);
val it : int list = _lazy

hd(List.drop(fromz 1, 5));
val it : int = 6
```

```
List.take(squarez(fromz 1), 10);
val it : int list = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Két lusta lista hasonlóan adható össze:

```
fun lazy addz (x::xs, y::ys) = x+y :: addz(xs, ys)
       | addz _ = [];
val addz : int list * int list -> int list = _fn

addz(fromz 1000, squarez(fromz 1));
val it : int list = _lazy

List.take(addz(fromz 1000, squarez(fromz 1)), 7);
val it : int list = [1001, 1005, 1011, 1019, 1029, 1041, 1055]
```

Az `appendz` függvény addig nem nyúl `ys`-hez, amíg `xs` ki nem ürül – vagyis csak akkor nyúl hozzá, ha `xs` korlátos.

```
fun lazy appendz (x::xs, ys) = x :: appendz (xs, ys)
       | appendz ([], ys) = ys;
val appendz : 'a list * 'a list -> 'a list = _fn

appendz([1,2,3],[4,5,6]);
val it : int list = _lazy

appendz([1,2,3],[4,5,6]) @ [];
val it : int list = [1, 2, 3, 4, 5, 6]

List.take(appendz([1,2,3], fromz 10), 7);
val it : int list = [1, 2, 3, 10, 11, 12, 13]

List.take(appendz(fromz 10, [4,5,6]), 7);
val it : int list = [10, 11, 12, 13, 14, 15, 16]
```

Az előző szakaszban definiált `squarez`-t egyszerű felírni `mapz`-vel:

```
val squarez = mapz (fn x => x*x);
val squarez : int list -> int list = _fn
```

Valós számokra:

```
val squarerz = mapz (fn x : real => x*x);
val squarerz : real list -> real list = _fn
```

Most olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egészek vannak:

```
val sevens = filterz (fn n => n mod 10 = 7) (fromz 50);
val sevens : int list = _lazy

List.take(sevens, 8);
val it : int list = [57, 67, 77, 87, 97, 107, 117, 127]
```

Az ugyancsak magasabbrendű `iteratez` függvény – a `fromz` egy általánosítása – a következő sorozatot állítja elő (v.ö. a korábban definiált `repeat`-tel):

```
fun lazy iteratez f x = x :: iteratez f (f x);
val iteratez : ('a -> 'a) -> 'a -> 'a list = _fn
```

Egy példa `iteratez` alkalmazására:

```
val zs = iteratez (fn x => x / 2.0) 1.0;
val zs : real list = _lazy

List.take(zs, 5);
val it : real list = [1.0, 0.5, 0.25, 0.125, 0.0625]
```

`fromz`-t az `iteratez`-vel így definiálhatjuk:

```
val fromz = iteratez (fn k => k+1);
val fromz : int -> int list = _fn

List.take(fromz 3, 6);
val it : int list = [3, 4, 5, 6, 7, 8]
```

```
                val t = a * seed
            in
                t - real(floor(t/m))*m
            end
in
    fun randomz s = mapz (fn x => x/m) (iteratez nextrandom s)
end;
val randomz : real -> real list = _fn
```

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 lusta lista előállítására `iteratez`-t a `nextrandom`-ra és a `seed` valós számmá átalakított kezdőértékére alkalmazzuk. `mapz` gondoskodik arról, hogy a lusta listában minden értéket elosszunk `m`-mel, és így `randomz` 1.0-nél kisebb nemnegatív értékeket adjon eredményül. Látható, hogy a lusta lista a megvalósítás részleteit szépen elrejti a felhasználó elől.[1]

Az előállított álvéletlenszámok tehát 1.0-nél kisebb nemnegatív valós számok; `mapz`-vel alakíthatjuk át őket 0 és 9 közötti egészekké:

```
val rs = mapz (floor o (fn x => 10.0 * x)) (randomz 1.0);
val rs : Int.int list = _lazy

List.take(rs, 9);
val it : Int.int list = [0, 0, 1, 7, 4, 5, 2, 0, 6]
```

13.5.2. Prímszámok

Következő programunk az *eratosztenészi szita* megvalósítása. Idézzük föl az ismert algoritmust:

1. Vegyük az egészek 2-vel kezdődő sorozatát: $2, 3, 4, 5, 6, 7, \ldots$
2. Töröljük az összes 2-vel osztható számot: $3, 5, 7, 9, 11, \ldots$
3. Töröljük az összes 3-mal osztható számot: $5, 7, 11, 13, 17, 19, \ldots$
4. Töröljük az összes 5-tel osztható számot: $7, 11, 13, 17, 19, \ldots$

[1] Ezt az algoritmust Park és Miller dolgozta ki 1988-ban. Helyes működéséhez a mantisszának 46 bitesnek (!) kell lennie. Az ix86 architektúrákon az smlnj mantisszája 52 bites, a többi SML-megvalósításé jóval rövidebb. — Kevésbé jó statisztikai tulajdonságú álvéletlenszámokat kaphatunk, ha `a`-nak és `m`-nek kisebb relatív prímeket választunk.

```
val sievez : int list -> int list = _fn

fun lazy sievez (p::ps) = p :: sievez(siftz p ps);
1.9-1.49: warning: match is not exhaustive, because e.g.
    nil
is not covered
val sievez : int list -> int list = _fn

val primes = sievez(fromz 2);
val primes : int list = _lazy

List.take(primes, 11);
val it : int list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

13.5.3. Gyökvonás

Számítsuk ki egy szám négyzetgyökét Newton-Raphson módszerrel, lusta lista alkalmazásával. (A feladat egy megoldását már ismertettük a 6.4. szakaszban.)

```
fun nextapprox a x = (a/x+x)/2.0;
val nextapprox : real -> real -> real = _fn
```

`nextapprox` x_k-ból a gyök egy x_{k+1} közelítését számítja ki az

$$x_{k+1} = \frac{\frac{a}{x_k} + x_k}{2}$$

képlet alapján. A befejeződés megállapítására egyszerű tesztet írunk:

```
exception Impossible;
fun within (eps : real) (x::(yys as y::ys)) =
      if abs(x-y) <= eps
      then y
      else within eps yys
  | within _ _ = raise Impossible;
val within : real -> real list -> real = _fn
```

A második klózt csak azért írtuk, hogy ne kapjunk figyelmeztetést a lefedetlen esetek miatt: mivel a `within` függvényt lusta listára fogjuk alkalmazni, ennek a klóznak a kiértékelésére sohasem kerül sor.

```
fun approx a
      let
          fun nextapprox x = (a/x+x)/2.0
      in
          iteratez nextapprox 1.0
      end;
val approxz : real -> real list = _fn
```

Ezzel a `qroot` függvény egy tisztább változata:

```
val qroot = within 1e~6 o approxz;
val qroot : real -> real = _fn


qroot 5.0;
val it : real = 2.23607
```

13.6. Lusta listák listája és egymásba ékelése

Legyen xs és ys egy-egy lusta lista. Képezzünk új lusta listát az (x_i, y_j) párokból, ahol $x_i \in xs$ és $y_j \in ys$! 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.

13.6.1. 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 : 'a -> 'b -> 'a * 'b = _fn

fun flat xss = foldl op@ [] xss;
val flat : 'a list list -> 'a list = _fn
```

A részlegesen alkalmazható `(pair x)` függvény a rögzített `x` értékből és a `(pair x)` függvény argumentumából képez párt. Ha ezt a függvényt `map`-pel az `ys` lista elemeire alkalmazzuk:

```
map (pair x) ys
```

```
fun pairss xs ys = map (fn x => map (pair x) ys) xs;
val pairss : 'a list -> 'b list -> ('a * 'b) list list = _fn
```

`flat` elvégzi a szükséges simítást a kitűzött feladatot megoldó `pairs` függvényben:

```
fun pairs xs ys = flat (map (fn x => map (pair x) ys) xs);
val pairs : 'a list -> 'b list -> ('a * 'b) list = _fn
```

13.6.2. Keresztszorzatokból álló lusta lista

Térjünk vissza az eredeti feladathoz, vagyis legyen most xs és ys egy-egy lusta lista! Képezzünk új lusta listát az (x_i, y_j) párokból, ahol $x_i \in xs$ és $y_j \in ys$!

`pairss` lusta megfelelője `pairssz`:

```
fun pairssz xs ys = mapz (fn x => mapz (pair x) ys) xs;
val pairssz : 'a list -> 'b list -> ('a * 'b) list list = _fn
```

Példa `pairssz` alkalmazására:

```
val pss = pairssz (fromz 30) primes;
val pss : (int * int) list list = _lazy
```

A `pss` lusta listának, mint tudjuk, csak *véges része* írható ki. A következő `takeRect` függvény a bal felső saroktól számított első `m` sorból és `n` oszlopból álló *téglalapot* jeleníti meg az argumentumként átadott `xss` lusta listából:

```
fun takeRect (xss, (m, n)) =
      map (fn y => List.take(y, n)) (List.take(xss, m));
val takeRect : 'a list list * (int * int) -> 'a list list = _fn
```

Állítsunk elő például párok lusta listájából álló olyan lusta listát, amelyben a párok első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:

```
takeRect(pss, (3, 5));
val it : (int * int) list list =
   [[(30, 2), (30, 3), (30, 5), (30, 7), (30, 11)],
    [(31, 2), (31, 3), (31, 5), (31, 7), (31, 11)],
    [(32, 2), (32, 3), (32, 5), (32, 7), (32, 11)]]

pss;
```

Vegyük észre, hogy `interleavez` a *rekurzív hívásban* váltogatja a két lusta listát. Nézzünk egy példát `interleavez` alkalmazására:

```
List.take(interleavez(fromz 0, fromz 50), 10);
val it : int list = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54]
```

Most `enumeratez` néven olyan függvényt definiálunk, amely lusta listák listájából egyetlen lusta listát állít elő. Jelöljük a kétszeres mélységű lusta lista fejét `xs`-sel, a farkát `xss`-sel, alkalmazzuk `enumeratez`-t rekurzívan `xss`-re, majd az eredményt ékeljük be `xs`-be:

```
fun lazy enumeratez [] = []
       | enumeratez (xs::xss) = interleavez(xs, enumeratez xss);
val enumeratez : 'a list list -> 'a list = _fn
```

Állítsuk elő például a pozitív egészekből álló párok egy lusta listáját!

```
val pozIntss = pairssz (fromz 1) (fromz 1);
val pozIntss : (int * int) list list = _lazy

List.take(enumeratez pozIntss, 15);
val it : (int * int) list =
   [(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)]
```

14.1. 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 legyenek, amelyek vagy csak a megadott, zárt intervallumba tartozó, vagy csak az ezen kívül eső karakterekből állnak. Használhatja a `String.tokens` és a `String.isPrefix` magasabbrendű 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
*)
```

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
             osszefuttat : 'a list * 'a list * 'a list -> 'a list
             PRE : |length xs - length ys| <= 1
         *)
          fun osszefuttat (x::xs, y::ys, zs) =
                                       osszefuttat(xs, ys, y::x::zs)
            | osszefuttat ([], [y], zs) = rev(y :: zs)
```

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;
```

Példa

```
mezok ("Ali baba + a 40 rablo", #"a", #"n") =
  ["A", "li", " ", "baba", " + ", "a", " 40 r", "abl", "o"];
```

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;
```

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 "/" = "";
```

14.3. 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
   rootname : string -> string
   PRE: s <> ""
*)
```

Megoldás

A megoldás most is nagyon egyszerű, ha a megfelelő könyvtári függvényeket használjuk.

```
rootname "///file.ext///" = "/file.ext";
rootname "/" = "/";
rootname "///" = "/";
```

14.4. Füzér egyes elemeinek azonosítása (`parPairs`)

Írjon SML-függvényt `parPairs` néven, amely az argumentumként átadott füzér egymáshoz tartozó kerek nyitó- és csukózárójeleinek sorszámából alkotott párok listáját adja eredményül, tetszőleges sorrendben! A füzér karaktereit 1-től számozzuk. Segédfüggvényt definiálhat.

A függvény specifikációja:

```
(* parPairs s = az s füzér egymáshoz tartozó kerek nyitó- és
                csukózárójeleinek szorszámából alkotott párok listája;
                a füzér karaktereit 0-tól számozzuk
   parPairs : string -> (int * int) list
*)
```

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, beírja az eredeti füzér szerinti sorszámát 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 kiütül, a sorszámpárok `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 sorszáma az eredeti
                         füzérben (a sorszámozás 0-tól kezdődik) ;
                    bs = a még le nem zárt nyitózárójelek sorszámának
```

Példák

```
parPairs "Zárójelmentes." = [];
parPairs ")" = [];
parPairs "(" = [];
parPairs "real(3*4) + (sin(0.5) - (11.4+3.4)) * 1.2" =
                              [(4, 8), (16, 20), (24, 33), (12, 34)];
```

14.5. 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!

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)
```

```
  | szomsor _ = [];
```

Példa

```
szomsor [3,2,4,9,7,1,8,5,6] = [[3,4,5,6], [7,8]];
```

14.6. 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 `O`. A függvény specifikációja:

```
(* binc ns = az ns-ben tárolt bináris számot inkrementálja
   binc : bin list -> bin list
*)
```

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, ahol a bs
                          lista feje a legkisebb helyiértékű bit, c
                          a növekmény
         bi : (bin list * bin * bin list) -> bin list
      *)
      fun bi (b::bs, O, rs) = bi(bs, O, b::rs)
        | bi (O::bs, I, rs) = bi(bs, O, I::rs)
        | bi (I::bs, I, rs) = bi(bs, I, O::rs)
        | bi ([], O, rs) = rs
        | bi ([], I, rs) = I::rs
in
      (* binc ns = az ns bináris szám inkrementáltja
```

```
                | NONE   => 0;

app load ["Int"];

(* i2b i = az i egész bináris megfelelője (0 -> O, nem 0 -> I),
           a vezető O-k nélkül
   i2b : int -> bin list
*)
fun i2b i = map (fn #"0" => O | _ => I) (explode(Int.toString i));
```

Példák

```
binc [O] = [I];
binc [I] = [I,O];
binc [I,I,I,I,I,I,I,I] = [I,O,O,O,O,O,O,O,O];
```

14.7. 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
*)
```

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)
```

```
                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)));
```

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]];
```

15.1. 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-nek és C-nek ugyanannyi A gyermeke van
   ugyanannyi : 'a fa -> bool
*)
```

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 = (b, c, a), ahol b a B és c a C csomópont A leveleinek a
          száma, a pedig 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);
```

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
```

```
ugyanannyi(B(C(B(A,A), B(A,A),B(A,A)), B(C(A,A,A), C(A,A,A)))) = true;
```

15.2. 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
*)
```

1. megoldás

A `ba` segédfüggvény az olyan `B`-ket számlálja meg, amelyeknek egyetlen utódja sem `C`.

```
(* ba f = (b, c), 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
```

---

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;
```

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;
```

15.3. Fa adott tulajdonságú részfáinak száma (`testverE`)

Írjon `testverE` néven olyan SML-függvényt, amely a

---

zettük 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;
```

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;
```

15.4. 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, harmadik eleme a harmadik szinten tárolt, legfeljebb négy elem összege s.í.t. A fa típusa legyen:

```
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
*)
```

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.

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
```

```
in
      fun szintOssz t = sO(t, [])
end;
```

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 szereplő fa és a szintenkénti összegek grafikusan:

```
             1           1
           /   \
          2     3        5
         / \   / \
        4   5 6   7     22
       / \
      8   9             17
```

15.5. 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
*)
```

```
  | 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;
```

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);
```

15.6. Kifejezésfa egyszerűsítése (`coeff`)

Tekintse az alábbi típust és adattípust:

```
type term = int * char;
datatype expr = ## of expr * term | Z;
infix 6 ##;
```

---

zés, 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;
```

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;
```

Ebben a fejezetben egy nagyobb példát mutatunk be a funkcionális programozásra SML-nyelven: fordítóprogramot[1] írunk *spl*-nyelvű programok fordítására *(spl* = Simple Programming Language). Az *spl*-nyelvet és Prolog-nyelvű fordítóprogramját D.H.D. Warren publikálta először.[2] Több oka is van annak, hogy miért választottuk ugyanazt a példát.

- Ez egy elég összetett, mégis aránylag könnyen megérthető példa, amellyel be lehet mutatni a funkcionális programozás sok szép, szokásos vagy éppen szokatlan jellemzőjét.
- Mivel a fordítóprogramokat rendszerint imperatív nyelveken, pl. assembly, C vagy C++ nyelven írják, a példa alkalmas annak érzékeltetésére, hogy az imperatív programozással szemben milyen előnyöket jelent a funkcionális programozás.
- Az érdeklődő olvasó összevetheti a funkcionális megoldást a logikaival.
- Azoknak, akik még sohasem foglalkoztak fordítóprogramokkal, a példa betekintést ad a fordítóprogramok működésébe, amely a szoftvertechnológia egyik legfejlettebb területe.

16.1. A forrásnyelv

Az *spl*-nyelv a Pascalra emlékeztető, de a Pascalnál is sokkal egyszerűbb nyelv: nincsenek benne típusok, deklarációk és eljárások, műveleteket csak egész számokkal lehet végezni, az állandók csak nemnegatív egészek lehetnek stb.

A 16.1. ábrán látható program az *spl*-nyelv összes lehetőségét bemutatja. Az `intvalsum` program azt a lehető leghosszabb $[i, j]$ zárt intervallumot határozza meg valamely $s \geq 1$ egészre, amelyre $1 \leq i \leq j$ és az s az $[i, j]$ intervallumba eső számok összegével egyenlő.[3]

16.2. A forrásnyelv konkrét szintaxisa

Az *spl*-nyelv konkrét szintaxisa EBNF (Extended Backus Normal Form) jelöléssel a 16.2. ábrán látható. A nemterminális szimbólumokat < és > csúcsos zárójelek fogják közre, a terminális szimbólumokat **félkövér**, a megjegyzéseket ún. slanted betűtípussal írjuk.

[1] Az SML-nyelven készült *splc*-fordítóprogram előző változata *A simple compiler written in SML – V3.2* címen jelent meg, ld. Hanák D. Péter: *Programozási paradigmák*. Oktatási segédlet. Standard ML. BME Számítástudományi és Információelméleti Tanszék, Budapest, 1996.

[2] Warren, D. H. D.: *Logic Programming and Compiler Writing* in *Software Practice and Experience* 10(2), pp. 97-125, 1980.

[3] A feladat SML-megoldását a 2.3. szakaszban dolgoztuk ki.


```
                write 0);
```

16.1. ábra. Az `intvalsum` program *spl*-nyelven

```
<program> ::=      <statements>
<statements> ::=   <statement> ; <statements> | <statement>
<statement> ::=    <name> := <expression> |
                   if <test> then <statement> else <statement> |
                   while <test> do <statement> |
                   read <name> |
                   write <expression> |
                   ( <statements> )
<test> ::=         <expression> <compOp> <expression>
<expression> ::=   <expression> <op2> <expr1> | <expr1>
<expr1> ::=        <expr1> <op1> <expr0> | <expr0>
<expr0> ::=        <name> | <number> | ( <expression> )
<compOp> ::=       = | < | > | =< | >= | /=
<op2> ::=          + | -
<op1> ::=          * | /
<name> ::=         betűvel kezdődő tetszőleges alfanumerikus karaktersorozat
<number> ::=       számjegyekből álló tetszőleges karaktersorozat
```

16.2. ábra. Az *spl*-nyelv konkrét szintaxisa

16.3. A célnyelv

A célnyelv egy olyan egyszerű CPU egycímű utasításkészlete, amelynek csak egyetlen regisztere van, az *akkumulátor*. Az utasításkészlet be- és kiviteli, aritmetikai, adatmozgató, valamint feltételes és feltétel nélküli vezérlésátadó utasításokból áll. Aritmetikai és adatmozgató utasítás operandusa vagy tárcím, vagy literális (állandó) lehet; az utasítás ún. *mnemonikja* az operandus fajtájától függ.

- Aritmetikai és adatmozgató utasítások tárcím-operandussal: `ADD`, `SUB`, `MUL`, `DIV`, `LOAD`, `STORE`
- Aritmetikai és adatmozgató utasítások literális operandussal: `ADDC`, `SUBC`, `MULC`, `DIVC`
- Feltétel nélküli vezérlésátadó utasítások: `JUMP`, `HALT`
- Feltételes vezérlésátadó utasítások: `JUMPEQ`, `JUMPNE`, `JUMPLT`, `JUMPGT`, `JUMPLE`, `JUMPGE`
- Be- és kiviteli utasítások: `READ`, `WRITE`

```
 5        STORE      65  h          h := 1
 6  %L2:  LOAD       65  h          while
 7        MUL        65  h            h*h
 8        DIVC        2  2              div 2
 9        SUB        64  s                < s
10        JUMPGE     15  %L1
11        LOAD       65  h            do
12        ADDC        1  1              h := h+1
13        STORE      65  h
14        JUMP        6  %L2
15  %L1:  LOAD       65  h
16        SUBC        1  1
17        STORE      65  h          h := h-1
18        LOAD       65  h
19        ADDC        1  1
20        MUL        65  h
21        DIVC        2  2
22        STORE      67  %T0        (h+1)*h/2
23        LOAD       64  s
24        SUB        67  %T0
25        STORE      66  k          k := s-(h+1)*h/2
26  %L4:  LOAD       66  k          while
27        DIV        65  h
28        MUL        65  h
29        STORE      67  %T0
30        LOAD       66  k
31        SUB        67  %T0
32        SUBC        0  0            k-k/h*h > 0
33        JUMPLE     46  %L3
34        LOAD       65  h          do
35        SUBC        1  1
36        STORE      65  h            h := h-1
37        LOAD       65  h
38        ADDC        1  1
39        MUL        65  h
40        DIVC        2  2
41        STORE      67  %T0
```

16.3. ábra. Az `intvalsum` program célnyelvre lefordított változata

```
55        WRITE      67  %T0       write k/h+h
56        JUMP       63  %L5
57  %L0:  LOADC       0  0            0
58        STORE      67  %T0
59        WRITE      67  %T0       write 0
60        LOADC       0  0            0
61        STORE      67  %T0
62        WRITE      67  %T0       write 0
63  %L5:  HALT
64  s:    BLOCK       4  4
65  h:
66  k:
67  %T0:
```

16.4. ábra. Az `intvalsum` program célnyelvre lefordított változata (folyt.)

Figyeljük meg a `%T0` átmeneti tároló bevezetését és használatát: erre olyan kifejezések kiértékelésekor van szükség, amelyekben különböző precedenciájú operátorok vagy zárójeles részkifejezések vannak. Figyeljük meg a `%L0`–`%L5` címkék előállítását és használatát is.

16.4. A fordítás folyamata

A fordító négy fő fázisban állítja elő a forrásprogramból az abszolút gépi kódú programot (ld. 16.5. ábra). A dobozok a fordítás fő fázisait, a betűpárok pedig az egyes fázisok be- és kimeneti adatait jelölik.

16.5. A forrásnyelv absztrakt szintaxisa

Az absztrakt szintaxis, amelyet a 16.6. ábra mutat EBNF-jelöléssel, nagyjából megfelel a relokálható gépi kód belső ábrázolásának. A nemterminális szimbólumokat itt is < és > csúcsos zárójelek fogják közre, a terminális szimbólumokat pedig **félkövér** betűtípussal írjuk.

16.6. A fordítóprogram építőkockái

Az *splc*-fordító számos függvényből épül fel. Ezeket a függvényeket több modulba – SML-terminológiával: *struktúrába* foglaltuk.

```
<program> ::=      <statement>
<statement> ::=    read(<name>) |
                   write(<expression>) |
                   andThen(<statement>, <statement>) |
                   assign(<name>, <expression>) |
                   while(<test>, <statement>) |
                   if(<test>, <statement>, <statement>)
<test> ::=         test(<compOp>, <expression>, <expression>)
<expression> ::=   expr(<op>, <expr>, <expr>) |
                   const(<number>) |
                   name(<name>)
<compOp> ::=       = | < | > | =< | >= | /=
<op> ::=           + | - | * | /
```

16.6. ábra. Az *spl*-nyelv absztrakt szintaxisa

A struktúra a modul ún. *implementációs része*, amelynek a külvilág számára látható felületét, ún. *interfészét* a *szignatúrája* írja le. Egy struktúra szignatúráját automatikusan előállíthatja az SML-fordító, vagy megírhatja a programozó. Az utóbbi esetben a programozó bizonyos dolgokat (függvényeket, típusokat, értékeket stb.) elrejthet a külvilág elől, miközben lokálisan, az adott struktúrán belül használhatja őket. Szükség esetén egy struktúrának egynél több szignatúrája is lehet, hogy a struktúra különféle alkalmazásaiban különféle igényeket elégíthessen ki.

A struktúráknak meghatározott sorrendben kell előfordulniuk egy SML-programban, mert a fordítás során az értékeknek és típusoknak előre definiáltnak kell lenniük, más szóval minden értéket és típust az első alkalmazása előtt definiálni vagy deklarálni kell. Az *splc*-fordító struktúráinak a sorrendje a következő:

`Symtab`, `Lexical`, `Parsefun`, `Parse`, `Encode`, `Assemble`, `Compile`

A `Symtab` struktúra, amely az *SML Basis Library* `Binarymap` struktúrájának adattípusát és függvényeit használja fel, valósítja meg a *szimbólumtáblát*. A szimbólumtábla képezi le az *spl*-programokban előforduló összes nevet (azonosítókat, literálisokat, kulcsszókat stb.) *tokenekké*, amelyekre a fordítás folyamán van szükség. Az *splc*-fordító struktúráinak többsége használja a `Symtab` struktúrát.

A `Lexical` struktúra valósítja meg a *lexikai analízis* fázisát (ld. 16.5. ábra),) amelyet gyakran *szkennelésnek* is neveznek. A lexikai analízis *tokenizálja* az *spl*-programot, azaz a forráskódban előforduló nevek sorozatát tokenek (esetünkben: egészek) listájává transzformálja, és közben felépíti a szimbólumtáblát.

A `Parse` struktúra, amely felhasználja a `Parsefun` struktúra egyszerű elemző függvényeit, végzi a *tokenizált* program szintaktikai analízisét, azaz ellenőrzi, hogy a forráskód megfelel el az *spl*-nyelv *konkrét szintaxisának*. Bemenete a tokenek (esetünkben: egészek) listája, kimenete az *elemzőfa*, amely az

A fejezet hátralévő része az *splc*-fordító forráskódját tartalmazza.

Az *splc*-fordítót az alábbi `Makefile` felhasználásával a `make` paranccsal lehet előállítani olyan platformon, ahol az mosmlc fordító telepítve van. A telepítéstől függően szükség lehet az `MOSMLHOME`, `MOSMLTOOLS` és `MOSMLC` környezeti változók módosítására.

```
# Unix Makefile stub for separate compilation with Moscow ML.
# See also: Moscow ML Owner s Manual, Recompilation management using
#           mosmldep and make

MOSMLHOME=/usr/bin/mosml
MOSMLTOOLS=/usr/bin/camlrunm /usr/lib/mosml/tools
MOSMLC=/usr/bin/mosmlc -c

.SUFFIXES : .sig .sml .ui .uo

UO_FILES=Symtab.uo Lexical.uo Parsefun.uo Parse.uo Encode.uo \
        Assemble.uo Compile.uo

all:    splc

clean:
        rm -f *.ui
        rm -f *.uo
        rm -f Makefile.bak

.sig.ui:
        $(MOSMLC) $<

.sml.uo:
        $(MOSMLC) $<

splc:   $(UO_FILES)
        mosmlc $(UO_FILES) -o splc

depend:
        rm -f Makefile.bak
```

[4]Paulson, L. C.: *ML for the Working Programmer.* Cambridge University Press 1991, ISBN-0-521-39022-2

```
Encode.ui: Parsefun.ui Symtab.ui
Lexical.uo: Lexical.ui Symtab.ui
Assemble.uo: Assemble.ui Encode.ui Symtab.ui
Assemble.ui: Encode.ui
```

16.7.1. Symtab szignatúrája és struktúrája

```
(* Symtab.sig V4.0
   Symbols table
   for the Simple Programming Language
   P. Hanak, 12-Mar-2005
*)

signature Symtab =
sig

  structure Dict : Binarymap

  type iden  = string
  type token = int

  val FrstToken : int
  val FrstLiter : int
  val Comp : int
  val Load : int
  val Store : int
  val Uncond : int
  val AndThen : int
  val Assign : int
  val Closing : int
  val Do : int
  val Else : int
  val EqLess : int
  val Equal : int
  val Greater : int
  val GreaterEq : int
  val If : int
  val Less : int
```

```
  val isSymb : iden -> bool

  val symtab : (iden, token) Dict.dict ref
  val toDict : ('a*'a -> order) -> ('a*'b) list -> ('a, 'b) Dict.dict

  val lastid : token ref
  val tokenOfName : iden -> token
  val tokenOfTemp : int -> token
  val tokenOfLit : iden -> token
  val tokenOfSymbol : iden -> token

end (* sig *)


(* Symtab.sml V4.0
   Symbols table
   for the Simple Programming Language
   P. Hanak, 13-Mar-2005
*)

structure Symtab :> Symtab =
struct

  structure Dict : Binarymap = Binarymap

  type iden  = string
  type token = int

   (* tokens
   *)
  val FrstToken = ~4
  val FrstLiter = ~4
  val Comp = ~4
  val Load = ~3
  val Store = ~2
  val Uncond = ~1
  val AndThen = 1
  val Assign = 2
```

```
    val Plus = 17
    val Read = 18
    val Then = 19
    val Times = 20
    val While = 21
    val Write = 22
    val LastToken = 23

    val alphanumerics = [("read",Read), ("while",While), ("do",Do),
          ("if",If), ("then",Then), ("else",Else), ("write",Write)]

    and symbolics = [("+",Plus), ("-",Minus), ("*",Times), ("/",Over),
          (":=",Assign), ("<",Less), ("=<",EqLess), ("=",Equal),
          (">",Greater), (">=",GreaterEq), ("/=",NotEq), ("(",Opening),
          (")",Closing), (";",AndThen)]

    (* ---------------------------------------------------------------
       isSpec c = true if c needs special treatment because ???
       isSpec : char -> bool
    *)
    fun isSpec x = List.exists (fn y => x=y) (explode "+-*/:=<>();");

    (* ---------------------------------------------------------------
       isSymb t = true if t is a symbol
       isSymb : iden -> bool
    *)
    fun isSymb t = List.exists (fn (s,_)=> t=s) symbolics;

    (* ---------------------------------------------------------------
       toDict compare ls = an ordered dictionary consisting of the
         elements of list ls with the ordering relation compare
       toDict : ('a * 'a -> order) ->
                                  ('a * 'b) list -> ('a, 'b) Dict.dict
    *)
    fun toDict compare ls =
          let
              fun insert ((k, v), d) = Dict.insert(d, k, v)
          in
```

```
   lastid : token ref
*)
val symtab = ref symtable
and frstid = ref FrstToken
and lastid = ref LastToken;

(* ---------------------------------------------------------------
   tokenOf (symtab, sym, tok, inc) = token of sym
     as side effect assigns the next free token (!tok+inc) to
     sym and updates symtable
   tokenOf : ('a, int) Dict.dict ref * 'a * int ref * int -> int
*)
fun tokenOf (symtab, sym, tok, inc) =
      Dict.find(!symtab, sym)
        handle Dict.NotFound =>
                 let
                     val t = !tok + inc
                     val s = Dict.insert(!symtab, sym, t)
                 in
                     ( symtab := s
                     ; tok := t
                     ; t
                     )
                 end (* let *);

(* ---------------------------------------------------------------
   tokenOfName sym = token of the alphanumeric name sym
     as side effect inserts sym, if new, into symtable
   tokenOfName : iden -> token
*)
fun tokenOfName sym = tokenOf(symtab, sym, lastid, 1)

(* ---------------------------------------------------------------
   tokenOfTemp i = token of the temporary variable %Tn
     as side effect inserts %Ti, if new, into symtable
   tokenOfTemp : int -> token
*)
fun tokenOfTemp i = tokenOfName("%T" ^ Int.toString i)
```

```
    (* ---------------------------------------------------------------
       set the references to their initial value
    *)
    val _ = (symtab := symtable;
             frstid := FrstToken;
             lastid := LastToken
            );

  end (* struct *)
```

16.7.2. Lexical szignatúrája és struktúrája

```
  (* Lexical.sig V4.0
     Lexical analyser
     for the Simple Programming Language
     P. Hanak, 12-Mar-2005
  *)

  signature Lexical =
  sig

    val scan        : string -> Symtab.token list

  end (* sig *)


  (* Lexical.sml V4.0
     Lexical analyser
     for the Simple Programming Language
     P. & D. Hanak, 12-Mar-2005
  *)

  structure Lexical :> Lexical =
  struct

    (* ---------------------------------------------------------------
       alphanum (id, cs) = the pair (t, rs) where
           s = id augmented by an alphanum name extracted from cs and
```

```
     otherwise t = id and rs = cs
   literal : string * char list -> string * char list
*)
fun literal (id, ccs as c::cs) =
      if Char.isDigit c
      then literal(id ^ str c, cs)
      else (id, ccs)
  | literal (id, []) = (id, []);

(* ---------------------------------------------------------------
   symbolic (id, cs) = the pair (t, rs) where
       s = id augmented by a symbolic name extracted from cs and
       rs = rest of cs if cs starts with the symbolic name,
     otherwise t = id and rs = cs
   symbolic : string * char list -> string * char list
*)
fun symbolic (id, ccs as c::cs) =
      if Symtab.isSymb(id ^ str c)
      then (id ^ str c, cs)
      else (id, ccs)
  | symbolic (id, []) = (id, []);

(* ---------------------------------------------------------------
   scanning cs = tokenized form of the source program
                 contained in cs as a list of characters
   scanning : char list -> Symtab.token list
*)
fun scanning (c::cs) =
      if Char.isAlpha c
      then (* process next symbol as alphanumeric name *)
           let
               val (id, cs2) = alphanum(str c, cs)
           in
               Symtab.tokenOfName id :: scanning cs2
           end (* let *)
      else if Char.isDigit c
      then (* process next symbol as literal *)
           let
```

```
    (* ---------------------------------------------------------------
      scan t = tokenized form of the source program
               contained in t as a string
      scan : string -> Symtab.token list
    *)
    fun scan t = scanning(explode t)

  end (* struct *)
```

16.7.3. Parsefun szignatúrája és struktúrája

```
  (* Parsefun.sig V4.0
     Parser functions of a top-down recursive descent parser
     for the Simple Programming Language
     P. Hanak, 13-Mar-2005
  *)

  signature Parsefun =
  sig

    exception ParseErr of string * Symtab.token list;

    datatype 'a partree = N of ('a partree * 'a partree)
                        | L of 'a

    val ptree3 : 'a -> 'a partree -> 'a partree -> 'a partree

    val ofPtree1 : 'a partree -> 'a

    val isPtree1 : 'a partree -> bool

    val % : Symtab.token -> Symtab.token list ->
                            Symtab.token partree * Symtab.token list
    val %% : Symtab.token list ->
                            Symtab.token partree * Symtab.token list
    val ## : Symtab.token list ->
                            Symtab.token partree * Symtab.token list
```

```
  val infixes : (Symtab.token -> 'a -> 'a -> 'a) ->
                (Symtab.token -> int) ->
                (Symtab.token list -> ('a * Symtab.token list)) ->
                (Symtab.token list -> ('a * Symtab.token list))

end (* sig *)


(* Parsefun.sml V4.0
   Parser functions of a top-down recursive descent parser
   for the Simple Programming Language
   P. Hanak, 13-Mar-2005
*)

  (* Credit for the parser primitives, parser functions and the
     parser in SML is due to L. C. Paulson, see "ML for the Working
     Programmer", Cambridge University Press.
  *)

structure Parsefun :> Parsefun =
struct

  (* ------ infixing & precedence of primitive parser functions --- *)
  infix 5 >< -- << <? -/ /- >>;
  infix 0 ||;

  (* ----- execption declarations ---------------------------------- *)
  exception SynErr of string * Symtab.token list;
  exception ParseErr of string * Symtab.token list;
  exception ImpossibleParseTree;

  (* ------ type of the parse tree -------------------------------- *)
  datatype 'a partree = N of ('a partree * 'a partree)
                      | L of 'a;


  (* ****** global auxiliary parser functions ********************** *)
```

```
(* ----------------------------------------------------------------
   isPtree1 n = true if n is a single-leaf parse tree
   isPtree1 : 'a partree -> bool
*)
fun isPtree1 (L _) = true
  | isPtree1 _ = false;


(* ****** primitives parsers ********************************** *)

(* ----------------------------------------------------------------
   % x (t::ts) = (n, ts) where n is a parse tree node constructed
                 of t and ts is the rest of the token list if t is
                 the expected token x, exception SynErr otherwise
   % : Symtab.token -> Symtab.token list ->
         Symtab.token partree * Symtab.token list
*)
fun % (x: Symtab.token) (ts as t::tokens) =
          if x=t then (L t, tokens)
                 else raise SynErr("Unexpected token", ts)
  | % x _ = raise SynErr("Input exhausted, expected token", [x]);

(* ----------------------------------------------------------------
   %% (t::ts) = (n, ts) where n is a parse tree node constructed of
                 t and ts is the rest of the token list if t is the
                 expected identifier, exception SynErr otherwise
   %% : Symtab.token list ->
                       Symtab.token partree * Symtab.token list
*)
fun %% (ts as (t: Symtab.token) :: tokens) =
           if Symtab.LastToken < t then (L t, tokens)
           else raise SynErr("Name expected", ts)
  | %% _ = raise SynErr("Input exhausted, name expected", []);

(* ----------------------------------------------------------------
   ## (t::ts) = (n, ts) where n is a parse tree node constructed of
                 t and ts is the rest of the token list if t is the
                 expected numeral, exception SynErr otherwise
```

```
fun ? tokens = (L Symtab.None, tokens);


(* ****** parser functions ************************************** *)

(* ------- local auxiliary parser function -------------------------
   (ph1 >< ph2) ts = (x, y, rs) where x is a parse tree built by ph1
        and y is a parse tree built by ph2 from the token list ts,
        and rs is the rest of ts
   >< : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> 'b * 'd * 'e
*)
fun (ph1 >< ph2) tokens =
      let
          val (x, tokens2) = ph1 tokens
          val (y, tokens3) = ph2 tokens2
      in
          (x, y, tokens3)
      end (* let *);

(* ----------------------------------------------------------------
   (ph1 -- ph2) ts = (n, rs) where n is a parse tree node
        constructed of the two parse trees built by parser
        primitives ph1 and ph2 from the token list ts in THIS order,
        and rs is the rest of ts
   -- : ('a -> 'b partree * 'c) * ('c -> 'b partree * 'd) ->
                                          'a -> 'b partree * 'd
*)
fun (ph1 -- ph2) tokens =
      let
          val (x, y, ts) = (ph1 >< ph2 handle SynErr(s,t) =>
                                      raise ParseErr(s,t)) tokens
      in
          (N(x, y), ts)
      end (* let *);

(* ----------------------------------------------------------------
   (ph1 << ph2) ts = (n, rs) where n is a parse tree node
         constructed of the two parse trees built by ph1 and ph2
```

```
      (ph1 <? ph2) ts = (n, rs) where n is a parse tree node
            constructed of the two parse trees built by ANY parser
            primitives ph1 and ph2 from the token list ts in REVERSE
            order, and rs is the rest of ts
      <? : ('a -> 'b partree * 'c) * ('c -> 'b partree * 'd) ->
                                                 'a -> 'b partree * 'd
   *)
   fun (ph1 <? ph2) tokens =
         let
             val (x, y, ts) = (ph1 >< ph2) tokens
         in
             (N(y, x), ts)
         end (* let *);

   (* ----------------------------------------------------------------
      (ph1 -/ ph2) ts = (n, rs) where n is a parse tree node
            constructed of the parse tree built by the parser
            primitive ph1 from the token list ts, and rs is
            the rest of ts
      -/ : ('a -> 'b partree * 'c) * ('c -> 'd * 'e) -> 'a ->
                                                      'b partree * 'e
   *)
   fun (ph1 -/ ph2) tokens =
         let
             val (x, _, ts) = (ph1 >< ph2) tokens
         in
             (x : 'b partree, ts)
         end (* let *);

   (* ----------------------------------------------------------------
      (ph1 /- ph2) ts = (n, rs) where n is a parse tree node
            constructed of the parse tree built by the parser
            primitive ph2 from the token list ts, and rs is
            the rest of ts
      /- : ('a -> 'b * 'c) * ('c -> 'd partree * 'e) -> 'a ->
                                                      'd partree * 'e
   *)
   fun (ph1 /- ph2) tokens =
```

```
         value extracted by the parser primitive ph from the token
         list ts, and rs is the rest of ts
   >> :  ('a -> 'b * 'c) * ('b -> 'd) -> 'a -> 'd * 'c
*)
fun (ph >> f) tokens =
       let
           val (x, tokens2) = ph tokens
                     handle SynErr(s,t) => raise ParseErr(s,t)
       in
           (f x, tokens2)
       end (* let *);

(* ----------------------------------------------------------------
   infixes apply precOf ph = a parsed infix operator having
                             considered its precedence
   infixes : (Symtab.token -> 'a -> 'a -> 'a) ->
             (Symtab.token -> int) ->
             (Symtab.token list -> 'a * Symtab.token list) ->
                         Symtab.token list -> 'a * Symtab.token list
*)
fun infixes apply precOf ph =
    let
        (* ---------------------------------------------------------
           over k tokens =
           over : int -> Symtab.token list -> 'a * Symtab.token list
        *)
        fun over k tokens = next k (ph tokens)

        (* ---------------------------------------------------------
           next k (x, ts) =
           next : int -> 'a * Symtab.token list ->
                                             'a * Symtab.token list
        *)
        and next k (x, (a: Symtab.token)::tokens) =
              if precOf a <= k
              then (x, a::tokens)
              else next k ((over (precOf a) >> apply a x) tokens)
          | next k (x, tokens) = (x, tokens)
```

```
            (* -------------------------------------------------------
               over (apply, precOf, ph) k tokens =
               over : (Symtab.token -> 'a -> 'a -> 'a) *
                      (Symtab.token -> int) *
                      (Symtab.token list -> 'a * Symtab.token list) ->
                      Symtab.token -> Symtab.token list ->
                                                 'a * Symtab.token list
            *)
            fun over (apply, precOf, ph) k tokens =
                                 next (apply, precOf, ph) k (ph tokens)

            (* -------------------------------------------------------
               next (apply, precOf, ph) k (x, ts) =
               next : (Symtab.token -> 'a -> 'a -> 'a) *
                      (Symtab.token -> int) *
                      (Symtab.token list -> 'a * Symtab.token list) ->
                      Symtab.token -> 'a * Symtab.token list ->
                                                 'a * Symtab.token list
            *)
            and next (apply, precOf, ph)
                                       k (x, (a: Symtab.token)::tokens) =
                  if precOf a <= k
                  then (x, a::tokens)
                  else next (apply, precOf, ph) k
                              ((over (apply, precOf, ph) (precOf a) >>
                                                        apply a x) tokens)
              | next _ k (x, tokens) = (x, tokens)
         in
            over (apply, precOf, ph) 0
         end (* let *);
     *)

     end (* struct *)
```

16.7.4. Parse szignatúrája és struktúrája

```
     (* Parse.sig V4.0
        Top-down recursive descent parser
```

```
   for the Simple Programming Language
   P. & D. Hanak, 13-Mar-2005
*)

structure Parse :> Parse =
struct

local
      open Symtab Parsefun
in
   (* ------ aliases for some parser primitives ----------------------
      name, numb, empty : Symtab.token list ->
                                 Symtab.token partree * Symtab.token list
   *)
   val name = %%
   val numb = ##
   val empty = ?

   (* ------ infixing & precedence of primitive parser functions -----
   *)
   infix 5 -- << <? -/ /-
   infix 0 ||

   (* ------ simple predicates ----------------------------------------
      isAddop, isMulop, isOp : Symtab.token -> bool
   *)
   fun isAddop t = t = Plus  orelse t = Minus
   fun isMulop t = t = Times orelse t = Over
   fun isOp t = isAddop t orelse isMulop t

   (* ------ relative precedence of operators -------------------------
      precOf : Symtab.token -> int
   *)
   fun precOf t = if      isMulop t then 2
                  else if isAddop t then 1
                  else ~1

   (* ------ a parser function for infix operators -------------------
```

```
*)
fun stmts tokens =
      (   stmt <? %AndThen -- stmts
       || stmt) tokens

and stmt tokens =
      (   name << %Assign -- expr
       || %If -- test -/ %Then -- stmt -/ %Else -- stmt
       || %While -- test -/ %Do -- stmt
       || %Read -- name
       || %Write -- expr
       || %Opening /- stmts -/ %Closing
       || empty) tokens

and test tokens =
       (expr << compOp -- expr) tokens

and compOp tokens =
        (   %Equal || %Less || %Greater || %EqLess
         || %GreaterEq || %NotEq) tokens

and op2 tokens =
        (%Plus || %Minus) tokens

and op1 tokens =
        (%Times || %Over) tokens

and expr0 tokens =
        (   name
         || numb
         || %Opening /- expr -/ %Closing) tokens

and expr tokens = infixExpr expr0 tokens

(* ----------------------------------------------------------------
   parse ts = the parse tree of token list ts
   parse : Symtab.token list -> Symtab.token partree
*)
```

```
signature Encode =
sig

  type label = int
  datatype instr = Ins of (Symtab.token * Symtab.token)
                 | Jmp of (Symtab.token * label)
                 | Lab of label

  exception ImpossibleEncoding

  val encode : Symtab.token Parsefun.partree -> instr list

end (* sig *)


(* Encode.sml V4.0
   Relocatable code generator
   for the Simple Programming Language
   P. Hanak, 13-Mar-2005
*)

(* ************************************************************** *)
(* Implements the abstract syntax of spl                          *)
(* ************************************************************** *)

structure Encode :> Encode =
struct

(* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ *)
(* Primitive encoding functions                                   *)
(* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ *)

  (* --------- type of labels and instructions--------------------- *)
  type label = int
  datatype instr = Ins of (Symtab.token * Symtab.token)
                 | Jmp of (Symtab.token * label)
                 | Lab of label
```

```
      assign : instr list * Symtab.token -> instr list
   *)
   fun assigncmd (r, s) = Ins(Symtab.Store, s) :: r

   (* --------- arithmetic operators -------------------------------
      aritop (k, r, s) = encoded arithmetic operation
                         concatenated to r
      aritop : Symtab.token * instr list * Symtab.token -> instr list
   *)
   fun aritop (k, r, s) = Ins(k, s) :: r

   (* --------- input/output command -------------------------------
      iocmd (k, s) = encoded io operation
      iocmd : Symtab.token * Symtab.token -> instr list
   *)
   fun iocmd (k, s) = [Ins(k, s)]

   (* --------- comparison operators -------------------------------
      compop (k, r, s, l1) = encoded comparison operation
                             concatenated to r
      compop : Symtab.token * instr list * Symtab.token *
                 Symtab.token -> instr list
   *)
   fun compop (k, r, s, l1) =
         Jmp(k, l1) :: Ins(Symtab.Comp, s) :: r

   (* --------- if-then{-else} command -----------------------------
      ifcmd (t, r, s, l1, l2) = encoded if command
                                concatenated to r and t
      ifcmd : instr list * instr list * instr list * Symtab.token *
                Symtab.token -> instr list
   *)
   fun ifcmd (t, r, s, l1, l2) =
         Lab l2 :: s @ (Lab l1 :: Jmp(Symtab.Uncond, l2) :: r @ t)

   (* ---------- while command --------------------------------------
      whilecmd (t, s, l1, l2) = encoded while command concatenated
                                to s and t
```

```
(* ---------------------------------------------------------------
   execption declaration
*)
exception ImpossibleEncoding

(* --------- enc0 translates instructions with no operand --------
   enc0 node =
   enc0 : Symtab.token Parsefun.partree -> instr list
*)
fun enc0 (node as L key) =
      if   key = None
      then []
      else raise ImpossibleEncoding
  | enc0 _ = raise ImpossibleEncoding

(* --------- enc1 translates instructions with a single operand --
   enc1 node =
   enc1 : Symtab.token Parsefun.partree -> instr list
*)
and enc1 (node as N(L key, p)) =
      if   key = Read
      then iocmd(key, ofPtree1 p)
      else if key = Write
      then cmdConcat(assigncmd(enc2x(p, 0),
                               Symtab.tokenOfTemp 0),
                     iocmd(key, Symtab.tokenOfTemp 0)
                    )
      else enc0 node
  | enc1 node = enc0 node

(* --------- enc2 translates instructions with two operands ------
   enc2 (node, lab) =
   enc2 : Symtab.token Parsefun.partree * label ->
            instr list * label
*)
and enc2 (node as N(N(L key, p), q), lab0) =
      if key = AndThen
      then let
```

```
                                compop(key,
                                       enc2x(p, 0),
                                       Symtab.tokenOfTemp 0, lab0)
                                ),
                        lab0 + 1
                  )
            else if key = While
            then let
                     val (s1, _) = enc2(p, lab0)
                     val (s2, lab2) = enc3(q, lab0 + 2)
                  in
                    (whilecmd(s1, s2, lab0, lab0 + 1), lab2)
                  end (* let *)
            else (enc1 node, lab0)
          | enc2 (node, lab0) = (enc1 node, lab0)

        (* --------- enc2x translates instructions with two operands where
           --------- the precedence of the operands is important ---------
           enc2x (node, tmp) =
           enc2x : Symtab.token Parsefun.partree * int -> instr list
        *)
        and enc2x (node as N(N(L key, p), q), tmp) =
              if key = Over orelse key = Minus orelse
                 key = Plus orelse key = Times
              then if isPtree1 q
                    then aritop(key, enc2x(p, tmp), ofPtree1 q)
                    else cmdConcat(assigncmd(enc2x(q, tmp),
                                             Symtab.tokenOfTemp tmp),
                                   aritop(key,
                                          enc2x(p, tmp+1),
                                          Symtab.tokenOfTemp tmp)
                                  )

              else enc1 node
          | enc2x (L p, tmp) = loadcmd p
          | enc2x (node, tmp) = enc1 node

        (* --------- enc3 translates instructions with three operands ----
```

```
  (* --------- encode translates a parse tree, and returns the
     --------- relocatable code as a list of instructions ----------
     encode node =
     encode : Symtab.token Parsefun.partree -> instr list
  *)
  fun encode node = #1(enc3(node, 0))

end (* local *)

end (* struct *)
```

16.7.6. Assemble szignatúrája és struktúrája

```
(* Assemble.sig V4.0
   Absolute code generator
   for the Simple Programming Language
   P. Hanak, 13-Mar-2005
*)

signature Assemble =
sig

  val assemble : Encode.instr list -> string

end (* sig *)


(* Assemble.sml V4.0
   Absolute code generator
   for the Simple Programming Language
   P. Hanak, 13-Mar-2005
*)

structure Assemble :> Assemble =
struct

  structure Dict = Symtab.Dict
```

```
         cinstrs : (Symtab.token * Symtab.iden) list
      *)
      val cinstrs = [(Load,"LOADC"), (Comp,"SUBC"), (Over,"DIVC"),
                     (Minus,"SUBC"), (Plus,"ADDC"), (Times,"MULC")]
   end (* local *)


   (* ****** pass1 of assembling: address allocation*************** *)

   local
      (* ---------------------------------------------------------------
         p1 (ls, is, len, labs) = the triple (is, len, labs), where
            ls   = encoded instruction list with labels,
            is   = modified instruction list without labels,
            len  = length of modified instruction list,
            labs = list of label & address pairs
         p1 : instr list * instr list * int * (int * int) list ->
                                   instr list * int * (int * int) list
      *)
      fun p1 ((Encode.Lab l)::ls, is, len, labs) =
                                    p1(ls, is, len, (l, len)::labs)
        | p1 (l::ls, is, len, labs) =
                                    p1(ls, l::is, len+1, labs)
        | p1 ([], is, len, labs) = (is, len, labs)
   in
      (* ---------------------------------------------------------------
         pass1 ls = the triple (is, len, labs) where
            ls   = encoded instruction list with labels,
            is   = modified instruction list without labels,
            len  = length of modified instruction list,
            labs = list of label & address pairs
         pass1 : instr list -> instr list * int * (int * int) list
      *)
      fun pass1 ls = p1(ls, [], 0, [])
   end (* local *)

   (* ----------------------------------------------------------------
      instAddr adr = adr converted to string, prefixed by "\{}n" and
```

```
(* ----------------------------------------------------------------
   tokLits () = list of token-literal pairs from the symbol table
   tokLits : unit -> (token * iden) list
*)
fun tokLits () =
      map (fn (x, y) => (y, x))
          (List.filter (fn (_, y) => y <= Symtab.FrstLiter)
                       (Dict.listItems(!Symtab.symtab)))

(* ----------------------------------------------------------------
   findL x zs = v of (i, v) element of zs if i = x,
                exception Option otherwise
   findL : ''a * (''a * 'b) list -> 'b
*)
fun findL x = #2 o Option.valOf o (List.find (fn (i,_) => i=x))


(* ****** pass2 of assembling: assembly code generation ********* *)

local
  (* ----------------------------------------------------------------
     p2 (is, len, labs, mnemtr, littr, adr) = a string containing
        the assembled code where
          is     = modified instruction list,
          len    = length of modified instruction list,
          labs   = list of label & address pairs,
          mnemtr = tree of mnemonics,
          littr  = tree of literals,
          adr:    line number in the listing
     p2 : instr list * int * (int * int) list *
            (int, string) Dict.dict * (int, string) Dict.dict *
            int -> string
  *)
  fun p2 ([], len, labs, mnemtr, littr, adr) =
             endCodeGen(!Symtab.lastid - Symtab.LastToken, adr)
    | p2 (Encode.Ins(i, a)::is, len, labs, mnemtr, littr, adr) =
             instAddr adr  ^
             (if a < Symtab.FrstToken
```

```
              len    = length of modified instruction list,
              labs   = list of label & address pairs
          pass2 :  instr list * int * (int * int) list -> string
       *)
       fun pass2 (is, len, labs) =
         let
             val mnemtr = Symtab.toDict Int.compare instrs
             val littr =
                     Symtab.toDict Int.compare (cinstrs @ tokLits())
         in
             p2(is, len, labs, mnemtr, littr, 0)
         end (* let *)
     end (* local *)

     (* assemble performs pass1 than pass2
        assemble : instr list -> string
     *)
     val assemble = pass2 o pass1

   end (* struct *)
```

16.7.7. Compile szignatúrája és struktúrája

```
   (* Compile.sml V4.0
      IO processing and compilation
      for the Simple Programming Language
      P. Hanák, 13-Mar-2005
   *)

   structure Compile =
   struct

     structure Dict = Symtab.Dict

   val withTokens = false (* controls output for debugging *)

   local
```

```
      | [a] => a
      | _   => raise CmdLineErr
                     "Too many arguments. Usage: splc <filename>."

(* ----------------------------------------------------------------
   error msg = as a side effect prints an error message
   error : string -> unit
*)
fun error msg = print ("\{}n***** " ^ msg ^ " *****\{}n\{}n")

(* ----------------------------------------------------------------
   x >> f = f x  -- "pipe" argument x through function f
   >> : 'a * 'b -> 'c
*)
infix 2 >>;
fun op>> (b, a) = a b;

(* ----------------------------------------------------------------
   compile prg =  the assembly version of the spl-source prg
   compile : string -> string
*)
fun compile prg =
      prg >> Lexical.scan >> Parse.parse >>
             Encode.encode >> Assemble.assemble
      handle Parsefun.ParseErr(s, t) =>
        let
            (* ----------------------------------------------------
               tstab = ordered dictionary, mapping tokens to symbols
               tstab : (Symtab.token, Symtab.iden) Dict.dict
            *)
            val tstab = Symtab.toDict
                        Int.compare
                        (map (fn (x,y) => (y,x))
                             (Dict.listItems(!Symtab.symtab)))
            (* ----------------------------------------------------
               tsstr = spl-source rebuilt from token list
               tssrt : string
            *)
```

```
*)

  (* ----------------------------------------------------------------
     splc a = as a side effect, appends filename extensions to a,
              compiles the spl-source given as the input file and
              generates the assembly code as the ouput file
     splc : string -> unit
  *)
  fun splc a =
       let
           val inp = TextIO.openIn(a ^ ".spl")
           val out = TextIO.openOut(a ^ ".asm")
       in
          ( TextIO.output(out, compile(TextIO.inputAll inp))
          ; TextIO.closeIn inp
          ; TextIO.closeOut out
          )
       end
         handle Io {cause, function="openIn", name} =>
                  error("Input file '" ^ name ^ "' does not exists.")
              | Io {cause, function="openOut", name} =>
                  error("Output file '" ^ name ^
                                              "' cannot be opened.")
              | Io {cause, function, name} =>
                  error("File '" ^ name ^ "' caused " ^
                                                function ^ " error.")
in
  (* ----------------------------------------------------------------
     run splc using its single command line argument as the base name
     of both the input and the output files
  *)
  val _ = splc(Path.base(cmdLineArg())) handle CmdLineErr s => error s

end (* local *)

end (* struct *)
```

Ez a szintaxisleírás a *Moscow ML Language Overview – V1.44* c. kézikönyv alapján készült. A *V2.0* változat kézikönyvében leírt alapnyelv lényegében ugyanaz, bár 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 (ún. tapadójelek) 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:

`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 3. fejezetben ismertettük.

		`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

`open` $longstrid_1$ ... $longstrid_n$	$n \geq 1$	$n \geq 1$
	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.2.4. 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

---

`true`, `false`, `nil`, `::` és `Bool`-nak kapnak új értéket egy *datbind*, *exbind* vagy *datbind* deklarációban. Az `it` név nem kaphat új értéket egy *datbind* vagy *exbind* deklarációban.

A zárthelyin és a vizsgán az alábbi típusok, kivételek, függvények és konstruktorok ismeretét várjuk el. A *-gal megjelölteket csak az mosml ismeri. Az összefoglalót a Moscow ML 2.0 szignatúráiból készítettük.

Belső típusok, kivételek, függvények és konstruktorok. A felsoroltak nagy részét a `General` és a `Meta` struktúra definiálja.

- Belső típusok

  `bool`, `char`, `exn`, `int`, `´a list`, `´a option`, `order`, `real`, `string`, `unit`, `word`, `word8`.

- Belső kivételek

  `Bind`, `Chr`, `Div`, `Domain`, `Fail`, `Interrupt`, `Io`, `Match`, `Option`, `Ord`, `Overflow`, `Size`, `Subscript`.

- Belső függvények és konstruktorok a kezdeti környezetben

  `~`, `+`, `-`, `*`, `/`, `^`, `::`, `@`, `=`, `<>`, `<`, `<=`, `>=`, `>`, `abs`, `app`, `before`, `ceil`, `chr`, `concat`, `div`, `explode`, `false`, `floor`, `foldl`, `foldr`, `hd`, `implode`, `length`, *`makestring`, `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`, `use`.

`Bool` struktúra. `bool`, `not`, `toString`, `fromString`.

`Char` struktúra. `char`, `minChar`, `maxChar`, `maxOrd`, `chr`, `ord`, `succ`, `pred`, `isLower`, `isUpper`, `isDigit`, `isAlpha`, `isHexDigit`, `isAlphaNum`, `isPrint`, `isSpace`, `isPunct`, `isGraph`, `isAscii`, `isCntrl`, `toLower`, `toUpper`, `contains`, `notContains`, `fromString`, `toString`, `<`, `<=`, `>`, `>=`, `compare`.

`Int` struktúra. `int`, `precision`, `minInt`, `maxInt`, `~`, `*`, `div`, `mod`, `quot`, `rem`, `+`, `-`, `<`, `<=`, `>`, `>=`, `abs`, `min`, `max`, `sign`, `compare`, `toString`, `fromString`.

`List` struktúra. `´a list`, `null`, `hd`, `tl`, `last`, `nth`, `take`, `drop`, `length`, `rev`, `@`, `concat`, `revAppend`, `app`, `map`, `mapPartial`, `find`, `filter`, `partition`, `foldr`, `foldl`, `exists`, `all`, `tabulate`.

<=, >, >=.

Az alábbi típusok, kivételek, függvények és konstruktorok megismerését ajánljuk a jegyzetben szereplő egyes példák megértéséhez, a nagy házi feladat és más programozási feladatok megoldásához.

Binarymap struktúra. `('key, 'a) dict`, `mkDict`, `insert`, `find`, `remove`, `numItems`, `listItems`, `app`, `revapp`, `foldr`, `foldl`, `map`, `transform`.

Binaryset struktúra. `'item set`, `empty`, `singleton`, `add`, `addList`, `retrieve`, `isEmpty`, `equal`, `isSubset`, `member`, `delete`, `numItems`, `union`, `intersection`, `difference`, `listItems`, `app`, `revapp`, `foldr`, `foldl`, `find`.

ListPair struktúra. `zip`, `unzip`, `map`, `app`, `all`, `exists`, `foldr`, `foldr`.

Random struktúra. `generator`, `newgenseed`, `newgen`, `random`, `rangelist`, `range`, `rangelist`.

Regex struktúra. `Regex`, `regex`, `cflag`, `eflag`, `replacer`, `regcomp`, `regexec`, `regexecBool`, `regnexec`, `regnexecBool`, `regmatch`, `regmatchBool`, `replace1`, `replace`, `substitute1`, `substitute`, `tokens`, `fields`, `map`, `app`, `fold`.

Splaymap struktúra. `('key, 'a) dict`, `mkDict`, `insert`, `find`, `remove`, `numItems`, `listItems`, `app`, `revapp`, `foldr`, `foldl`, `map`, `transform`.

Splayset struktúra. `'item set`, `empty`, `singleton`, `add`, `addList`, `retrieve`, `isEmpty`, `equal`, `isSubset`, `member`, `delete`, `numItems`, `union`, `intersection`, `difference`, `listItems`, `app`, `revapp`, `foldr`, `foldl`, `find`.

StringCvt struktúra. `padLeft`, `padRight`.

TextIO struktúra. `instream`, `outstream`, `openIn`, `closeIn`, `input`, `inputAll`, `inputLine`, `endOfStream`, `lookahead`, `stdIn`, `openOut`, `openAppend`, `closeOut`, `output`, `output1`, `flushOut`, `stdOut`, `stdErr`, `print`.

Time struktúra. `time`, `Time`, `zeroTime`, `now`, `toSeconds`, `toMilliseconds`, `toMicroseconds`, `fromSeconds`, `fromMilliseconds`, `fromMicroseconds`, `fromReal`, `toReal`, `toString`, `fromString`, +, −, <, <=, >, >=, `compare`.

```
(* From SML/NJ lib 0.2, copyright 1993 by AT&T Bell Laboratories  *)
(* Original implementation due to Stephen Adams, Southampton, UK  *)

type ('key, 'a) dict

exception NotFound

val mkDict    : ('key * 'key -> order) -> ('key, 'a) dict
val insert    : ('key, 'a) dict * 'key * 'a -> ('key, 'a) dict
val find      : ('key, 'a) dict * 'key -> 'a
val peek      : ('key, 'a) dict * 'key -> 'a option
val remove    : ('key, 'a) dict * 'key -> ('key, 'a) dict * 'a
val numItems  : ('key, 'a) dict -> int
val listItems : ('key, 'a) dict -> ('key * 'a) list
val app       : ('key * 'a -> unit) -> ('key,'a) dict -> unit
val revapp    : ('key * 'a -> unit) -> ('key,'a) dict -> unit
val foldr     : ('key * 'a * 'b -> 'b)-> 'b -> ('key,'a) dict -> 'b
val foldl     : ('key * 'a * 'b -> 'b) -> 'b -> ('key,'a) dict -> 'b
val map       : ('key * 'a -> 'b) -> ('key,'a) dict -> ('key, 'b) dict
val transform : ('a -> 'b) -> ('key,'a) dict -> ('key, 'b) dict

(*
   [('key, 'a) dict] is the type of applicative maps from domain type
   'key to range type 'a, or equivalently, applicative dictionaries
   with keys of type 'key and values of type 'a.  They are implemented
   as ordered balanced binary trees.

   [mkDict ordr] returns a new, empty map whose keys have ordering
   ordr.

   [insert(m, i, v)] extends (or modifies) map m to map i to v.

   [find (m, k)] returns v if m maps k to v; otherwise raises NotFound.

   [peek(m, k)] returns SOME v if m maps k to v; otherwise returns NONE.

   [remove(m, k)] removes k from the domain of m and returns the
   modified map and the element v corresponding to k.  Raises NotFound
   if k is not in the domain of m.

   [numItems m] returns the number of entries in m (that is, the size
   of the domain of m).
```

```
    in m, in decreasing order of k.

    [map f m] returns a new map whose entries have form (k, f(k,v)),
    where (k, v) is an entry in m.

    [transform f m] returns a new map whose entries have form (k, f v),
    where (k, v) is an entry in m.
*)
```

B.2. Structure Binaryset

```
(* Binaryset -- sets implemented by ordered balanced binary trees *)
(* From SML/NJ lib 0.2, copyright 1993 by AT&T Bell Laboratories  *)
(* Original implementation due to Stephen Adams, Southampton, UK  *)

type 'item set

exception NotFound

val empty        : ('item * 'item -> order) -> 'item set
val singleton    : ('item * 'item -> order) -> 'item -> 'item set
val add          : 'item set * 'item -> 'item set
val addList      : 'item set * 'item list -> 'item set
val retrieve     : 'item set * 'item -> 'item
val peek         : 'item set * 'item -> 'item option
val isEmpty      : 'item set -> bool
val equal        : 'item set * 'item set -> bool
val isSubset     : 'item set * 'item set -> bool
val member       : 'item set * 'item -> bool
val delete       : 'item set * 'item -> 'item set
val numItems     : 'item set ->  int
val union        : 'item set * 'item set -> 'item set
val intersection : 'item set * 'item set -> 'item set
val difference   : 'item set * 'item set -> 'item set
val listItems    : 'item set -> 'item list
val app          : ('item -> unit) -> 'item set -> unit
val revapp       : ('item -> unit) -> 'item set -> unit
val foldr        : ('item * 'b -> 'b) -> 'b -> 'item set -> 'b
val foldl        : ('item * 'b -> 'b) -> 'b -> 'item set -> 'b
```

[add(s, i)] adds item i to set s.

[addList(s, xs)] adds all items from the list xs to the set s.

[retrieve(s, i)] returns i if it is in s; raises NotFound otherwise.

[peek(s, i)] returns SOME i if i is in s; returns NONE otherwise.

[isEmpty s] returns true if and only if the set is empty.

[equal(s1, s2)] returns true if and only if the two sets have the same elements.

[isSubset(s1, s2)] returns true if and only if s1 is a subset of s2.

[member(s, i)] returns true if and only if i is in s.

[delete(s, i)] removes item i from s. Raises NotFound if i is not in s.

[numItems s] returns the number of items in set s.

[union(s1, s2)] returns the union of s1 and s2.

[intersection(s1, s2)] returns the intersectionof s1 and s2.

[difference(s1, s2)] returns the difference between s1 and s2 (that is, the set of elements in s1 but not in s2).

[listItems s] returns a list of the items in set s, in increasing order.

[app f s] applies function f to the elements of s, in increasing order.

[revapp f s] applies function f to the elements of s, in decreasing order.

[foldl f e s] applies the folding function f to the entries of the set in increasing order.

[foldr f e s] applies the folding function f to the entries of the

```
val not        : bool -> bool

val toString   : bool -> string
val fromString : string -> bool option
val scan       : (char, 'a) StringCvt.reader -> (bool, 'a) StringCvt.reader

(*
   [bool] is the type of Boolean (logical) values: true and false.

   [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.
*)
```

B.4. Structure Char

```
(* Char -- SML Basis Library *)

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 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).
```

[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

---

```
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. For instance, 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.< .

```
(* Additional Moscow ML top-level types *)

datatype bool = false | true
eqtype char
eqtype int
datatype 'a option = NONE | SOME of 'a
type ppstream
eqtype real
eqtype string
type substring
type syserror
type 'a vector
eqtype word
eqtype word8
datatype 'a list = nil | op :: of 'a * 'a list
datatype 'a ref  = ref of 'a
datatype 'a frag = QUOTE of string | ANTIQUOTE of 'a

(* SML Basis Library exceptions *)

exception Bind
exception Chr
exception Div
exception Domain
exception Fail of string
exception Match
exception Overflow
exception Subscript
exception Size

(* Additional Moscow ML top-level exceptions *)

exception Graphic of string
exception Interrupt
exception Invalid_argument of string
exception Io of function : string, name : string, cause : exn
exception Out_of_memory
exception SysErr of string * syserror option

(* SML Basis Library values *)

val !          : 'a ref -> 'a
```

```
val <>     : ''a * ''a -> bool

val ceil   : real -> int                (* round towards plus infinity  *)
val floor  : real -> int                (* round towards minus infinity *)
val real   : int -> real                (* equals Real.fromInt          *)
val round  : real -> int                (* round to nearest even        *)
val trunc  : real -> int                (* round towards zero           *)

val vector : 'a list -> 'a vector

(* 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

val makestring : numtxt -> string

(* 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                   *)

(*
   [exn] is the type of exceptions.

   [unit] is the type containing the empty tuple () which equals the
   empty record  .
```

[real] is the type of floating-point numbers. Equals Real.real.

[string] is the type of character strings. Equals String.string.

[substring] is the type of substrings. Equals Substring.substring.

[syserror] is the abstract type of system error codes.
Equals OS.syserror.

[vector] is the type of immutable vectors. Equals Vector.vector.

[word] is the type of unsigned words. Equals Word.word.

[word8] is the type of unsigned bytes. Equals Word8.word.

['a list] is the type of lists of elements of type 'a.
Equals List.list.

['a ref] is the type of mutable references to values of type 'a.

['a frag] is the type of quotation fragments, resulting from the parsing of quotations ` ... ` and antiquotations. See the Moscow ML Owner's Manual.

[Bind] is the exception raised when the right-hand side value in a valbind does not match the left-hand side pattern.

[Chr] signals an attempt to produce an unrepresentable character.

[Div] signals an attempt to divide by zero.

[Domain] signals an attempt to apply a function outside its domain of definition; such as computing Math.sqrt(~1).

[Fail] signals the failure of some function, usually in the Moscow ML specific library structures.

[Match] signals the failure to match a value against the patterns in a case, handle, or function application.

[Overflow] signals the attempt to compute an unrepresentable number.

---

input/output operation (function) when operating on a file (name). The third field (cause) may give a reason for the failure.

[Out_of_memory] signals an attempt to create a data structure too large for the implementation, or the failure to extend the heap or stack.

[SysErr (msg, err)] signals a system error, described by msg. A system error code may be given by err. If so, it will usually hold that msg = OS.errorMsg err.

SML Basis Library values

[! rf] returns the value pointed to by reference rf.

[:=(rf, e)] evaluates rf and e, then makes the reference rf point to the value of e. Since := has infix status, this is usually written

```
rf := e
```

[o(f, g)] computes the functional composition of f and g, that is, fn x => f(g x). Since o has infix status, this is usually written

```
f o g
```

[ignore e] evaluates e, discards its value, and returns () : unit.

[before(e1, e2)] evaluates e1, then evaluates e2, then returns the value of e1. Since before has infix status, this is usually written

```
e1 before e2
```

[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.

[real i] is the floating-point number representing integer i. Equivalent to Real.fromInt.

[round r] is the integer nearest to r, using the default rounding mode. May raise Overflow.

[trunc r] is the numerically largest integer between r and zero (rounds towards zero). May raise Overflow.

[vector [x1, ..., xn]] returns the vector #[x1, ..., xn].

[< (x1, x2)]
[<=(x1, x2)]
[> (x1, x2)]
[>=(x1, x2)]

These are the standard comparison operators for arguments of type int, Word.word, Word8.word, real, char or string.

[makestring v] returns a representation of value v as a string, for v of type int, Word.word, Word8.word, real, char or string.

[~ x] is the numeric negation of x (which can be real or int). May raise Overflow.

[abs x] is the absolute value of x (which can be real or int). May raise Overflow.

[+ (e1, e2)]
[- (e1, e2)]
[(e1, e2)]*

These are the standard arithmetic operations for arguments of type int, Word.word, Word8.word, and real. They are unsigned in the case of Word.word and Word8.word. May raise Overflow.

[/ (e1, e2)] is the floating-point result of dividing e1 by e2. May raise Div and Overflow.

[div(e1, e2)] is the integral quotient of dividing e1 by e2 for

---

```
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 scan      : StringCvt.radix
                -> (char, 'a) StringCvt.reader -> (int, 'a) StringCvt.reader
val fmt       : StringCvt.radix -> int -> string

val toString  : int -> string
val fromString : string -> int option   (* Overflow      *)

(*
   [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.
```

Evaluating i div 0 raises Div. Evaluating i div ~1 raises Overflow if i is the most negative integer.

[mod] is the remainder for div. 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 mod 0 raises Div, whereas i mod ~1 = 0, for all i.

[quot] is integer division, rounding towards zero. 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 r is zero or has the same sign as i. 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).

```
   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]+
*)
```

B.7. Structure List

```
(* List -- SML Basis Library *)

datatype list = datatype list

exception Empty

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 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

(*
   ['a list] is the type of lists of elements of type 'a.

   [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.
```

```
   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
*)
```

B.8. Structure ListPair

```
(* ListPair -- SML Basis Library *)

val zip    : 'a list * 'b list -> ('a * 'b) list
```

```
    [unzip xys] returns a pair (xs, ys), where xs is the list of first
    components of xys, and ys is the list of second components from
    xys.  Hence zip (unzip xys) has the same result and effect as xys.

    [map f (xs, ys)] applies function f to the pairs of corresponding
    elements of xs and ys and returns the list of results.  Hence
    map f (xs, ys) has the same result and effect as List.map f (zip (xs, ys)).

    [app f (xs, ys)] applies function f to the pairs of corresponding
    elements of xs and ys and returns ().  Hence app f (xs, ys) has the
    same result and effect as List.app f (zip (xs, ys)).

    [all p (xs, ys)] applies predicate p to the pairs of corresponding
    elements of xs and ys until p evaluates to false or one or both
    lists is exhausted; returns true if p is true of all such pairs;
    otherwise false.  Hence all p (xs, ys) has the same result and
    effect as Lisp.all p (zip (xs, ys)).

    [exists p (xs, ys)] applies predicate p to the pairs of corresponding
    elements of xs and ys until p evaluates to true or one or both
    lists is exhausted; returns true if p is true of any such pair;
    otherwise false.  Hence exists p (xs, ys) has the same result and
    effect as Lisp.exists p (zip (xs, ys)).

    [foldr f e (xs, ys)] evaluates f(x1, y1, f(x2, y2, f(..., f(xn, yn, e))))
    where xs = [x1, x2, ..., x(n-1), xn, ...],
          ys = [y1, y2, ..., y(n-1), yn, ...],
    and    n = min(length xs, length ys).
    Equivalent to List.foldr (fn ((x, y), r) => f(x, y, r)) e (zip(xs, ys)).

    [foldl f e (xs, ys)] evaluates f(xn, yn, f( ..., f(x2, y2, f(x1, y1, e))))
    where xs = [x1, x2, ..., x(n-1), xn, ...],
          ys = [y1, y2, ..., y(n-1), yn, ...],
    and    n = min(length xs, length ys).
    Equivalent to List.foldl (fn ((x, y), r) => f(x, y, r)) e (zip(xs, ys)).
  *)
```

```
*)
```

B.10. Structure Math

```
(* Math -- SML Basis Library *)

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, that is,
   3.14159265358979323846.

   [e] is the base of the natural logarithm: 2.7182818284590452354.

   [sqrt x] is the square root of x.  Raises Domain if 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.
```

```
      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.
   *)
```

B.11. Structure Meta

```
   (* Meta -- functions available only in interactive Moscow ML sessions *)

   val printVal         : 'a -> 'a
   val printDepth       : int ref
   val printLength      : int ref
   val installPP        : (ppstream -> 'a -> unit) -> unit

   val liberal          : unit -> unit
   val conservative     : unit -> unit
   val orthodox         : unit -> unit

   val use              : string -> unit
```

```
val quit             : unit -> 'a

(*
   These values and functions are available in the Moscow ML
   interactive system only.

   [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 : ppstream -> ty -> unit
   at type ty.  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
```

functions: accept all extensions to the SML Modules language, but issue a warning for each use. The conservative mode may be set also by the mosml option -conservative. This is the default.

[orthodox ()] sets orthodox mode for the compilation functions: reject all uses of the extensions to the SML Modules language. That is, accept only SML Modules syntax. The orthodox mode may be set also by the mosml option -orthodox.

[compile "U.sig"] will compile and elaborate the specifications in file U.sig in structure mode, producing a compiled signature U in file U.ui. This function is backwards compatible with Moscow ML 1.44 and earlier. Equivalent to compileStructure [] "U.sig".

[compile "U.sml"] will elaborate and compile the declarations in file U.sml in structure mode, producing a compiled structure U in bytecode file U.uo. If there is an explicit signature file 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. This function is backwards compatible with Moscow ML 1.44 and earlier. Equivalent to compileStructure [] "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.

[compileStructure opnunits "U.sig"] compiles the specifications in file U.sig as if they form a signature declaration

```
     signature U = sig ... contents of U.sig ... end
```

The contents of opnunits is added to the compilation context in which the specifications in U.sig are compiled. The result is a compiled signature file U.ui. This corresponds to invoking the batch compiler as follows:

```
     mosmlc -c U1.ui ... Un.ui -structure U.sig
```

where opnunits equals ["U1", ..., "Un"].

[compileStructure opnunits "U.sml"] compiles the declarations in file U.sml as if they formed a structure declaration

```
     structure U = struct ... contents of U.sml ... end
```

---

where opnunits equals ["U1", ..., "Un"].

[compileToplevel opnunits "U.sml"] compiles the declarations in file U.sml, in a context in which all declarations from opnunits are visible, creating a bytecode file U.uo. If U.ui exists already, then the compiled declarations are matched against it; otherwise the file U.ui is created. This corresponds to invoking the batch compiler as follows

```
     mosmlc -c U1.ui ... Un.ui -toplevel U.sml
```

where opnunits equals ["U1", ..., "Un"].

[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.

```
      [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 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.  False by default.

      [valuepoly] determines whether value polymorphism is used or not in
      the type checker.  With value polymorphism (the default), there is
      no distinction between imperative ('_a) and applicative ('a) type
      variables, and type variables are generalized only in bindings to
      non-expansive expressions.  Non-generalized type variables are left
      free, to be instantiated when the bound identifier is used.  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.  True by default.
   *)
```

B.12. Structure Option

```
   (* Option -- SML Basis Library *)

   exception Option

   datatype option = datatype option

   val getOpt         : 'a option * 'a -> 'a
   val isSome         : 'a option -> bool
```

```
   [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.
*)
```

B.13. Structure Random

```
(* Random -- random number generator *)

type generator

val newgenseed : real -> generator
val newgen     : unit -> generator
val random     : generator -> real
val randomlist : int * generator -> real list
val range      : int * int -> generator -> int
val rangelist  : int * int -> int * generator -> int list

(*
   [generator] is the type of random number generators, here the
   linear congruential generators from Paulson 1991, 1996.

   [newgenseed seed] returns a random number generator with the given seed.
```

B.14. Structure Real

```
(* Real -- SML Basis Library *)

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
```

[abs x] is x if x >= 0, and ~x if x < 0, that is, the absolute value of x.

[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 the 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. 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).

```
    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]+)?
*)
```

B.15. Structure Regex

```
(* Regex -- regular expressions a la POSIX 1003.2 -- requires Dynlib *)

exception Regex of string

type regex                      (* A compiled regular expression         *)

datatype cflag =
    Extended                    (* Compile POSIX extended REs            *)
  | Icase                       (* Compile case-insensitive match        *)
  | Newline                     (* Treat \n in target string as new line *)

datatype eflag =
    Notbol                      (* Do not match ^ at beginning of string *)
  | Noteol                      (* Do not match $ at end of string       *)

val regcomp      : string -> cflag list -> regex

val regexec      : regex -> eflag list -> string -> substring vector option
val regexecBool  : regex -> eflag list -> string -> bool
```

```
val replace1     : regex -> replacer list -> string -> string
val replace      : regex -> replacer list -> string -> string

val substitute1  : regex -> (string -> string) -> string -> string
val substitute   : regex -> (string -> string) -> string -> string

val tokens       : regex -> string -> substring list
val fields       : regex -> string -> substring list

val map          : regex -> (substring vector -> 'a) -> string -> 'a list
val app          : regex -> (substring vector -> unit) -> string -> unit
val fold         : regex
                   -> (substring * 'a -> 'a) * (substring vector * 'a -> 'a)
                   -> 'a -> string -> 'a

(*
   This structure provides pattern matching with POSIX 1003.2 regular
   expressions.

   The form and meaning of Extended and Basic regular expressions are
   described below.  Here R and S denote regular expressions; m and n
   denote natural numbers; L denotes a character list; and d denotes a
   decimal digit:

        Extended    Basic       Meaning
       ---------------------------------------------------------------
        c           c           Match the character c
        .           .           Match any character
        R*          R*          Match R zero or more times
        R+          R\+         Match R one or more times
        R|S         R\|S        Match R or S
        R?          R\?         Match R or the empty string
        Rm        R\m\      Match R exactly m times
        Rm,       R\m,\     Match R at least m times
        Rm,n      R\m,n\    Match R at least m and at most n times
        [L]         [L]         Match any character in L
        [^L]        [^L]        Match any character not in L
        ^           ^           Match at string's beginning
        $           $           Match at string's end
        (R)         \(R\)       Match R as a group; save the match
```

```
[[:digit:]]     Match decimal digit; same as [0-9]
[[:graph:]]     Same as [:print:] but not [:space:]
[[:lower:]]     Match lowercase letter
[[:print:]]     Match printable character
[[:punct:]]     Match punctuation character
[[:space:]]     Match SML #" ", #"\r", #"\n", #"\t", #"\v", #"\f"
[[:upper:]]     Match uppercase letter
[[:xdigit:]]    Match hexadecimal digit; same as [0-9a-fA-F]
[[:lower:]čřÍ]  Match lowercase Danish letters (ISO Latin 1)
```

Remember that backslash (\) must be escaped as "\\" in SML strings.

[regcomp pat cflags] returns a compiled representation of the regular expression pat. Raises Regex in case of failure.

[cflag] is the type of compilation flags with the following meanings:

[Extended] : compile as POSIX extended regular expression.
[Icase] : compile case-insensitive match.
[Newline] : make the newline character \n significant, so ^ matches just after newline (\n), and $ matches just before \n.

Example: Match SML integer constant:
regcomp "^~?[0-9]+$" [Extended]

Example: Match SML alphanumeric identifier:
regcomp "^[a-zA-Z0-9][a-zA-Z0-9'_]*$" [Extended]

Example: Match SML floating-point constant:
regcomp "^[+~]?[0-9]+(\\.[0-9]+|(\\.[0-9]+)?[eE][+~]?[0-9]+)$" [Extended]

Example: Match any HTML start tag; make the tag's name into a group:
regcomp "<([[:alnum:]]+)[^>]*>" [Extended]

[regexec regex eflags s] returns SOME(vec) if some substring of s matches regex, NONE otherwise. In case of success, vec is the match vector, a vector of substrings such that vec[0] is the (longest leftmost) substring of s matching regex, and vec[1], vec[2], ... are substrings matching the parenthesized groups in pat (numbered 1, 2, ... from left to right in the order of their opening parentheses). For a group that does not take part in the

---

[regnexec regex eflags sus] returns SOME(vec) if some substring of sus matches regex, NONE otherwise. The substrings returned in the vector vec will have the same base string as sus. Useful e.g. for splitting a string into fragments separated by substrings matching some regular expression.

[regnexecBool regex eflags sus] returns true if some substring of sus matches regex, false otherwise. Equivalent to, but faster than, Option.isSome(regnexec regexec eflags sus).

[regmatch pat, tgt cflags eflags] is equivalent to
regexec (regcomp pat cflags) eflags tgt
but more efficient when the compiled regex is used only once.

[regmatchBool pat, tgt cflags eflags] is equivalent to
regexecBool (regcomp pat cflags) eflags tgt
but more efficient when the compiled regex is used only once.

[replace regex repl s] finds the (disjoint) substrings of s matching regex from left to right, and returns the string obtained from s by applying the replacer list repl to every such substring (see below). Raises Regex if it fails to make progress in decomposing s, that is, if regex matches an empty string at the head of s or immediately after a previous regex match.
Example use: delete all HTML tags from s:
replace (regcomp "<[^>]+>" [Extended]) [] s

[replace1 regex repl s] finds the leftmost substring b1 of s matching regex, and returns the string resulting from s by applying the replacer list repl to the match vector vec1 (see below).

Let x0 be a substring matching the entire regex and xi be the substring matching the i'th parenthesized group in regex; thus xi = vec[i] where vec is the match vector (see regexec above). Then a single replacer evaluates to a string as follows:

[Str s] gives the string s
[Sus i] gives the string xi
[Tr (f, i)] gives the string f(xi)
[Trs f] gives the string f(vec)

[map regex f s] finds the (disjoint) substrings of s matching regex from left to right, applies f to the match vectors vec1, ..., vecn, and returns the list [f(vec1), ..., f(vecn)]. Raises Regex if it fails to make progress in decomposing s.

[app regex f s] finds the (disjoint) substrings of s matching regex from left to right, and applies f to the match vectors vec1, ..., vecn. Raises Regex if the regex fails to make progress in decomposing s.

[fields regex s] returns the list of fields in s, from left to right. A field is a (possibly empty) maximal substring of s not containing any delimiter. A delimiter is a maximal substring that matches regex. The eflags Notbol and Noteol are set. Raises Regex if it fails to make progress in decomposing s.
Example use:

```
     fields (regcomp " *; *" []) "56; 23 ; 22;; 89; 99"
```

[tokens regex s] returns the list of tokens in s, from left to right. A token is a non-empty maximal substring of s not containing any delimiter. A delimiter is a maximal substring that matches regex. The eflags Notbol and Noteol are set. Raises Regex if it fails to make progress in decomposing s. Equivalent to

```
     List.filter (not o Substring.isEmpty) (fields regex s)
```

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 three fields: "abc" and "" and "def"
     "abc||def" contains two tokens:   "abc" and "def"
```

[fold regex (fa, fb) e s] finds the (disjoint) substrings b1, ..., bn of s matching regex from left to right, and splits s into the substrings

```
     a0, b1, a1, b2, a2, ..., bn, an
```

where n >= 0 and where a0 is the (possibly empty) substring of s preceding the first match, and ai is the (possibly empty) substring between the matches bi and b(i+1). Then it computes and returns

```
     fa(an, fb(vecn, ..., fa(a1, fb(vec1, fa(a0, e))) ...))
```

where veci is the match vector corresponding to bi. Raises Regex

---

B.16. Structure Splaymap

```
(* Splaymap -- applicative maps implemented by splay-trees       *)
(* From SML/NJ lib 0.2, copyright 1993 by AT&T Bell Laboratories *)

type ('key, 'a) dict

exception NotFound

val mkDict    : ('_key * '_key -> order) -> ('_key, '_a) dict
val insert    : ('_key, '_a) dict * '_key * '_a -> ('_key, '_a) dict
val find      : ('key, 'a) dict * 'key -> 'a
val peek      : ('key, 'a) dict * 'key -> 'a option
val remove    : ('_key, '_a) dict * '_key -> ('_key, '_a) dict * '_a
val numItems  : ('key, 'a) dict -> int
val listItems : ('key, 'a) dict -> ('key * 'a) list
val app       : ('key * 'a -> unit) -> ('key,'a) dict -> unit
val revapp    : ('key * 'a -> 'b) -> ('key,'a) dict -> unit
val foldr     : ('key * 'a * 'b -> 'b)-> 'b -> ('key,'a) dict -> 'b
val foldl     : ('key * 'a * 'b -> 'b) -> 'b -> ('key,'a) dict -> 'b
val map       : ('_key * 'a -> '_b) -> ('_key,'a) dict -> ('_key, '_b) dict
val transform : ('a -> '_b) -> ('_key,'a) dict -> ('_key, '_b) dict

(*
   [('key, 'a) dict] is the type of applicative maps from domain type
   'key to range type 'a, or equivalently, applicative dictionaries
   with keys of type 'key and values of type 'a.  They are implemented
   as ordered splay-trees (Sleator and Tarjan).

   [mkDict ordr] returns a new, empty map whose keys have ordering
   ordr.

   [insert(m, i, v)] extends (or modifies) map m to map i to v.

   [find (m, k)] returns v if m maps k to v; otherwise raises NotFound.

   [peek(m, k)] returns SOME v if m maps k to v; otherwise returns NONE.

   [remove(m, k)] removes k from the domain of m and returns the
   modified map and the element v corresponding to k.  Raises NotFound
```

```
    [foldl f e m] applies the folding function f to the entries (k, v)
    in m, in increasing order of k.

    [foldr f e m] applies the folding function f to the entries (k, v)
    in m, in decreasing order of k.

    [map f m] returns a new map whose entries have form (k, f(k,v)),
    where (k, v) is an entry in m.

    [transform f m] returns a new map whose entries have form (k, f v),
    where (k, v) is an entry in m.
*)
```

B.17. Structure Splayset

```
(* Splayset -- applicative sets implemented by splay-trees       *)
(* From SML/NJ lib 0.2, copyright 1993 by AT&T Bell Laboratories *)

type 'item set

exception NotFound

val empty        : ('_item * '_item -> order) -> '_item set
val singleton    : ('_item * '_item -> order) -> '_item -> '_item set
val add          : '_item set * '_item -> '_item set
val addList      : '_item set * '_item list -> '_item set
val retrieve     : 'item set * 'item -> 'item
val peek         : 'item set * 'item -> 'item option
val isEmpty      : 'item set -> bool
val equal        : 'item set * 'item set -> bool
val isSubset     : 'item set * 'item set -> bool
val member       : 'item set * 'item -> bool
val delete       : '_item set * '_item -> '_item set
val numItems     : 'item set ->  int
val union        : '_item set * '_item set -> '_item set
val intersection : '_item set * '_item set -> '_item set
val difference   : '_item set * '_item set -> '_item set
val listItems    : 'item set -> 'item list
val app          : ('item -> unit) -> 'item set -> unit
```

[singleton ordr i] creates the singleton set containing i, with the given ordering relation.

[add(s, i)] adds item i to set s.

[addList(s, xs)] adds all items from the list xs to the set s.

[retrieve(s, i)] returns i if it is in s; raises NotFound otherwise.

[peek(s, i)] returns SOME i if i is in s; returns NONE otherwise.

[isEmpty s] returns true if and only if the set is empty.

[equal(s1, s2)] returns true if and only if the two sets have the same elements.

[isSubset(s1, s2)] returns true if and only if s1 is a subset of s2.

[member(s, i)] returns true if and only if i is in s.

[delete(s, i)] removes item i from s. Raises NotFound if i is not in s.

[numItems s] returns the number of items in set s.

[union(s1, s2)] returns the union of s1 and s2.

[intersection(s1, s2)] returns the intersectionof s1 and s2.

[difference(s1, s2)] returns the difference between s1 and s2 (that is, the set of elements in s1 but not in s2).

[listItems s] returns a list of the items in set s, in increasing order.

[app f s] applies function f to the elements of s, in increasing order.

[revapp f s] applies function f to the elements of s, in decreasing order.

[foldl f e s] applies the folding function f to the entries of the

```
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 immutable strings of characters, with
   constant-time indexing.

   [maxSize] is the maximal number of characters in a string.
```

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.

[fromString s] scans the string s as an ML source program string,
converting escape sequences into the appropriate characters. Does
not skip leading whitespace.

```
      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.

      [<]
      [<=]
      [>]
      [>=] compare strings lexicographically, using the representation
      ordering on characters.
   *)
```

B.19. Structure StringCvt

```
   (* StringCvt -- SML Basis Library *)

   datatype radix = BIN | OCT | DEC | HEX

   datatype realfmt =
       SCI of int option   (* scientific,  arg = # dec. digits, dflt=6 *)
     | FIX of int option   (* fixed-point, arg = # dec. digits, dflt=6 *)
     | GEN of int option   (* auto choice of the above,                *)
                           (* arg = # significant digits, dflt=12      *)

   type cs                 (* character source state *)

   type ('a, 'b) reader = 'b -> ('a * 'b) option

   val scanString : ((char, cs) reader -> ('a, cs) reader) -> string -> 'a option

   val splitl     : (char -> bool) -> (char, 'a) reader -> 'a -> string * 'a
   val takel      : (char -> bool) -> (char, 'a) reader -> 'a -> string
   val dropl      : (char -> bool) -> (char, 'a) reader -> 'a -> 'a
   val skipWS     : (char, 'a) reader -> 'a -> 'a

   val padLeft    : char -> int -> string -> string
   val padRight   : char -> int -> string -> string
```

```
   A character source scanner takes a character source reader getc as
   argument and uses it to scan a data value from the character
   source.

   [scanString scan s] turns the string s into a character source and
   applies the scanner 'scan' to that source.

   [splitl p getc src] returns (pref, suff) where pref is the
   longest prefix (left substring) of src all of whose characters
   satisfy p, and suff is the remainder of src.  That is, the first
   character retrievable from suff, if any, is the leftmost character
   not satisfying p.  Does not skip leading whitespace.

   [takel p getc src] returns the longest prefix (left substring) of
   src all of whose characters satisfy predicate p.  That is, if the
   left-most character does not satisfy p, the result is the empty
   string.  Does not skip leading whitespace.  It holds that
        takel p getc src = #1 (splitl p getc src)

   [dropl p getc src] drops the longest prefix (left substring) of
   src all of whose characters satisfy predicate p.  If all characters
   do, it returns the empty source.  It holds that
        dropl p getc src = #2 (splitl p getc src)

   [skipWS getc src] drops any leading whitespace from src.
   Equivalent to dropl Char.isSpace.

   [padLeft c n s] returns the string s if size s >= n, otherwise pads
   s with (n - size s) copies of the character c on the left.
   In other words, right-justifies s in a field n characters wide.

   [padRight c n s] returns the string s if size s >= n, otherwise pads
   s with (n - size s) copies of the character c on the right.
   In other words, left-justifies s in a field n characters wide.
*)
```

B.20. Structure TextIO

```
(* TextIO -- SML Basis Library *)
```

```
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.
```

[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.

```
        [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.

        [print s] outputs s to stdOut and flushes immediately.


        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.
```

```
   [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.
*)
```

B.21. Structure Time

```
(* Time -- SML Basis Library *)

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
```

```
        [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.

        [fromReal 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.

        [toReal 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.
```

```
   [>]
   [>=] compares time values.  For instance, for reals r1, r2 >= 0.0
   it holds 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.
*)
```

B.22. Structure Timer

```
(* Timer -- SML Basis Library *)

type cpu_timer
type real_timer

val startCPUTimer  : unit -> cpu_timer
val totalCPUTimer  : unit -> cpu_timer
val checkCPUTimer  : cpu_timer ->
                      usr : Time.time, sys : Time.time, gc : Time.time

val startRealTimer : unit -> real_timer
val totalRealTimer : unit -> real_timer
val checkRealTimer : real_timer -> Time.time

(*
   [cpu_timer] is the type of timers for measuring CPU time consumption
   (user time, garbage collection time, and system time).

   [real_timer] is the type of timers for measuring the passing of
   real time (wall-clock time).

   [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
```

B.23. Structure Word

```
(* Word -- SML Basis Library *)

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 toInt      : word -> int
```

---

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 2 raised to the n'th power, where n=wordSize. The operations (i div j) and (i mod j) raise Div when j=0. Otherwise no exceptions are raised.

[<]
[<=]
[>]
[>=] compare words as unsigned integers.

```
   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.

```
*)
```

```
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] equals 8.
```

[~>>(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.

[<]
[<=]
[>]
[>=] compare words as unsigned integers.

[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]+

---

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.
*)