

BUDAPESTI MŰSZAKI EGYETEM
VILLAMOSMÉRÖKI ÉS INFORMATIKAI KAR

DEKLARATÍV PROGRAMOZÁS

OKTATÁSI SEGÉDLLET

Bevezetés a funkcionális programozásba

Harmadik, átdolgozott kiadás

Hanák D. Péter

Irányítástechnika és Informatika Tanszék

Budapest, 2000

Tartalomjegyzék

1. Bevezetés	9
1.1. A programozási paradigmákról	9
1.2. Az imperativ programozási paradigmára	9
1.3. A deklaratív programozási paradigmára	10
1.3.1. A logikai programozási paradigmára	10
1.3.2. A funkcionális programozási paradigmára	11
1.4. Egyszerű példák SML-ben	11
1.4.1. A funkcionális program	11
1.4.2. Negyezre emelés	12
1.4.3. A <i>read-end-print</i> ciklus	12
1.4.4. Penzváltás	12
1.5. Hivatalosítási általánosság	12
1.6. A függvények tárba	13
1.7. Funkcionális nyelvek jellemzői	13
1.8. Az SML fontosabb jellemzői	13
1.9. SML-érdekmények és fordítók	15
1.10. Információforrások	15
1.11. Kiszöntetűírásban	15
1.12. Hibajelentés	16
2. Nevek, függvények, egyszerű típusok	19
2.1. Értékkételelés	19
2.1.1. Névadás állandonak	19
2.1.2. Névadás függvények	20
2.1.3. Nevek tíjárdefiníciója	21
2.1.3.1. Nevek képzése	21
2.2. Egész, valós, fizér, karakter és más egyszerű típusok	22
2.2.1. Egészek és valósak	22
2.2.1.1. A <code>real</code> , <code>floor</code> , <code>ceil</code> , <code>abs</code> , <code>round</code> és <code>trunc</code> függvény	22
2.2.1.2. Alapnövelek előjel esetén	23
2.2.1.3. Alapnövelek valós számokkal	24
2.2.1.4. Precedencia	24
2.2.1.5. Alapnövelek előjel nélküli egészekkel	25
2.2.1.6. Típusnemkötés	26
2.2.2. Fizérek	26
2.2.2.1. Escape-székvenciók	27
2.2.2.2. Gyakori műveletek fizérekkel	27
2.2.3. Karakterek	27
2.2.3.1. Gyakori műveletek karakterekkel	28
2.2.4. Igazságértékek, logikai kifejezések, feltétel esetén kifejezések	28
3. Párok, ennek, rekordok	31
3.1. Pár, ennek	31
3.1.1. Tipuskifejezés	31
3.1.2. Példa: vektorok	31
3.1.2.1. Függvény több argumentummal és eredménnyel	31
3.1.2.2. Gyakorló feladatak	32
3.1.2.2.1. Gyenes eleminek kiválasztása mintaillesztéssel	33
3.1.2.2.2. A nulla	33
3.1.4. Mi a különbség?	33
3.2. Rekordminta	33
3.2.1. Rekordminta	33
4. Kifejezések	37
4.1. Újra az infix operátorról	37
4.1.1. Infix operátor kölcsönös	38
4.1.2. Kifejezések körbefelése az SML-ben	39
4.2.1. Moho körbefelés	40
4.2.1.1. Moho körbefelés rekurzív függvények esetén	40
4.2.1.2. Iteratív függvények	40
4.2.1.3. Felételes kifejezések speciális körbefelés	41
4.2.1.4. Gyakorló feladat	42
4.2.2. Lusia körbefelés	42
4.2.3. A mohó és a lusia körbefelés összavetése	42
5. Lokalis kifejezés, lokalis és egysidejű deklaráció, polimorf típusellenőrzés	45
5.1. Lokalis kifejezés	45
5.1.1. Lokális kifejezés	45
5.2. Lokális deklaráció	45
5.3. Egyidejű deklaráció	45
5.4. Polimorf típusellenőrzés	46
6. Listák	49
6.1. Tipuskifejezések és típusoperátorok	49
6.1.1. Lista létrehozása	49
6.1.2. Egyesű műveletek listákra	50
6.1.3. Lista elemeinek szorzata	50
6.3.2. Lista legnagyobb eleme	51
6.3.3. Karakter, fizér és lista	51
6.4. Listák vizsgálata és darabokra szedése	52
6.5. Listák és egész számok	52
6.6. Listák összefűzése és megfordítása	54
6.7. Listákból álló lista, párokbeli álló lista	55
7. Rekurzív függvények	57
7.1. Egész körbefelő hat változás	57
7.2. Fibonacci-számok	58
7.3. Egész negyzetgyökök körbefeléssel	59
7.4. Valós szám negyzetgyöke Newton-Raphson módszerrel	60
7.5. $\pi/4$ körbefelő értéke körösökös rekurzióval	61

7.6. A következő permutált	62
8. Polimorfizmus	67
8.1. Polimorf halmozániűveletek	68
9. Adattípusok	71
9.1. A datatype deklaráció	71
9.2. A felsorolásos típus	72
9.3. Polimorf adattípusok	73
9.4. A case-kifejezés	74
10. Részlegesen alkalmazható függvények	75
10.1. Az fn jelölés	75
10.1.1. Függvény definíciója fun, val és val rec kulcsszóval	76
10.2. Részlegesen alkalmazható függvények	76
10.3. Függvény mint argumentum és mint eredmény	77
11. Magasabb rendű függvények	79
11.1. Magasabb rendű függvények	79
11.1.1. sec és secr	79
11.1.2. Kombinátorok	80
11.1.2.1. Két függvény kompozíciója	80
11.1.2.2. Az S, a K és az I kombinátor	81
11.1.3. map és filter	81
11.1.4. takewhile és dropwhile	82
11.1.5. exists és forall	82
11.1.6. foldl és foldr	83
11.1.7. További rekurzív függvények	85
11.1.8. curry és uncurry	85
11.1.9. map újratelvezetése foldr-rel	86
12. Kivételezés	87
12.1. Kivétel deklaráása az exception kulcsszóval	87
12.2. Kivétel jelzése a raise kulcsszóval	87
12.2.1. Belép kivételek	88
12.3. Kivétel feldolgozása a handle kulcsszóval	88
12.4. Néhány példa a kivételezésre	88
13. Listák használata: rendezés	91
13.1. Beszűrő rendezés	91
13.1.1. Generikus negoldások	91
13.1.2. Beszűrő rendezés foldr-rel és foldl-lel	92
13.1.3. A futási idők összehasonlítása	92
13.2. Gyorsrendezés	93
13.3. Összefüggő rendezés	94
13.3.1. Fölfelről lefelé haladó összefüggő rendezés	94
13.3.2. Alulról fölfelé haladó összefüggő rendezés	95
13.4. Simarendezés	97
14. Bináris fák	99
14.1. Egyszerű műveletek bináris fákon	100
14.2. Lista elbállítása bináris fa elemeiből	102
14.3. Bináris fa elbállítása lista elemeiből	103
14.4. Elem törlése bináris fához	104
14.5. Bináris keresésfák	105
15. Lista lista	107
15.1. Elémi feldolgozási műveletek sorozatokkal	108
15.2. Magasabb rendű függvények sorozatokra	109
15.3. Néhány összetett példa	110
15.3.1. Álvételek-számok	110
15.3.2. Prímszámok	111
15.3.3. Numerikus számítások	111
15.4. Sorozatok sorozata és egyrészt bája ékelése	112
15.4.1. Kereszszorzatokból álló lista	112
15.4.2. Kereszszorzatokból álló sorozat	113
16. Példaprogramok: fák kezelése	115
16.1. Fa adott tulajdonságának ellenőrzése (ugyanannyi)	115
16.2. Fa adott tulajdonságú részfáinak száma (bea)	117
16.3. Fa adott tulajdonságú részfáinak száma (testverE)	118
16.4. Fa adott elemének összegére (szintössz)	119
16.5. Kifejezésfa egyszerűítése (egyszerűsít)	121
16.6. Kifejezésfa egyszerűítése (coeft)	122
16.7. Szövegfeldolgozás (parPairs)	123
17. Példaprogramok: füzerek és listák kezelése	125
17.1. Füzer adott tulajdonságú elemei (mezők)	125
17.2. Füzer adott tulajdonságú eleméi (basenamé)	126
17.3. Füzer adott tulajdonságú elemei (rootname)	127
17.4. Lista adott tulajdonságú részlistái (szonson)	128
17.5. Bináris számok inkrementálása (binc)	128
17.6. Mátrix transponálja (trans)	130
18. Az SML szintaxisa	133
18.1. Fogalmak és jelölések	133
18.1.1. Novel	133
18.1.2. Infix operátorok	134
18.1.3. Jelölések	134
18.2. Az SML Core language szintaxisa	135
18.2.1. Kifejezések és klözsorozatok	135
18.2.2. Declarációk és kötések	136
18.2.3. Tipuskifejezések	137
18.2.4. Minák	137
18.2.5. Szintaktikai korlátozások	137
A. Válogatás az SML '97 könyvtárból	139
A.1. Structure Binaryset	141
A.2. Structure Bool	143
A.3. Structure Char	144
A.4. Structure General	148
A.5. Structure Int	150

A.6. Structure Intset	153
A.7. Structure List	155
A.8. Structure Listsort	158
A.9. Structure Math	159
A.10. Structure Meta	161
A.11. Structure Option	165
A.12. Structure Random	166
A.13. Structure Real	167
A.14. Structure Regex	170
A.15. Structure String	175
A.16. Structure StringCvt	178
A.17. Structure Substring	180
A.18. Structure Susp	185
A.19. Structure TextIO	186
A.20. Structure Time	190
A.21. Structure Timer	192
A.22. Structure Vector	193
A.23. Structure Word	196
A.24. Structure Word8	199

- eljárásek, függvények, biztonságos vezérlési szerkezetek,
- elemi és összetett adattípusok, absztrakt adattípusok, osztályok, objektumok, valamint önműfordítási egységek, kapcsolatlerősök, generikus programrészek megjelenése, használata jellemzi.

1. fejezet

Bevezetés

Ez a jegyzet oltatási segéddokumentum a Deklaratív programozás (korábban *Programozási paradigmák*) c. tárgy funkcionális programozással foglalkozó részhez készült.

1.1. A programozási paradigmákról

A programozási paradigmák két alapfogalmakat felhasználják valamely programozási területen: *Ilegén szavak és kifejezések szóhárna*² szerint görög-latin eredetű, és két jelentése is van:

1. bizonításra vagy összehasonlításra alkalmazott példa;
2. (nyelvtani) ragozásti minta.

Az *Akadémiai Kislexikon*³ a fenti két jelentést említi:

3. valamely tudományterület sarkalatos megállapítása.

Programozási paradigmának nevezünk:

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 paradigmának két alapípusa van: *imperativ* és *deklaratív*.

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

- lineáris szerkezetű, azaz a programszövegen egy más után álló programelemeket rendszerint az adott sorrendben kell végrehajtani vagy kiérkezni, és nincsenek benne bonyolultabb adatszerzetek;
- egyetlen fordítási egység, azaz változás esetén a teljes programszöveget újra kell fordítani.

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

²1999-ig a most Deklaratív programozásnak nevezett tantárgynak Programozási paradigmák volt a neve.

³Akadémiai Kiadó, Budapest 1989

1.2. Az imperatív programozási paradigmára

A több évtizedes tapasztalatok és kutatások megrátoltatták a programozás mibenlétéről kialakult képet: egyre nagyobb jelentőséget tulajdonítunk a követelmények elemzésének, a (formális) *spezifikációk*, a módszeres és szabványos *tervezésnek* és *dokumentálásnak*, a *programmátrajznak*, a *karbantartásnak*, a *módosításiágynak*, a *változáskövetésnek*, a *hordozhatóságynak*, a *minőségynek*, és egyre kevésbé magának a kódolásnak. E tekintetben itt elsősorban a *specifikáció* és a *megtöltsés*, a *mit* és a *hogyan* szérváltozásának fontosságát emeljük ki.

Az imperatív⁴ (inás néven procedurális) programozási paradigmára a legelterjedtebb, a legrégebbi, erősen kötődik a Neumann-féle számítógép-architektúrához. Két közé jellemzője a *parancs* és az *állapot*.

A program *állapottere* az a sokdimenziós tér, amelyet a program változónak értelmezni tartalmaz. A program állapotának állapotnámi állapotát váltózónak illusztrálni lehet, a *memória* neve. A program állapotát *értekelődésssel* – azaz a változók frissítésével – változtathatjuk meg. Állapotváltozás nélkül nem lehet, de legalábbis körtülményes modellezni az irodát, a valós világ jeleniségeit.

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 mindenek az alapja, hiszen pl. bármely párhuzamos program szekvenciális programozáshoz kötőd. A parancsokból, minthi jó tudunk, *vezérelsi szerkezetek* – felsorolás, valaszthatás, ismétlés – felhasználásával összevetít páratlosokat, *absztraktiósai* pedig eljáráskat hozunk létre; ezért szoktunk az imperatív programozásról mint procedurális programozásról beszélni.

A valós idejű programozás erősen kötődik a párhuzamos és elosztott programozáshoz, u. valós idejű rendszerekben egyes programozések rendszereit egységekben, egymással párhuzamosan kell végrehajtanunk. Párhuzamos végrehajtásra ugyanakkor más esetekben, pl. numerikus számítások elvégzéséhez, aritmetikai kifejezések kivételekések kor is szükség lehet.

A ma oly divatos objektum-orientált programozás az imperatív programozási paradigmával egyik válfaja, szoros rokonágban 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 között a FORT-RAN, a COBOL és az eredeti Pascal alig, a Turbo, Borland) Pascal és a C inkább, az Ada, a Modula, a C++ és a Java még inkább támogatja e szétválasztást. Azonban sikertőlőn bárminivel jól a szétválasztás, a feladatot megoldó algoritmusok megránya a programozó dolga marad.

⁴Latin szó, jelentése: parancsoló (vő: imperativus, imperátor).

1.3. A deklaratív programozási paradigmá

Az imperatív stílussal ellentétben a deklaratív⁵ stílusban programozanak – elvileg – csak azt kell megmondania, hogy *mit* akarunk, az algoritmust, az értelmező- vagy fordítóprogram állítsa elő. A deklaratív programozás rét válját szókás megtüntözheti: a logikai és a funkcionális programozást.

1.3.1. A logikai programozási paradigmá

A programozási paradigmák közül, amint a neve is mutatja, a legérősebbben kötődik a matematikai logikához. Jellemző:

- a tények,
- a szabályok, és
- a következtelőrendszer.

A legelterjedtebb logikai programozási nyelv a Prolog. Professzionális, gyakorlati feladataik megoldására alkalmazás megalapozásai a deklaratív nyelvi elemek mellett imperativ elemeket is taralmazznak. Természetesen más logikai programozási nyelvek is vannak, pl. az OPS5, a CLP, a Mercury. (Az utóbbit a Prolog-öt átvett logikai programozási elemeket a típusfogalommal és a funkcionális programozást támogató nyelvi elemekkel egészíti ki.)

1.3.2. A funkcionális programozási paradigmá

A funkcionális⁶ programozás két fő jellemzője

- az érték (a függvényeket is értékel) és
- a függvényalkalmazás.

A funkcionális programozás nevét a függvények kitüntetett szerepének köszönheti. A tiszán funkcionális programozási nyelvek a matematikában megszokott függvénymagainat valósítják meg: *a függvény egyértelmű leképzés a függvény argumentuma és eredménye között*, a függvény alkalmazásának nincs semmilyen más hatása. Tiszán funkcionális programozás esetén telítő nincs állapot, minős (mellek)hatás, nincs értékalkalmas.

Funkcionális program pl. az $\lambda e. e$ kifejezés, ahol az e -nek függvényéről érdekményező kifejezésnek kell lennie, az $\lambda e. e$ pedig tetszőleges kifejezés lehet. A matematikában megszokott módon zérust mondjuk, hogy az e függvényt (vagy kissé körülmenetben: a függvény alkalmazásáról) lévő szó, funkcionális programozásban esetén telítő nincs állapot, szinonimaként gyakran *application* programozásról beszélünk.

Az applikatív programozás elnétele a λ -kalkulus (lambda-kalkulus), az a függvényelnelet, amelyet Alonzo Church az 1930-as években dolgozott ki, majd Moses Schönfinkel és Haskell Curry fejlesztett tovább. A λ -kalkuluson alapul elso funkcionális programozási nyelv, a LISP-öt (List Programming). John McCarthy dolgozta ki az 1950-es évek közepe, 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 nyelkek.

Az első típusos funkcionális nyelv az ML (Meta Language) egyik korai változata volt a 70-es évek közepén, amelyben R. Milner megalakította típusellenőri eredményeit. Eredetileg *logikai algoritmusok gyártására, teljesítőkészítésre* terveztek, erre utal a nem túl ötletek *Meta Language* elnevezés –

⁵ Szintén latin szó, jelentése: kijelentő, kinyilatkozatot (vá) dokumentáció.

⁶ Vannak, akik a deklaratív programozást a logikával azonosítják, és a funkcionális programozást nem tekintik deklaratívnak.

Szintén latin szó, jelentése: alkalmazó (vá) alkalmazásával megoldani. Két esetet kell megkülönböztetnünk:

is. A HOPE-pal és más funkcionális nyelvekkel szemben tapasztalatok alapján dolgozták ki a Standard ML (SML) nyelvet a 80-as évek közepétől kezdve. Számos megalapozására készült el többfélé számtípusekre, és természetesen megszülettek különösfélé dialektusai is, pl. a Caml. A SML-családba tartozó nyelvek, kevés kivétellel, ún. *modulo körterhelést*, azaz érték szerinti paraméterteradást alkalmaznak. Ez azt jelenti, hogy amikor egy függvény kifejezést alkalmazunk egy argumenetu művelettel, akkor az SML-értelmező először az argumentumot értékel ki, és csak ezután lát hozzá a függvénykifejezés kiértékeléséhez.

A Miranda, az 1990-ben megjelent Haskell, és a még újabb Clean nyelv szemben *hasura kiértekelést* használ. A hasura körterhelés az 1960-as években alkalmazott *név számlári paraméterteradás* modern, lesemanticszerű átverésével. Ezben a Pacbelben és más nyelveken használt *cím szerinti paraméterteradással*. Megtegyezendő, hogy Lazy SML-neyen az SML-nek is van hasura kiértekelésű változata.

Az SML – akárcsak a körülömben számtalan más nyelv – gyakorlati programozási feladatait megholdására készül, ezért nemcsak a tiszta funkcionális, hanem az imperativ szílusú programozáshoz szükséges nyelvi elemek is megtalálhatók benne. Frissítésekkel változik tömbök, mellékhatással járó függvények stb., továbbá a nagyban programozást segítő fejlett modulrendszer.

A Miranda, az 1990-ben megjelent Haskell, és a még újabb Clean nyelvben alkalmazott név számlári paraméterteradás modern, lesemanticszerű átverésével. Ezben a Pacbelben és más nyelveken használt cím szerinti paraméterteradással. Megtegyezendő, hogy Lazy SML-neyen az SML-nek is van hasura kiértekelésű változata.

Az SML – akárcsak a körülömben számtalan más nyelv – gyakorlati programozási feladatait megholdására készül, ezért nemcsak a tiszta funkcionális, hanem az imperativ szílusú programozáshoz szükséges nyelvi elemek is megtalálhatók benne. Frissítésekkel változik tömbök, mellékhatással járó függvények stb., továbbá a nagyban programozást segítő fejlett modulrendszer.

1.4. Egyeszerű példák SML-ben

1.4.1. 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 *definíciója* a kvárt tulajdonságával, kiszámítástra nem csupán *ellenőrzésre* alkalmas. A kiszámítási szabályban a keresett mennyiséget csak az egyenlőségejel bal oldalán fordulhat elő, pl. *sqrt*(x) = ...

1.4.2. Négyzetre emelés

A funkcionális program: mennyiségek közötti kapcsolatokat leíró egyenletek halmaza. Pl. a *square*(x) = $x * x$;
 > val square = fn : int -> int
 az *sqrt*(x) * *sqrt*(x) = x alakú egyenlet csak *definíciója* a kvárt tulajdonságával, kiszámítástra nem csupán *ellenőrzésre* alkalmas. A kiszámítási szabályban a keresett mennyiséget csak az egyenlőségejel bal oldalán fordulhat elő, pl. *sqrt*(x) = ...

1.4.3. A *read-eval-print* ciklus

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

1.4.4. Pénzváltás

Adott különböző címeltű pénzérémek érték szerinti csökkenő sorrendű listája, pl.:

[20, 10, 5, 2, 1]

Most olván SML-programot írnunk, amely tétszéleges összeget apróra vált, és az eredményt ugyanakkor ismét adja vissza! Feltessük, hogy a megadott összeg minden felvátható, azaz az érmék között van 1-es eredmény. Az SML-program tanulmányozása előtt azt javasoljuk az olvasónak, hogy oldja meg a feladatot valamilyen általa ismert programozási nyelven.

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

1. a felváltandó összeg 0,

2. a felváltandó összeg nem 0.

Az 1. (triviális) esetben semmit nem kell tennünk, a feladat meg van oldva. A 2. esetben megpróbálunk visszavezetni a feladatot egy másik ismert részfeladatara.

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

> val change = fn : int * int list -> int list

Nézzük a jelöléseket! A példában felkóvertrel a nyelv *kuccsawat*, betűvel írt *terminális szimbólumokat* jelöli. A *val* és *fn* két funkcióneve, melyeket a következőkben mindenhol használnak. A *change* nevű függvény a többi minta után a definíciótól eltérően. A *sum* paraméter a többi mintához hasonlóan a *change* függvénytől eltérően, hogy az SML-ben minden függvényt egy paramétereit tekintünk. A paraméterek megfogadott értékei alapján *minimállesztés* állásunkat az esetek közül. Ha a paraméter *konkrét érték* (pl. most 0), akkor az SML-értelmező az adott ágat csak akkor hajtja végre, ha a függvényt ponosan ilyen értékű argumentummal hívjuk meg. Ha a paraméter *nev* (más szóval *azonosító*, most pl. *coins* vagy *sum*), akkor az tétszéges *minimállesztés* állapotba illeszkedik. A (*coin* :: *coins*) olyan összetett minta, amely legalább egyelemű (most: egész számokból álló) lista illeszkedik: *coins*-nak egy eleme, *coins*-nak egy – esetleg töres – lista formájára kell illeszkednie. (Jelölésszabányt később egyszerűsíténi fogjuk.)

A program helyessége: könyen belátható: Ha a felváltandó összeg 0, az eredmény az üres lista. Ha a felváltandó összeg nem 0, két további esetet kell megkülönböztetni az *if*-*then*-*else* felteles operátor alkalmazására! (Mit gondol, használhatnánk-e feliratkozás kifejezést miniallesztés helyett a 0 és a 1 közötti összeg? Ha a felváltandó összeg (*sum*) nem kisebb a soron következő címletnél (*coin*), akkor ez jó érték, és be kell raktani annak az eredménylistájának az előjére, amelyet úgy tapunk, hogy a maradék összeget (*sum* - *coin*) is megpróbálunk felváltani ugyanúgy, mint a kisebb értékű címletekkel (*coin*::*coins*). De ha a felváltandó összeg kisebb a soron következő címletnél, akkor ez nem jó érték, és a váltást a soron következő címleettel kell megrögzálni.

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

1.5. 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 *eugenői helyettesíthetők eugenőkkel*, pl.: egy kifejezés az értékekkel, az $E_1 + E_2$ kifejezes az $E_2 + E_1$ kifejezes (ahol a + jel a kommutatív aritmetikai összeadás jelent).⁸

A hivatkozási átlátszóság megléte esetén egy *kifejezés* értelme, *jelenlése* egyszerűen a kértekélezésének az eredménye, és ezért egyes részkifejezéseket egymástól függetlenül lehet kiértékelni. Ezzel szemben egy *parancs* végrehajtása azt jelenti, hogy a program *állapotát* megváltozik, vagyis a parancs megerősítései meg kell érteni a parancs *hatását* a program teljes *állapotára*.

⁸Feltessük, hogy felváltandó összegnek csak nemnegatív számot adunk meg. Később látni fogjuk, hogy mit tehetünk a libbás bennei adatok kiszövéséért.

Lássunk egy példát, nézzük a jó 1-szert *euklideszi algoritmus* egy megvalósítását a legmagyarobb közös osztó kiszámítására! Matematikai definíciója (feltessük, hogy $0 \leq m \leq n$):

```
gcd(0, n) = n
gcd(m, n) = gcd (n mod m, m), ha  $m > 0$ 
```

Egy lehetőséges kódoldása Pascalban:

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

és egy változata SML-ben:

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

Az utolsóhoz hasonló programot Pascalban is lehet írni, csak hogy a Pascalban kevésbé hatékony a rekurzió megvalósítása, és több tölfelkészüveget kell írunk:

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

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

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

1.6. A függvényértékről

A függvény általános programozási nyelvekben

- a függvényalkalmazás eredménye lehet
 - tétszéges típusú érték és
 - függvény;
- a függvény tétszéges adatszerkezet eleme lehet
 - az $f(z) + u/2$ és az $u/2 + f(z)$ kifejezés eredménye minden egyszerű, mert az f nem változtathatja meg az u -t (ezzel szemben a Pascalban pl. egy kifejezés kiértékelésekor figyelmen kívül venni a részkifejezések értékét és az esetleges mellekfáshatásukat a program állapotára.)

Mindez az SML-ben is lehetséges, annak ellenére, hogy az SML nem tisztá funkcionális nyelv.

1.7. Funkcionális nyelvek jellemzői

A funkcionális nyelveket többek között az alábbiak használata jellemzi:

- függvényhívás és rekurzió mint elsőleges vezérlesi szerkezet,
- rekurzív adattípusok: listák és fák,
- magasabb rendű függvények (*higher-order functions*, *functionals*: olyan függvények, amelyek más függvényeken végeznek műveleteket vagy függvénytéket adnak eredményül), pl.

```
map f [x1, ..., xn] = [f(x1), ..., f(xn)]
reduce f e [x1, ..., xn] = f(....f(f(e, x1), x2), ..., xn)
```

(Ha például $e = 0$ és $f = +$, a reduce eredménye:

$$\dots((0 + x_1) + x_2) \dots + xn$$

és ha $e = 1$ és $f = *$, a reduce eredménye:

$$\dots((1*x_1)*x_2)\dots)*xn$$

e -nek és f -nek természetesen más érték is választható.)

- végtelen adatszerkezetek (listák, fák stb.),
- lista kiértékelés.

1.8. Az SML fontosabb jellemzői

A sikeres programozási nyelvezet meghatározott célra tervezették. Lássunk néhány példát:

- LISP: mesterséges intelligencia,
 - FORTRAN: numerikus számítások,
 - PROLOG: természeti nyelvek feloldogzása,
 - Pascal: strukturált programozás oktatása,
 - C: gépközelű programozás magasabb szinten,
 - BASIC: interaktív számlítógéphasználat.
- Az általános célú nyelvkhöz (pl. ALGOL60 és ALGOL68) főleg ötleteket lehetett meríteni, gyakorlati eszközökkel kevésbé voltak be.
- Az SML-t *tételezonytasa* tervezések Edinburgh-ban (LCF = Logic for Computable Functions). Az SML előszörban funkcionális programozásra használható, de megtalálhatók benne az imperatív programozáshoz szükséges, fejlett nyelvi elemek is:
- jó olvasható szint axis,
 - mintálezés destruktif függvények helyett,
 - változatok,
 - frissíthető változók hiányá,

1.9. SML-értelmezők és fordítók

- hivatkozási átlátszóság,
- mellékeltásmennesség,
- polimorf típusok,
- absztrakt adattípusok,
- erős típusosság,
- típusvezetés,
- rekurzió,
- magasabb rendű függvények,
- fejlett modularendszer.

1.9. SML-értelmezők és fordítók

Az SML-nyelvnek két szintje van. A nyelv magját a *Core Language* képezi a nagyobb programok írását a *Module Language* támogatja. A nyelvbe beépített elemeket (az ún. belső nyelvi elemeket) gazdag és egyszer hőriő könyvtár (*Basis Library*) egészíti ki. A SML-nyelv és a Basis Library definícióját legutóbb 1996-ban vizsgálták fölül.

Az első ML-értelmezőt 1977-ben írták az edinburgh-i egyetemen. Az évek során számos értelmező és fordító program került el, egy részük öncenkötőes, más részük szabadon használható. Az utolsók körtől kettőt ajánljunk az olvasó figyelmébe (mindekeitől már az 1997-es, módosított definíciót követő):

- Az *mosml* (*Moscow SML*) a *Core Language*-t teljes egészében, a *Module Language*-t pedig csak részben és korlátozásokkal valósítja meg. *Basis Library*-je folyamatosan bővül, a már elérhető modulok kielégítik az 1997-es definíciót. Többek között linux, unix, Win95, Win98 és WinNT alatt elérhető, kis *erőforrásigényű* programrendszer.
- Az *smbyg* (*SML of New Jersey*) a teljes SML-nyelvet, azaz a szabványos *Core Language* mellett az igyancsak szabványos *Module Language*-et, valamint az 1997-es leírásoknak megfelelő *Basis Library*-t valósítja meg. Többek között linux, unix, Win95, Win98 és WinNT alatt elérhető, kis *erőforrásigényű* programrendszer.

Az SML-nyelv most ismertedők igényeinél a kevésbé erőforrásigényes *mosml* is mindenben megfelel.

Mindkét SML-értelemzőnek van egy nagy hátránya: a kezelő felületük irónépszerű, alig van mód az elülsők javítására, és nincs lehetőség a korábban leírt sorok elbírálására. Ezén az *emacs* szövegszerkesztő SML-programozást támogató környezet segít, nevezetesen az SML-üzemnövőn. (A Prolog-értelemzők kényelmes használatahoz az *emacs*-ra, mégpedig az *emacs* Prolog-üzemmodjára van szükség.)

A funkcionális nyelvek általában interaktívak, megrajtolásukra értehetségi programot (interpret) írnak. Az SML-értelemzők és a programozók többek között a *készleltetjel*, a *fordítójel*, a *kérdezőjel* és a *válaszjel* révén társalgunk egytársal. Az alábbi táblázatban a bal oldali oszlopban az *mosml*, ill. az *smbyg* által használt jeleket adjuk meg:

- a sor elején álló *készleltetjel* hatására megkezdődik a kiértékelés,
- ; a bevitelt záró *kérdezőjel* hatására megkezdődik a kiértékelés,

- = a sor elején álló *folytatójel*: az SML a megkezdett kifejezés folytatására vagy lezárására (a kiértékelőjére) vár (az *smtnj*-ben; az *mosml*-ben a folytatósor nem jelzi kilöni jel),
- > a sor elején álló *válaszjel*: az SML válaszát jelöli (az *mosml*-ben; az *smtnj*-ben a válaszsort nem jelzi kilöni jel).

Az *mosml* és az *smtnj* válaszai és fölleg hibáüzeneti különbségek egnymástól. Ebben a jegyzetben rendszerint az *mosml* 1.44 verziójának válaszát és hibáüzeneteit adjuk meg, és utalunk rá, ha ettől valamilyen ok miatt eltérünk.

Az SML-ből kilépni a készüléti jellel adott többféle válasszal lehet:

- a *quit()* függvény hívással,
- az *exit arg* függvény hívással, ahol *arg* helyébe a kilépési kódot jelentő egész számot kell írni,
- MS-DOS alatti a *ctrl-z*, majd az *enter* leítéssel,
- unix (linux) alatti a *ctrl-d* leítéssel.

Az SML-értelmező kalkulátorként is használható, pl.

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

Math.sqrt a Math könyvtábeli srt függényt jelöli. Egyes SML-könyvtárak tartalmát az A függeléken ismerhetjük.

1.10. Információforrások

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

Az angol nyelvű könyvek közül kilönösen a következőket javasoljuk:

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

Az on-line információforrások közül javasoljuk a következőket:

1. COMP LANG ML Frequently Asked Questions and Answers. <http://www.cis.ohio-state.edu/hypertext/faq/usenet/meta-lang-faq.html>
 2. Andrew Cunningham: *A Gentle Introduction to ML*. <http://www.dcs.napier.ac.uk/course-notes/sml/manual.html>
 3. Stephen Gilmore: *Programming in Standard ML '97: An On-line Tutorial*. <http://www.dcs.ed.ac.uk/fcsreps/EXPORT/97/ECSS-LFCS-97-364/>
- Az *mosml* és az *smtnj* megvalósítások honlapjának címe:
1. Moscow ML: <http://www.dina.kvl.dk/~sestoft/mosml.html>
 2. Standard ML of New Jersey: <http://cm.bell-labs.com/cm/cs/what/smlnj>

1.11. Köszönnetnyilvánítás

Köszönet illeti

- az *ML for the Working Programmer* c. könyv szerzőjét, Lawrence C. Paulson: a könyvből sok érdekeset tanultam az SML-ről, többek között a jegyzetben bemutatott példák és megoldások jó része is ehhez a könyvből származik,
- a *Moscow ML értelmező/forrás program* szerzőit, Peter Sestoftot és Szterej Romanenyt az oktatási céira (is) kitűnő SML-megvalósításért.

1.12. Hiba-jelentés

A szerző köszönettel fogad a hanakinf.bme.hu címre érkező bármilyen (sajtóhíbakra, tartalomra vonatkozó) észrevételt a jegyzettel kapcsolatban.

A nevekben a kis- és nagybetűk, a decimalis számjegyek, az aláhúzás-jel ($_$) és a perjel (‘, más néven felírásos, apozitív) használhatók. Jegyezzük meg, hogy az SML különbözetet tesz a kis- és nagybetűk között!

2. fejezet

2.1.2. Névadás függvénynek

Legyen²

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

akkor

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

vagy függvényként

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

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

Az SML-ben a függvény maga is: érték!³ 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 meglejtő, de az fn r => pi * r * r maga is kanonikus kifejezés, hiszen tovább nem esy- szerűsíthető⁴.

Egy függvény argumentumának típusa – mint halmaz – tartalmazza az értelmezéstől tartományát (domain), eredménynek típusa – mint halmaz – pedig tartalmazza az értékészletet (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 minden állítja elő a függvény (pl. az egész típusú eredményt adó sqrt, amely csak nemnegatív eredményt állíthat el).

A függvényt leképezésnek, transzformációnak (angolul mappingnek) is nevezik. A függvény típusa adja meg, hogy milyen típusú értéket minden típusú értékkel képezi le. Pl. az area függvény típusa: real -> real. Végül ezne, hogy a függvény eredménynek típusa nemazonos a függvény típusával!

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

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

Az állandókat tekintethetjük függvényeknek is, mégpedig argumentum nélküli függvényeknek. A jól ismert unáris (egoperandús, monadikus) és bináris (kétp operandusú, diadikus) operatorok

²A pi állandó a fáth könyvtárban is megvan.

³Az fn jelölésről részletesen egy későbbi fejezetben szólunk. Az fn jelet sokszor *lambda* eljárásként, ami az eredetéről a λ-kalkulusról utal. A lambdafeljeljelésben a fonti függvénnyel: $\lambda x \cdot \pi : r$. A késargumentumú szozásfüggvény definíciója a λ-kalkulusban: $\lambda x \bullet \lambda y \bullet x \cdot y$. SML-jelbissele: fn x => fn y => xy.

⁴A típus és a halmaz rekonkéntű fogalmak: a típus határozza meg azoknak az értékeknél a halmazat, amelyeket az adott típusba tartozó azonosítók, nevezékelhetnek.

Nevek, függvények, egyszerű típusok

2.1. Értékdeklaráció

Deklaráció: valamilyen értéknak (pl. egésznek, valósnak, karakternek, fizernek, függvénynek), típusnak, szignatírúnak, struktúrának, funkcionak 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 jellemzése.)

2.1.1. Névadás állandónak

Egy állandó lehet

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

SML-példák (az SML-értelmező választ 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 tövább nem egyszerűíthető, ún. kanonikus kifejezések. Az SML-értelmező válasza minden esetben kanonikus kifejezést, az it, amely minden a legfelső szintű kifejezés értékét veszi fel. A fenti kifejezessorozat kiértékelése után pl. az it értéke 86400.

```
> val it = 86400 : int
- it;
- it div 24;
> val it = 3600 : int
it értékét elhalkítaiuk későbbre, pl.
- val secsInHour = it;
> val secsInHour = 3600 : int
```

¹Az SML-ben rendszerint normáltesztű olyan éles különbözetet tesz a C-ben.

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

2.1.3. Nevezékek újratelvezetése

Tetszőleges értéknak adhatunk *nevet* az SML-ben. A név egy érték, esetleg egy másik név *azonosítójá*. Név, szinonimána helyett egykori *azonosítójáról* ritkábban matematikai értelemben vett.) *változord* beszélünk. Az utóbbi elnevezés egyes programozók számára felhevesséző lehet, ugyanis az SML-beli „változók” másnéppen viselkednek, mint az imperatív nyelvkből jóI ismert társak: *nem frissíthető változók*, azaz nem kapnaknak új értéket a megszokott értékükkel. Nem frissíthető változók esetén *érték-szemantikával*, frissíthető változók esetén *hivatkozás-szemantikával* beszélünk.

Ha az SML-ben egy *azonosító státikus* (és *nem dinamikus*) kötést hoz létre. Az alábbi példában hába definíljuk újra *pi*-t, az *area* függvény definiciójába a pi *körtheti értéké* (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 függvény tijáradéfinílunk, az egész programot újra le kell fordítanunk, különben a változtatás *halászatban maradhat*.

2.1.3.1. Nevezékek képzése

A nevezékek (azonosítók) tetszőleges hosszúságúak lehetnek. Az SML-ben *alfanumerikus* (azaz kis- és nagybetűkből, számjegyekből, aláírásjelből, valamint percből), valamint *percfelbontásban* (azaz egyéb jelekkel álló, angolul *symbolic*) neveket különböztetünk meg.

Az írásjelekből képzett neveken 20-re jel fordulhat elöl:

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

Egyes jelisorozatoknak különleges jelentésük van, ezeket *femlartolt azonosítóknak* vagy *sintaktikai jeleknek* nevezünk. Példák:⁵

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

Lassunk egy példát írásjelekből képzett nevek deklarárárára!

```
> 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 ázert van igy, mert az SML tervéből az automatikus *tipuslezerzés* (*type inference*) megrálosítását fontosabbnak tartották a vele történő többszörös terhelésnél (*overloading*). A nevek többszörös terhelésének csökkenet, a jelentősége a moduláris programozás elterjedésével, hiszen a modulinek (az SML-ben a struktúra- és a funkcionamelek) *szektorában*, a nevek elhagytaként használhatók. Erről bővebben a moduláris programozásról szóló fejezetekben lesz majd szó.

⁵ int típusú, list típusoperátor, real pedig egyséjeg függvény tipusú is. Jegyezzük meg, hogy ugyanaz a név *egyidejűleg* jelölihet értéket, tipust, modult (strukturát, ill. funkciót), valamint rekordmezőt.

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

Az SML-ben a más programozási nyelvben megszokott egyszerű típusok neve: int, real, char, string és bool. Vannak további egyszerű típusok is, pl. word, word8, order, unit és substring. A gyakran használt függvényeket a vannak építve a nyelvekhez *belőfűggvényeknek* (built-in functions) nevezik. További gyakran használt függvények definícióját elindításakor olvassa be az SML-értelmező: ezek az *előre definált függvények* (predefined functions) a *kezdeti környezet* (initial environment) részét képezik. Sok hasznos függvény található az *Alapkönyvtárban* (SML Basic Library).

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

2.2.1. Egészek és valósok

A négy numerikus típus az int, a word, a word8 az előjeles, word és word8 az előjeles nélküli egész, real pedig valós (tulajdonképpen racionalis) számok típusa. A numerikus típusok gépi ábrázolása függ a megrálosítási ól. A legnagyobb int típus szám neve Int.maxInt, a legkisebb Int.minInt. A word típusú értékek bitszáma Word.wordSize adja meg, a word8 típusnak minden megrálosításban 8-bitesek. E négy típus közös, csak hivatkozásra használt meghatározott típusa az alábbi táblázat mutatja:

megnevezés	hivatkozott egyszerű típusok
realInt	int, real
wordInt	int, word, word8
num	int, real, word, word8

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

A belső real függvény egész (int) értéket alakít át valossá (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 -∞, ceil pedig a +∞ felé kerékít.⁷ E harom függvény típusa:

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

A ket utóbbi függvény szemantikáját pontosabban az alábbi, tulajdonságot *definiáló egysülytlen-* séggel 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:

⁶ Emelékezzünk vissza: ugyanaz a név többszöröle dolgozhat! A real név itt egyszerű tipust, másrészt egy függvényt ismerünk.
⁷ A floor és ceil valamint az alábbi ismertetett round és trunc függvényeket a General és a Real könyvtár definíálja. A General könyvtárban definít real függvény azonos a Real könyvtárban definít front függvényel. Az ugyancsak alább ismertetett abs függvény egészre korlátozott az Int, valósakra alkalmazható valtozatot a Real könyvtár definíálja.

Az `fn` szintaktikai jelzéje, hogy `real függvényéről` jelöl, az `int` -> `real típusú fejezés` pedig a függvény típusát adja meg.

Egy szám abszolút értékét a belső `abs` függvény állítja elő. Az `abs` azonosító többszörös terjesű, minden `int`, minden `real` típusú értéket alkalmazható.⁸

`abs : real → real`

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

> val `it = fn : int → int`

trunc : `real → int`

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

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

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

`round : real → int`

`trunc : real → int`

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

`round r = floor(r + 0.5)`

`abs(x) = trunc r) <= abs r`

2.2.1.2. Alapműveletek előjeles egész számokkal

Az előjeles egész számokra alkalmazható alapműveleteket a következő táblázat foglalja össze. Szemantikájukat *kiszámítási szabályként* is használható *egyenlettel* írjuk le; round a 'legközelebbi' egész szám, trunc pedig a 0 felé kerül:

`round r = floor(r + 0.5)`

`abs(x) = trunc r) <= abs r`

2.2.1.3. Alapműveletek valós számokkal

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

`0.01`

`~1.2E12`

`7E-5`

Az E után írás számtrendszerben kell megadni a kitévő pozitív vagy negatív egészéket. A valós számokra alkalmazható alapműveleteket a következő táblázat foglalja össze.

jele	jelentése	jellege	pozíciója
<code>-</code>	negatív előjelek monadicus	monadicus	prefix
<code>+</code>	összeadás	diadicus	prefix
<code>-</code>	kivonás	diadicus	infix
<code>*</code>	szorzás	diadicus	infix
<code>/</code>	osztás	diadicus	infix

A kétoperandusú (diadicus, bináris) valós műveletek típusa: `real * real → real`, az egyoperandumú (monadicus, unáris) valós műveletek: `real → real`. A `~, ~` és `* művelei jelen többszörösen terhelyi irányelvűk` keretet nyújtanak.

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

név	jelentés	tipus	megjegyzés
<code>pi</code>	a π állandó	real	1
<code>e</code>	az e állandó	real	2
<code>sqr</code>	\sqrt{x} négyzetgyöké	real → real	
<code>sin</code>	$\sin x = x$ szinusza	real → real	x radikálban
<code>cos</code>	$\cos x = x$ koszinusz	real → real	x radikálban
<code>tan</code>	$\tan x = x$ tangensz	real → real	x radikálban
<code>exp</code>	$\exp x = e$ az x-edikén	real → real	
<code>pow</code>	$\text{pow}(x, y) = x$ az y-edikön	real * real → real	3
<code>ln</code>	$\ln x = x$ természetes alapú logaritmusza	real → real	
<code>log10</code>	$\log_{10} x = x$ 10-es alapú logaritmusza	real → real	x > 0.0

Megjegyzések a táblázathoz:

1. Az egységesített áltérkörrel kerülhet:

2. A természetes alapú logaritmus áltára.

3. `pow` alkalmazásának összetett a feltétele: $y \geq 0.0 \wedge (\text{integral } y \vee x \geq 0.0) \vee y < 0.0 \wedge (y \wedge x \neq 0.0) \vee x > 0.0$, ahol integral x SML-beli megalosítása pl. a `real(round x)` feltétel lehet, integral x olyan valós számra teljesül, amelynek „nincs” örtízese.

Bármiel függvényalkalmazás (más szóval az összes monadicus műveleti jel, azaz az összes prefx operátor) erősebb köt a diadicus műveleti jeleknel (más szóval az infix operátoroknál). Ezért pl. `exp a * b = (exp a) + b ≠ exp(a + b)`.

2.2.1.4. Precedencia

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

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

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

Megjegyzések a táblázathoz:

1. quot és rem prefér valóozatát az Int könyvtár definíja, a többi operátor infix pozíójú és a kezelői környezet része.
2. quot és rem használához be kell tölteni az Int könyvtárat: load "Int". Betöltés után vagy a *haszi_nevükkel* hivatkozhatunk rájuk (Int.quot, Int.rem), vagy láthatóvá tehetjük az Int könyvtárban definált függvényt (open Int), vagy pedig értékkékkarcióval új nevet adhatunk a két függvénynek (val quot = Int.quot; val rem = Int.rem). Ha a két függvényt infix pozícióban alkaijuk használáni, alkalmaznának kell rájuk az infix deklarációt (ajánlott precedenciaszintjük a 7-es): infix 7 quot rem.
3. A numtxt megnöveztés a num-csoportba tartozó típusokon kívül még a char és a string típusú jelentő (reszletek a fejezet hátralévő részében).
4. bool típusú érték az igazságáról (reszletek a fejezet hátralévő részében).
5. Az - a lín. típusváltozó, és olyan férőképek típusát jelöli, amelyekre az *egyenlőségiságú* elvezethető a részleteket egy későbbi fejezet ismerteti). Jegyezzük meg, hogy valós számok egyenlőségét nem lehet tesztelni az SML-ben, és más nyelv használata esetén sem célszerű, mert az ábrázolás pontatlansága, a kerekítési hibák miatt még azonosnak vélt értékekre is hamis eredményt adhat a vizsgálat. Két valós szám egyenlősége helyett azt kell megvizsgálni, hogy a különbségek kisebb-e egy elengedően kicsi valós számnál. (Az *egyenlőségiságú* kérésében nem egységesek az SML-megvalósítások: pl. az mostani megegyező, az smby nem engedi meg, hogy az = és a <> operátor valós számokra alkalmaznazzuk.)
6. Vannak más típusokra alkalmazható infix operátorok is. Ezek precedenciájáról később lesz szó.

2.2.1.5. Alapműveletek előjel nélküli egészekkel

Az előjel nélküli egészek típusa az SML-ben: word, ill. word8. word bitszáma függ a meghatalmítástól, word minden meghatalmításban 8 bites. Jelöléskre lassunk néhány példát!

```
Or241
Orxf1
OrxF1
```

Ow (a nulla után kis w áll!) jelöli, hogy word vagy word8 típusú értéktől van szó. A 0w után decimalis vagy hexadecimális egésznek kell jönnie. A hexadecimális számot kis x betűvel kell kezdeni, jegyei 0 és 9 közötti számjegyek, továbbá A és F (vagy a és f) közötti betűk lehetnek. Alapértelmezés szerint az így megadott állandók word típusuktól word8 típusú értékek elbállításához típusmegközelést kell alkalmaznunk (l. a 2.2.1.6 szakasz). Vegyük észre, hogy nemcsak nevek, hanem állandók is lehetnek többszörösen terhelve: az SML-ben ílyenek a word és word8 típusú állandók, a C-ben és a Pascalban ílyen az egész értékű int vagy real típusú számok jelölése.

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

2.2.1.6. Tipusmegkötés

Az SMI a legtöbb kifejezés típusát a tudja vezetni a kifejezésben előforduló értékek (nevek, állandók) típusából. A *tipuslezetés* (type inference) csak néhány többszörösen terheltető névű belső függvény esetében nem lehetséges. Ilyenkor, ha az alapértelmezéstől el akartunk terni, típusmegközelést kell használni. (Mint tudjuk, alapértelmezés szerint a többszörösen terheltető névek int típusú értéket vannak.) Pl.

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

Egy kifejezés vagy részkifejezés típusát így köthetjük meg, hogy utána kettősponttal elválasztva megadjuk a típusnevét: *kifejezés: típusnév*. Példák:

```
- val mask = 0x7F : 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;
```

Zárójeléssel szabhatjuk meg, hogy a típusnevek minél vonalakkozzon, a típusoperátor precedenciája kisebb, mint a függvényrallamazásé vagy az aritmetikai műveleteké.

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

2.2.2. Füzérek

A string típusú füzér a szokásos módon, idezőjelek között áll, esetleg egyetlen karaktert sem tartalmazó karaktercsorozattal jelölik az SML-ben, például

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

¹⁰Az *mostak*-ban a Word8-word típusú csak aklór használható, ha a Word8 könyvtár be van töltve. Word8 betöltése nélkül is használható az *mostak*-ban a word8 típusú, használata azonban nem javasolt, mert az ilyen program csak módszirák után futtatható az *smby* alatt.

2.2.2.1. Escape-szakaszok

Különleges karakterek beírására (a C nyelvből is jól ismert) *escape-szakaszokat* használhatók, ezeket az alábbi táblázatban soroljuk fel. Egyes döjök megérettéselelhetők, hogyan jeöli a karaktereket az SML, és mit csinál az ord függvény. Ezekről a kérdésköről a 2.2.3 szakaszban lesz szó.

jelölés	escape-szakasz meghozzávalója	decimális ASCII-kódja	megjegyzés
\a	csempő (alert)	7	
\b	visszapéns (backspace)	8	
\t	vízszintes (horizontális) tabulátor	9	
\n	újsort-jel (newline)	10	
\v	függeléges (vertikális) tabulátor	11	
\f	lapdobás (form feed)	12	
\r	sorejtéje-jel (return)	13	
\^c	vezérölök karakter	ord c - 64	1
\ddd	letezőleges karakter	ddd	2
\"	idezőjel (double quote)	34	
\\\	hátrátöröl-vonal (backslash)	92	
\w...w	figyelmen kívül hagyandó	3	

Megjegyzések a táblához:

- A c helyéhe olyan karakter jelle írátható, amelyre "#\"<= c <="#" -".
- dd háromjegyű decimális számot jelöl. Minthaaron számjegyet ki kell römi, a vezető nullákat is.

- Az obján sorozatot, amelyben w helyén egy vagy több szóköz jellegű formázó karakter (szóköz, tabulátor, lapdobás, újsort stb.) áll, az SML nem veszi figyelembe.¹¹

2.2.2.2. Gyakori műveletek fizérekkel

Két fizért kapcsol egybe a string * string -> string típusú ~ (felfelé mutató nyíl, kalap, circumflex) infix operátor, fizér hosszát aifja eredményül a string -> int típusú size függvény: "alma" ~ "fa";

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

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

¹¹ Ha egyetlen sorba akarunk kiratni egy olyan fizért, amely a begépcelskor több sorban fer csak el, *folytatásor* (continuation line) használunk: a formázó karakterek elő és mögött egyszerű hatrátör-vonaltat (\) rakunk, pl. \abra;

> "abracadabra" : string
> "abracadabra" : string
A folytatásor nyitó és záró hatrátör-vonaltak között formázó karakter nem adható meg escape-szakencsík következő példában az SML a \t-ból a \t betű szentíne, a \a párt pedig csengőjének vészí:
- "abracadab\ta";

2.2.3. Karakterek

Az SML-ben a karaktertípus a char típusúval jelölik.

Az SML-ben a karakterjelöles elég hosszadalmas: egyelen karakterből álló füzér állandó, amely előtt # áll. Ez azért van így, mert eddig nem volt karakterjelöles az SML-ben, karakter helyett az egyetlen karakterből álló füzert használták. Példák:

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

2.2.3.1. Gyakori műveletek karakterekkel

Karakter ASCII-kódját állítja elő az ord, adott ASCII-kódú karaktert ad vissza a chr függvény (ord típusa char -> int, chr típusa int -> char), mindenbeli belső függvény. ord 0 és 255 közötti értéket állít elő, chr 0 és 255 közötti értékekre alkalmazható, más argumentumra az SML-értelemező libbát jellez.¹² Példák:

```
- fun digit i = chr(i + ord #"0");
  > val digit = fn : int -> char
char típusú értékek összehasonlítására használhatók a szokásos relációs operátorok (<, <=, >, >=).
```

2.2.4. Igazságértékek, logikai kifejezések, feltétel es kifejezések

Az SML igazságérték típusa (bool) ugyanaz, mint pl. a Pascal boolean típusa. Csupán kétfele bool típusú állandó van, jelölések: true és false.

Igazságérteket általán eredményül a relációs operátorok:

- előjeles és előjelek nélküli egészekre, valósakra, karakterekre, fizérekre: <=, <, >, >=,
- az ún. egyenlőségi típusokra (equality types): =, <>.¹³

A bool típusú értéket eredményül adó kifejezést gyakran nevezik predikáturnak.

2.2.4.1. Feltétel es operátor

A feltétel kifejezés az SML-ben

```
if E then E1 else E2
alakú, ahol
```

- az E logikai kifejezés bool típusú értéket ad eredményül,
- az ord függvény esetén az értékehez tartozó csak részhalmaza az argumentum típusának, az ord függvény esetén pedig az értékeket szintén csak részhalmaza az eredmény típusákat. Függvénynek az olyan leképzést nevezik, amely az értékelmezesztés minden elemének az értékkészlet pontossága (gyánmástel) nem feltételei különböznek. Ha ez nem teljesül, azaz ha az értékelmezesztés tartomány egy-egy elemének több elem is megfelelhető az értékkészletben, akkor a leképzést relációtartalomnak hívjuk.
- Mint említettük, a valós számok közötti egyenlőség és egyenlőtlenségekkel könyen lihás eredményt vezet, ezért az smlnk a real tipust nem tekinti egyenlőségi típusnak.

- az E1 és E2 kifejezések tétszöges, de egyforma típusú értéket adnak eredményül, és

- az else ág sohasem maradhat el.

Félda:

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

A feltételezés kifejezés **if-then-else** operátora, mint lájuk, *három* operaudust. Az operandusokat az SML-értelmező *hútán* értékeli ki. Ez annyi jelent, hogy az E1, ill. az E2 kifejezés kiértékelésére össak akkor kerül sor, ha az E kiértékelésénél true, ill. false az eredménye.

2.2.4.2. Logikai operátorok

Az SML-ben logikai kifejezésekben három logikai operátor alkalmazhatunk, ezek a kető operandustú andalso és orelse, valamint az egyoperandusú not, andalso és orelse *listák* kiértékelésű: ha a bal operandus kiértékelése elég az eredménynek meghatározásához, az SML-értelmező a jobb operandust egyszerűen nem értékeli ki.¹⁴¹

A lista kiértékelésű operátorok hasznosak pl. tömbök indexelhetővé vágy listák végénél a kezelésére (hogy az utolsó utáni elem feldolgozására már ne kerüljön sor), a 0-val osztás elkerülésére (amikor egy változó értéke 0 is lehet) stb.

Jegyezzük meg, hogy orelse precedenciája kisebb andalso precedenciájánál, és mindenkitől kisebb bármely más infix operátor precedenciájánál. Ezzel szemben a not precedenciája a lehető legmagyarobb, a többi prefix helyzetű egycopandusú operátorhoz (más szóval: függvényjelzé) hasonlónan.

andalso és orelse felhasználásával definíthatjuk például a logikai *környezeteket* és *alternaciót* (diszjunktót) megrajstó függvényeket **&&**, ill. ||| néven:

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

A lényeges különbség andalso és **&&**, ill. orelse és ||| között nem az, hogy andalso és orelse *infix*, **&&** és ||| pedig *prefix* helyzetben használhatók, hanem az, hogy az előbbiek *listák*, az utóbbiak pedig *mohó* kiértékeléstek! Ha látna lenne előtérben a teljes kifejezés eredménye **&&**, ill. ||| első argumentumára alapján, az SML-értelmező kiértékelését előtt a második argumentumukat is kiértekelni (tervábel részletekben).

Ternérseiseken e két függvényt *infix operatorként* használhatjuk, ha *infix* voltukat deklarálunk, például így (l. még a 4.1 szakasz):

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

Ha E, E1 és E2 egyaránt bool típusú kifejezések, if E then E1 else E2 helyett írhatjuk, hogy E andalso E1 orelse not E andalso E2¹⁴², mert andalso és orelse is lista kiértékelésűk, de helyettük nem használhatnánk a most definált **&&** és ||| operátorokat.

¹⁴¹Más nyelvek a lista kiértékelésű andalso és orelse operátorot *shortcircuited* operátornak nevezik, és például *and* és *or* néven emlegetik. A C-ben e tét operátor jelenleg **&&** és |||. ¹⁴²Az utóbbi, kikrakva a zárójelkéket, így értendő: (E andalso E1) orelse ((not E) andalso E2).

2.2.4.3. Tesztelő függvények

Predikátumot használunk annak előírásére, hogy bizonyos értékek adott feltételeit kielegítik-e. Az ilyen cétra használt predikátumot gyakran tesztelő függvénynek nevezzük. A Char könyvtárban például sok olyan jól használható függvény van, amelyek karakterek osztályozását teszik lehetővé. Ilyen függvényeket persze saját magunk is definíthetünk. Nezzük 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 könyvról leghasznosabb tesztelő függvényeit az alábbi táblázatban soroljuk föl. Ezek a függvények mind char -> bool típusúak.

A függvények minden paramétere a contains : string -> char -> bool függvény segítségével adhuk meg,¹⁴³ contains s c akkor igaz, ha a c karakter benne van az s füzetében. (A contains függvényt ugyancsak a Char könyvár definíálja.)

függvényneve	arg.	jelentés
isLower	c	contains "abcdefghijklmnopqrstuvwxyz" c
isUpper	c	contains "ABCDEFGHIJKLMNPQRSTUVWXYZ" 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
isString	c	c látható karakter vagy szóköz (# ")
isspace	c	contains "\tr\n\r\f" c
isPunct	c	isPrint c andalso not(isSpace c orelse isAlphaNum c)
isStraph	c	not(isSpace c) andalso isPrint c
isASCII	c	0 <= ord c <= 127
isCtrl	c	not(is Print c)

¹⁴³A string -> char -> bool típuskifejezés jelentését a részlegesen alkalmazható függvényekről szóló fejezetben magyarázzuk meg.

```

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

```

Típusdeklarációval új nevet adtunk egy típusnak:

```

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

```

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

3.1. Pár, ennes

A *(firstname, lastname)* egy pár, a *(day, month, year)* egy hármas stb. Az *ennes* (angolul: *tuple*) elemet vissz vételez a vezetéknél, az elnevezések tipusa térszögeles, sorrendjük fontos.¹ Egy ennes elnevi különböző típusukt lehetnek. Példák:

```

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

```

Mivel az elemek sorrendje fontos, a fenti két bár különböző típusú. A *rekord* olyan ennes, amelyben az egyes elemek megegyenek: a sorrend többé nem fontos, az elemekre nem a helyük szerint, hanem a címkejükkel hivatkozunk. A rekordról a 3.2 szakaszban lesz szó.

Típuskifejezés. *string* * *int* * *string* speciális kifejezések, ún. *tipuskifejezések*, ún. *tipuskifejezések*, élenei *tipusállandók* (*int*, *real*, *string* stb.), *tipusoperátorok* (*, -> stb.), *tipusoperátorok* (*, *a*, *b* stb.), *tipusoperátorok* (*, -> stb.) és más típuskifejezések lehetnek. A típusoperátoroknak is van precedenciájuk: a *kereszszorozás* (Descartes-szorzat, jele a *) precedenciája nagyobb a *leképzés* (jele a ->) precedenciájánál. Típuskifejezésben is használható *zárójel* a műveletek sorrendjének megállapítására.²

Látni fogjuk, hogy az eddig megszintenzített kívül vannak más típusállandók és típusoperátorok is, sőt a programozó maga is definíálhat típusállandókat, típusváltókat és típusoperátorokat.

3.1.1. Példa: vektorok

A bemutatott példákból a vektor *real* * *real* típusú.

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

```

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

```

¹Később, a 3.1.4 szakaszban látni fogjuk, hogy kitintett szerepe van az egyetlen elemet sem tartalmazó ennesnek, a 0 jelűt *nullasnak*. Az egyszerűen ennek a 0 ismert zárolójeles kifejezés, ha nem okoz felürcsök, a zárójel elhagyható. Kétféle ennies a *pair*, háronelemek a *hármas* stb.

²A típuskifejezés fogalma kevésbé üjű, mint elő hallásra gondolnánk: pl. a *TYPE numm = ARRAY [...] OF ...* Pascal-deklaráció jobb oldala típuskifejezés, bár a Pascalban ritkán használják ezt a fogalmat.

3.1.2. Függvény több argumentummal és eredménnyel

Nézzük a következő definíciót:

```

- fun average (x, y) = (x + y) / 2.0;
- fun addvec ((x1, y1), (x2, y2)) : vec = (x1 + x2, y1 + y2);

```

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

Az SML szemléltetőmódra szerint minden függvénynek *egyedben* argumentuma és *egyedben* eredménye van, egy-egy *enness*. Ez azért jó, mert egyszerű.

```

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

```

Az *mosml* válasza:

```

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

```

A válaszban a *vec* típusnev helyett a vele egyenértékű *real* * *real* típusról írunk! Az *mosml* tervezői ezzel is tüdőtostani akarják, hogy a *vec* név csak *szinonimá*, nem új típus. Az *mosml* válaszra kicsit más:

```

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

```

Mivel a programozó addvrec eredményét vec típusnak deklaráálta, válaszából az *smml* 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 közé a *vec* típusnévezéhez.

3.1.2.1. Gyakorló feladatok

1. Hány argumentuma van az *addvec* függvénynek? 1, 2 vagy éppen 4?

2. Definiáljon *subvec* néven olyan függvényt, amely két vektor, v1 és v2 különbségét állítja elő!

A függvény típusa: *subvec* : *vec* * *vec* -> *vec*.

3. Definiáljon *scalevec* néven olyan függvényt, amely az *r* valós számúval megszorozza a vektort! A függvény típusa: *scalevec* : *real* * *vec* -> *vec*, deklaratív specifikációja: *scalevec(r, v) = az r skalár és a v vektor szorzata.*

³Új adattípus férhető el a vezetéken, azaz ennek *absztrakteinek* is van az SML-ben. Az ennek a sokféléképen használható *datatype* deklaráció, amelyet az XML modulcsatolóhoz a *structure* és *signature* rejtve valósítottunk meg eads absztraktió. A másik lehetőség a már ideféműlnak tekintetű *abstype* deklaráció. Mindenről később lesz szó a jogyzetben.

3.1.3. Ennes elemeinek kiválasztása mintaillesztéssel

Egy ennes elemeit elegánsan *mintaillesztéssel* azonosítjuk. Példa:

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

Az `(xc, yc)` párr *illeszkedik* `scalevec` eredményére: `xc` a pár bal, `yc` pedig a jobb oldali tagjára. Az értékekkláracíban alkalmazott miatta éppolyan összetett lehet, mint az argumentum-mintha a függvénydeklarációból.

3.1.4. A nullas

A *nullas* (angolul *0-tuple*) olyan ennes, amelynek egyetlen eleme sincs. Az SML-ben a nullast a C jelleggel jelölik. Az angol szakirodalom néha *hernal-nek*, nemet ének nevezi, ugyanis ez az egyetlen eleme a unit az ALGOL68 és a C nyelv szóhasználata szerint: *void* típusnak.

A unit típus a típusnivellek *egységeleme*. Olyan esetben használjuk, amikor a függvénynek nincs argumentuma, vagy amikor függvényt a *meillehetősá* miatt alkalmazzunk, mert nincs fellásználásra eredménye.

3.1.4.1. A print, a use és a load üggyény

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

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

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

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

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

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

A load-dal a már lefordított *tárgyprogramokat* töltethetjük be az interaktív *mosm*-környezetben (az *smlnj* másiképpen kezeli, ezért ott csakkeppen kell betölteni a modulokat). Pl. `Load "Math"` a Math.wo nevű könyvtári modult, `load "x"` az x.sml program lefordított változatát, x.wo-t tölti be. Jegyezzük meg, hogy Load a .wo névkitörjesztést tetelez fel, akár kintjük, akár nem.

3.1.4.2. Mi a különbség?

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

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

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

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

3.2. Rekord

A rekord olyan *felcímkézett* ennes, amelyben – a címek használata miatt – az elemek sorrendje közömbös. A rekord elemeit *kapsos zárolják* között kell felsorolni. Ugyanaz az eredményre például az alábbi két deklarációnak:

```
- val emp1 = {fname = "Jones", age = 25, salary = 15300} ;
  > val emp1 =
    {age = 25, name = "Jones", salary = 15300}
    : {age : int, name : string, salary : int}
- val emp1 = {name = "Jones", salary = 15300, age = 25};
  > val emp1 =
    {age = 25, name = "Jones", salary = 15300}
    : {age : int, name : string, salary : int}
```

Az SML-errelnevezők válaszokban a rekord elemire rendszírni a címeket *elhelyezzék* előttjük ki. A deklaráció után az elemekre a címkejükkel hivatkozhatunk a program szövegében (a címek természetesen lokálisak az adott rekordot dokumentáló program szövegében), például:

```
- #name emp1;
  > val it = "Jones" : string
- #age emp1;
  > val it = 25 : int
```

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

```
- val negyes = {f1="a", 2="b", 3="c", 4="d"};
  > val negyes = ("a", "b", "c", "d") : string * string * string * string
- val negyes = {f3="c", 4="d", 2="b", 1="a"};
  > val negyes = ("a", "b", "c", "d") : string * string * string * string
```

A két valorát egymértekű, amint az SML-értelemező válaszából látszik. Ugyanez a megsokott rövid alkalan, a címkek ellagytásával:

```
- val negyes = ("a", "b", "c", "d");
  > val negyes = ("a", "b", "c", "d") : string * string * string * string
- val negyes = {f3="c", 4="d", 2="b", 1="a"};
  > val negyes = ("a", "b", "c", "d") : string * string * string * string
```

Akárhogyan is definíáltuk a negyes-t, az elneire, ha szükséges, a címkekkel 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
- harom helyett, ahol csak lehet, inkább az elegáns mintaillesztést használjuk:
- val (a, b, c, d) = (#"a", #"b", 3, false);
  > val c = 3 : int
```

3.2.1. Rekordminta

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

Például:

```
- val {name = ename, salary = esalary, age = eage} = emp1;
  > 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 minősűrű helyett ...-ot írunk. A ...-ot tartalmazó *recordspecifikáció* részlegesnek (parciálisnak) nevezik, például:

```
- val {name = ename, salary = esalary, ...} = emp1;
  > val ename = "Jones" : string
  > val esalary = 15300 : int
```

Mintának magukat a mezőneveket is használhatjuk:

```
- val {name, age, salary} = emp1;
  > val name = "Jones" : string
  > val age = 15300 : int
  > val salary = 25 : int
```

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

```
- val {name, ...} = emp1;
  > val name = "Jones" : string
```

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

Következő példánk a *kizáro-nagy* művelet definíciója:

- infix 1 xor;
- fun p xor q = (p andalso not q) orelse (q andalso not p); vagy ha így jobban írunk:
- 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 történő megváltoztatása nincs.

- nonfix xor;

A nonfix deklarációt a fejlesztők saját maguk számára találták ki. Hasonlaltat nem javasoljuk a programozói gyakorlatban, mert például a nonfix + deklaráció után a + jel szintaxisa szokatlan lenne, és használata váratlan hibajelzéshez vezet. Az op kulcsszóval, amit egy előző példában láttuk, *infix* helyzetű név *lokális érvényű* alakában írható át *prefix* helyzetűvé, ezért biztonságos a használata.

4.1.1. Infix operátor kötése

Tudjuk, hogy a nagyobb precedenciájú operátor erősebben köt. A kifejezés kiértekelése szempontjából az sem közönös, hogy az operátor balra vagy 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 hatványozás: $a^b = a^{(b)}$. A balra kötő operátorokat a már ismert infix, a jobbra kötőket az infixr különböző deklarációink.

Példák:

- infix 6 plus;
- fun a plus b = (" ~ a ~ "+" ~ b ~ ") ;
- infix 7 times;
- fun a times b = (" ~ a ~ "*" ~ b ~ ") ;
- infix 8 pr;
- 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" por "j" por "2" times "n";
- > val it = "(m*(i**j)*2)*n)" : string

Egy *infix* operátor az op kulcsszóval nemcsak függvénydeklarációban alkotható át átmennetileg *prefix* helyzetűvé, anint korábban látottuk, hanem tetszőleges kifejezésben. Például:

- op plus ("a", "b");
- > val it = "(a+b)" : string
- op +(1, 2);
- > val it = 3 : int
- 1. Az op kulcsszó alkalmazásával az esetleg *infix* helyzetűnek deklárált nevet *átnemetileg prefix* helyzetűre tesszük:
- fun op &&& (b, j) = b andalso j;
- infix 2 &&&;
- fun op ||| (b, j) = b orelse j;
- infix 1 |||;
- 2. A deklarációk és a definíciók sorrendjét megváltoztatjuk, és már a definícióban infix helyzetben használjuk a &&& és ||| nevet:
- infix 2 &&&;
- fun b &&& j = b andalso j;
- infix 1 |||;
- fun b ||| j = b orelse j;

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

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

```

op plus ("a", "b");
 ! Top-level input:
 ! opplus("a","b");
 ! ~~~~~
 ! Unbound value identifier: opplus

```

a következő kifejezés azonban *tiltás*, mert `opplus-t` tíj (alfanumerikus) névnek tekinti az SML-értelmező, és nincs definiáltan, tilbát jelez:

- `opplus ("a", "b");`
- `! Top-level input:`
- `! opplus("a","b");`
- `! ~~~~~`
- `! Unbound value identifier: opplus`

Az alábbi kifejezések jelentése nyilvánosak, hiszen a nevek vagy alfánumerikusak lehetnek, vagy írásjelekből állhatnak, és a '`'` nem használható írástelekből álló nevek képzésére (1. 2. 1.3.1):

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

**4.2. Kifejezések kiértékelése az SML-ben**

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

*Sztatikus körések* beszélünk, ha egy függvény (eljárás) forrásáskor az értelmező a formális paraméter minden előfordulását az argumentumum (az aktuális paraméter) értékével helyettesíti a függvény (eljárás) törzsében. (Ne felejtük, hogy az argumentum és a formális paraméter ennek, azaz összetett is lehet.) Kérdez, hogy az aktuális paraméterként által 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 kiérték előtt a behelyettesítés, azaz a függvény (eljárás) tényleges megnevezása *elő*.

Tiszán funkcionális nyelvekben szoktak a *lusta* (szükség szerinti, normál sorrendű, angolul *lazy*, call-by-need, normal-order) kiértékelést: egy függvény (eljárás) argumentumát csak akkor értékeli ki, ha *az amikor szükséges van rá* a behelyettesítés után.<sup>2</sup> Nézzünk két egyszerű függvényt!

## 4.2. Kifejezések kiértékelése az SML-ben

A kifejezések kiértékelést sorrendje, mint már említettük, alapvetően kétféle lehet: *mohó* és *lusta*.  
*Sztatikus körések* beszélünk, ha egy függvény (eljárás) forrásáskor az értelmező a formális paraméter minden előfordulását az argumentumum (az aktuális paraméter) értékével helyettesíti a függvény (eljárás) törzsében. (Ne felejtük, hogy az argumentum és a formális paraméter ennek, azaz összetett is lehet.) Kérdez, hogy az aktuális paraméterként által 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 kiérték előtt a behelyettesítés, azaz a függvény (eljárás) tényleges megnevezása *elő*.

Tiszán funkcionális nyelvekben szoktak a *lusta* (szükség szerinti, normál sorrendű, angolul *lazy*, call-by-need, normal-order) kiértékelést: egy függvény (eljárás) argumentumát csak akkor értékeli ki, ha *az amikor szükséges van rá* a behelyettesítés után.<sup>2</sup> Nézzünk két egyszerű függvényt!

```

(* sq x = x négyzet
 sq : int -> int
*)
fun sq x = x * x;
(* zero x = x*től függőenül minden 0
 zero : int -> int
*)

```

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

```

(* zero x = x*től függőenül minden 0
 zero : int -> int
*)

```

<sup>1</sup>Nem szatikus, hanem *dinamikus* az olyan kiérte, amely a formális paraméter minden előfordulását a függvény (eljárás) megfelelőként (végrehajtásakor) helyettesíti az argumentummal (az aktuális paramétereivel).

<sup>2</sup>Ha már ritkán találkozní az ALGOL-60 név szerevi (call-by-name) paraméterterjesztéshez szüntelen a behelyettesítés *után* kerül sor. A név szerevű paramétereket az argumentumként áradott kifejezést betűszavat adja át a meglévőt eljárásnak. Később elnagyarázunk a kilóbiszget a név szerevű paraméterekről és a *lusta* kiértékelésről.

A teljesleg kódolási eredményt a *hiestholzis szervizi* (call-by-reference) paraméterteradást, amelyet számos programozási nyelv alkalmaz, töbök között a Pascal és a C. Ez az áradási mod késgelelő hatékony, hiszen fórtárolói címet vesz át a hívó eljárás, aholon közvetlenül olvashat, ill. átvevő követői írhat. De éppen ez a hatáltsági elosztás miatt, hiszen pl. egy eljárás sikerteitessé esetén lehetővisszaláthatni a korábbi állapotot.

a zéró függvényt meghívjuk, *mohó* kiértékelés esetén az argumentumát *feltesz*eszen számítjuk ki, hiszen nem használjuk semmire.

### 4.2.1. Mohó kiértékelés

Az SML-ben egy kifejezés állandókból, változókból, függvény hivásokból és feltételek kifejezéséből állhat.

$f(E)$  értékének kiszámításához először az  $E$  kifejezés értékét határozzuk meg, majd  $f$  törzseben ezzel az argumentummal helyettesítjük a formális paraméter minden előfordulását. A *minimális szisztema* nem okoz gondot: az  $E$  kifejezés értékét most is ki kell számítani, majd az argumentummal szemben fel kell hoztanunk, és az egyes összetevői kell behelyettesíteni a függvény törzseben a megfelelő helyre. Legyen pl. fun  $f (x, y, z) = törzs$ , ekkor az  $E$ -t fel kell bontani az  $(x, y, z)$  összetevőkre, majd a törzsenben  $x$ -et,  $y$ -t és  $z$ -t kell helyettesíteni a megfelelő értékkékkel.

Példaképpen nézzük az `sq (sq (sq (2)))` kifejezés kiértékelését, más szóval egy szisztemát, rendkívül! (Az egyszerűsítés eredményének nyilvánvalóan olyan kifejezésnek kell lennie, amelyik tovább nán nem egyszerűsíthető, ún. *kanonikus* kifejezés.) A három függvényhivásból csak a legbelosztó hivásnak érték az argumentuma, ezért:

```

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

```

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

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

### 4.2.1.1. Mohó kiértékelés rekurzív függvények esetén

A faktoriális matematikai definíciója:

```

fac(n) = if n = 0 then 1 else n * fac(n-1)
fac n = n*fac(n-1)

```

A faktoriális-függvény megvalósítása SML-ben:

```

(* fac n = n!
 fac : int -> int
*)
fun fac n = if n = 0 then 1 else n * fac(n-1)
Mohó kiértékelésénél minden n = 4 mellett:
fac(4)→4*fac(4-1)→4*fac(3)→4*(3*fac(3-1))
→4*(3*fac(2))→...→4*(3*(2*1))→...→24

```

A rekurzív kiértékelés szigorúan követi a matematikai definíciót.

### 4.2.1.2. Iteratív függvények

A fenti kiértékelésben az a rossz, hogy a rekurzív végrejátszás 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. Írunk 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, np)
 fun fac : int -> int
 *)
 fun fac n = faci(n, 1)
```

Nézzük a kiértékélés (egyes trivialis lépéseket összevonunk):

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

Kiérkélezés közben a P segédváltozóban (*az ún. gyűjtőargumentumban, akkumulátorban*) gyűjti üp. Az ilyen a részrendelényt, ezért a tárigény állandó marad. A kiértékélés lehát *iteratív* jellegű. Az ilyen rekurziót *terminális rekurzióknak* vagy *jobbreakurzióknak* (angolul: *tail* vagy *terminal recursion*) nevezzük. A jó fordítóprogramok felismerik az iteratívról rekurziót, és még hatékonynabb tükrözéket állítanak el. A faci(n-1, np) rekurzív hívás eredménye – további számítások nélküli – közelítőleg a faci(n, p) eredményét adja. Az ilyen, ún. *terminális hívás* végre lehet hajtani úgy, hogy az értelmező vagy fordítóprogram az n és a p új értékeit, majd visszaigirk a kod elejére áthelyet, hogy tenylegesen, tibol megihnia a függvényt. Az előző változatban a fac(n-1) hívás  *nem terminális hívás*, mert az eredményét még kell szorozni n-hel.

A faci-t rekurzív függvényt definíáltuk, ezért viszonylag könnyű belátni a helyességet, ugyanakkor a kiértékélés a segédváltozó bemezetésérrel literárvá vált. Nagyon sok rekurzív függvény (de nem minden!) literárvá alakítható segédtároló bemezetéssel, és így tárterületet takaríthatunk meg. Sokszor a végrehajtási idő is csökken, de sajnos néha nöhet is. Ha nem nyilvánvaló a nyereség, a lehető legtermészetesebb módon kell felirni az algoritmust. Esetünkben **faci** valanival rötekeronyabb **fac-nal**, ugyankakor a működése nehezebb érthető meg.

#### 4.2.1.3. Feltételes kifejezések speciális kiértékélése

A feltételes operátor (*if-then-else*) nem függvényhívás: a részhelyettesítések kiértékélésére csak akkor kerül sor, ha és amikor szükség van rajta:

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

E1-re akkor van szükség, ha E igaz, E2-re akkor, ha E hamis. Az E1-re akkor van szükség, ha és amikor szükség van rajta:

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

Nézzük most egy olyan példát, amelyben a végétlen rekurziót kerüljük el a használattukkal:

```
(* even n = (n mod 2 = 0) ;
(* isPwr0f2 : int -> bool
even : int -> bool
*)
fun even n = (n mod 2 = 0) ;
(* isPwr0f2 {x = y * y} ly = 2 * 2] → x * x [x = y * y] ly = 4]
*)
fun isPwr0f2 n = n = 1 orelse even n andalso isPwr0f2(n div 2) ;
```

isPwr0f2 megvizsgálja, hogy egy szám 2 egész hatványa-e. Kiérkélezés azonál véget ér, mivelyst eldönthető az eredménye.

Ha **andalso** és **orelse** fenti alkalmazásakor minden alkalmonnal minden a két operandusukat ki kellene értékelni, a rekurzió sohasem fejeződne be, **andalso** és **orelse** épben azért lassítja kiértékélését, hogy az ilyen és hasonló eseteket elegánsan lehessen kezelni.

#### 4.2.1.4. Gyakori feladat

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

#### 4.2.2. Lusta kiértékélés

Láttuk, hogy mitteletvegés, függvényhívás előtti szösszefoglalések vagy épben káros előre kiszámított az operátorokat, mert az végétlen rekurzízból, illegális indexelésről (indexhatók tőlőre indexeléshez, 0-ral való osztáshoz stb.) vezethet. Az SML-ben, mint láttuk, a programozó nem írhat olyan függvényt, amely hista kiértékélését lenne. Vannak azonban olyan nyelvek, amelyekben az argumentumot nem értékként, hanem kiérjeziként adjuk át.

A procedurális programozást megeremégi Algo60 a korábban már említett *név szerinti (call-by-name)* paraméterát alkalmazza: a függvény törzse minden parameter összes előfordulását az aktuális paraméterként átradott teljes kifejezéssel helyettesíti. (Ne téveszük össze a számos procedurális nyelvben, így például a Pascalban és a C-ben alkalmazott *hivatkozás szerinti (call-by-reference)* paraméterátadással!) Például a

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

hívás *név szerinti paramétertudás esetén* azonnal, az argumentum kiértékélése nélküli 0-t ad eredményt!

Sajnos, a név szerinti paramétertudás viselkedése sem minden kedvező. Pl. az **sq(sq(sq(2)))** hívás esetén **sq** minden egyes meghívása megkészíti az argumentumok számat:

```
sq(sq(sq(2))→sq(sq(2)) * sq(sq(2))→(sq(2 * sq(2)) * sq(sq(2)))
→((2*2) * sq(2)) * sq(sq(2))→...→(* (2*2) * sq(sq(2))→...
```

Aligha ezt akarjuk. A *lusta kiértékélés* (azaz a *szükséges szerinti hívás*) garantálja, hogy minden argumentumot csak egyszer kelljen kiérteni! azikor, amikor *elözött* van rá szükseg. Nem a kiérjezés helyettesítjük lehűt a törlésbe, hanem egy, a *kiérjezés utaló hívóhöz* (egy olyan mutató), amely *el von rejt*, amelyhez a programozó nem fejhett 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 számítóegységben a függvények és argumentumaiat irányított gráfalkatában: a gráf egy részének kiértékélésékor a graffot az eredménytől kapott értékkel frissítik. A lusta kiértékélés működési elvétől megérthetően irányított gráf helyett most jelenjük  $x = [E]$ -vel azt, hogy  $x$  összes eljáratulása osztózik az  $E$  értéken. Nézzük pl. **sq(sq(2))** lusta kiértékélését!

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

#### 4.2.3. A mohó és a lusta kiértékélés összehetetése

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

1.  $\text{zero}(E) = 0$  értelmes lenne akkor is, amikor  $E-t$  nem lehet kiértékelni. Ez ellenintmond a hagyományos matematikai szemléletnek, amely szerint egy kifejezés csak akkor értékkelhető ki, ha minden részkifejezése kiértékellehető.
2. A végtelelen adatszerkezetek bonyolulttá teszik a program helyességének igazolását. A lusta kiértékelt kifejezés. De ha állandóan a kiértékelési mechanizmusról kell gondolunk programozás közben, akkor nem sokkal jutottunk elõbbre, mint a változók pillanatnyi állapotát (azaz a program állapotterében) figyelembe vevő imperativ programozás esetén.
3. A hatékonysággal is vannak bajok: néha nyerünk, máskor veszünk a lusta kiértékellessel. Mint láttuk, faci mödhó kiértékelés mellett hatékonyabb fac-nál, mert az `if p` szorzást, azonban vegrehajtja. Lusta kiértékelés mellett faci ( $n$ ,  $p$ ) ugyan n-et azonnal kiszámítja, hiszen szüksége van rá az  $n=0$  vizsgálat elvégzéséhez, a  $p$  kiértékelését azonban késlelteti és a százokat akkumulálja:

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

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

A lokális kifejezéssel szemben a lokális deklarációban az in és az end kulcsszavak közötti is *deklaráció* áll, nem kifejezés. A lokális deklaráció célja az, hogy egy deklarációt egy másik deklarációtól belül lokálissá tegyen, elegendően a körüljárásban a szekvenciális deklarációt (szekvenciális deklaráció) is lehet.

## 5. fejezet

### 5.3. Egyidejű deklaráció

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

`val id1 = E1 and ... and idn = En`

Az egyidejű deklaráció előbb kiszámítja az összes  $E_{i-t}$  (kiszámításuk sorrendje), mivel nem lehet mellekjével, közömbös, majd a kiszámított értékeket balról jobbra haladva, rendre hozzárendeli a megfelelő *id<sub>i</sub>*-hez. Példáknak bemutatunk egy nem igazán hatékony megoldást az egész számok páros, il. páratlan voltali tesztelő even, ill. odd függvény megvalósítására:

```
(* even n = igaz, ha n páros
even : int -> bool
odd n = igaz, ha n páratlan
*)
let D in E end
D sokszor nem egyetlen deklaráció, hanem $D_1 ; D_2 ; \dots ; D_n$ alakú deklarációsorozat, más néven szekvenciális deklaráció, ahol a ; (pontosvesző) opcionális.
```

Lokális kifejezésben *értelek*, *függvény típus* és *kivétele* deklarátható. Az így deklarált lokális érték csak magában a lokális kifejezésben látható.

Most arra mutatunk példát, hogy mikor ne használhatunk lokális kifejezést:

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

Ez a programrészlet azért nem szerencses, mert az if-then-else felételek kifejezésben már nem lászik, hogy az a valójában a sin x-ét, a b pedig a cos x-et jelöl. Inkább az általai 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 változóban a min név világosan utal arra, hogy a sin x és a cos x közül a minimálisat, vagyis a kisebbiet kell eredményül adni.

Törekedjünk arra, hogy értelmes jelentésű (és értelmes nevű) függvényeket, definíljunk, és használjuk ki a programozási nyelv absztraktós lehetségeit!

### 5.2. Lokális deklaráció

A lokális deklarációt a local kulcsszó vezeti be:

`local D1 in D2 end`

Milyen típusú ítt az  $x^2$ ? Mindegy! Az id függvény ún. *polimorf függvény*, az x pedig *politípusú* azonosító. A tipusnák elnémelében a politípus jele  $\alpha$ ,  $\beta$ ,  $\gamma$  stb., az SML-ben  $\alpha$ ,  $\beta$ ,  $\gamma$  stb. Az SML-értelmező válasza a fenti definícióra telthet a következő:

`> val id = fn : 'a -> 'a`

A *perejellel* kezdődő típusneveket ('a-t, 'b-t, 'c-t stb.) típusválfázónak nevezük, és *alfának*, hétnak, gummának stb. olvassuk.

Az egyenlőségviszsgálatot is megengedő ún. *egyenlőségi típusok* (equality types) típusváltozóinak jelölésére két *perejellel* kezdődő neveket használunk:  $\text{`a}$ ,  $\text{`b}$ ,  $\text{`c}$  stb.

A polinorf típus: *típusérzéma*. Amikor a típusváltozói konkrét típussal helyettesítjük, e séma egy-egy példányát kapjuk.

Nézzünk két újabb, nagyon egyszerű példát polinorf függvények definíálására!<sup>1</sup> Egy pár első, ill. második tagának kiválasztására használhatók az alábbi *projektios* függvények, ahol  $\text{`a}$  és  $\text{`b}$  nem feltétlenül különböző típusok:

```
(* fst : `a * `b -> `a
*)
fun fst(x,_) = x
(* snd : `a * `b -> `b
*)
fun snd(., y) = y
```

## 6.2. Lista létrehozása

Két konstruktorművelet használható lista létrehozására: a  $\square$  (`nil`) *konstruktőr állando* és az *infix pozícióú konstruktőr operátor*.<sup>2</sup>

Egy lista vagy üres ( $\square$ ) lehet, vagy  $x :: xs$  alakú, ahol  $x$ -sel a lista *fejé*,  $xs$ -sel pedig a lista *farkája*, azaz az eredetivel egyelőző részlistáját jelölik. Könnyen elérhető egy lista *első*, sok munkával az *utolsó* eleme.

A [3, 5, 9, 13, 17, 21] *jelelés*<sup>3</sup> rövidítés, mégpedig a  $3 :: (6 :: (9 :: nil))$  operátor *jobbra két*: 3 :: 5 :: 9 :: nil. Egy lista elemeinek *zárójelet* írásnál, az *infix* :: \ (nagyító) operátor *jobbra két*: 3 :: 5 :: 9 :: nil. Egy 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)
```

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

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

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

- vagy üres,

- vagy egy elemből és ez az elemet követő listából áll.

Az üres listát – a listaüvegetek *egységelementet* –  $\square$ -vel vagy *nil*-el jelöljük. A legalább egy elemből álló lista első elemét a lista *fejének*, a többi elemből álló listát a lista *farkának* nevezünk.

A listában az elemek sorrendje fontos. Egyes elemek ismétlődhetnek. Az elemek típusa tetszőleges, de egy listának csak azonos típusú elemei lehetnek.<sup>1</sup>

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

## 6.1. Típuskifejezések és típusoperátorok

Ha egy lista elemeinek a típusa ‘a’, akkor a lista típusa ‘a’ list. Az üres lista típusa is ‘a’ list, ha csak nem alkalmazunk típusnegatítest. Az ‘a’ list az int list-hez hasonlóan *típuskifejezés*; a list, akárcsak a \* és a -> típusoperátor.

A típusoperátoroknak van *precedenciája*. A list *postfix*: a \* és a -> *infix* pozíciójú típusoperátor. \*, a legkisebb pedig a -> precedenciája. A list *precedenciája* a legmagyobb, nála kisebb a egyenlőségiel beljes jobb oldalán egyenmással ekvivalens típuskifejezések vannak.

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

<sup>1</sup> Más funkcionális nyelvekben, pl. a LISP-ben a listának különböző típusú elemei is lehettek.

## 6.3. Egyszerű műveletek listákkal

### 6.3.1. Lista elemeinek szorzata

A rekurzív megoldást általában az előforduló esetek elemzésével, a jellenző 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 egysége.

A  $\square$  *mintha* csak az üres listára illeszkedik. Az  $n :: ns$  mintha csak olyan listára illeszkedik, amelynek legalább egy eleme van; a mintha zárójelbe kell rakni, mert a *függvényalkalmazás precedenciája nagyon a négyespontról*.

```
(* 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 *ágak* (a Prolog szóhasználatával: *klázik*) a jelen esetben *kölcsönösen kizárták* egnámi (a mintha diszjunktak), ezért a két ág sorrendje az *eredmény szempontjából* közömbös. De nem közömbös a sorrendjük *hatékony szempontból*, mivel a vizsgált lista minden adott nem üres, amint a rekurzív feloldogozás során el nem fogya az elemet, az összesítés-vizsgálat az utolsó eset kivételével meghiúsul. Ezért a hatékonyág érdekelben a két ágat célszerű fordított sorrendben felrinni:

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

Ha a függvény meghívásakor már az első minta illeszkedik az argumentumra, a második ág kiérte teljesére nem kerül sor. (Amint tuduk, a kifejezés kiértékelése balról jobbra és fejlőről lefelé halad.) Hasonló esetekben mindenig törekedni fogunk arra, hogy hatékony megoldást alkalmazzunk.

A listaelemek összege hasonlóan képezihető. Az összadás egységeleme a 0.

<sup>2</sup>  $\square$  *es nil* jelentése azonos. A :: konstruktőr operator helyett solezor a sole azonos hatású de prefix pozíciójú *cons* konstruktőr függvényről beszélünk, amely azonban nincs belső függvényként definíálva az SML-ben.

<sup>3</sup> Az SML-lista származixa csak *hasonló* a Prologban való [5 | 6] és [5, 6]-tal azonos.

### 6.3.2. Lista legnagyobb eleme

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

- törés listának minden elem legnagyobb eleme,
- egyelemlű listában az egyetlen elem a legnagyobb,
- legalább kételőnlisztá esetén a legnagyobb elemet úgy kapjuk meg, hogy vesszők a két elem közül a nagyobbat, továbbá a maradványlistá elemei közül a legnagyobbat, és e kettő közül kiválasztjuk a nagyobbat.

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

Az SML-értelmező erre a függvénydefinícióra figyelmeztető üzenettel válaszol:

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

- Megjegyzések:
1. maxl üres listára nem alkalmazható, erre figyelmeztet a fenti üzenet. Később megnutatjuk, hogyan kell kezelni az ilyen, ún. *kiveteléket*.
  2. az [m] minta csak *egyetlen elemű* álló listára illeszkedik.
  3. az (m::n::ns) minta csak olyan listára illeszkedik, amelynek legalább két eleme van.

4. Az algoritmus szempontjából mindenleg lenne, hogy a lista minden milyen típusúak, de a  $>$  reláció, mint tudunk, többszörösen terhelhető módon politom (l. 5.4. szakasz). Mivel a programozó nem hozhat le létre többszörösen terhelhető neveket az SML-ben, a függvény definíálásakor el kell döntenünk, hogy a  $>$  relácionál helyük változtatni kell beépíténi maxl-be (alapértelmezés szerint írt a többszörösen terhelhető névfeleletet argumentumainak típusa). Később megnutatjuk, hogyan kell ún. *generikus* algoritmusokat irni.

### 6.3.3. Karakter, füzér és lista

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

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

A kapott lista minden eleme egyetlen karakter.

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

Füzérrelből álló lista elemeit egyszerűen a concat-tal lehet:

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

A következő szakaszokban néhány fontos listakezelő függvényt definiálunk.

### 6.4. Listák vizsgálata és darabokra szedése

Hárrom függvényt mutatunk be bennben 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.<sup>4</sup>

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

Az alábbiást (.) minden esetben illeszkezik. Olyankor használhatjuk, amikor az illeszkedő értékre nem kell hivatkoznunk a függvény törzsében.
Az eredményt szempontjából a minták felirásának sorrendje közömbös ebben a függvényben is.

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

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

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

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

```
(* drop(xs,i) = xs
drop(xs,i) = [x0,x1,...,xi-1,xi,xi+1,...,xn-1]
*)
length xs = xs eléméinek száma
```

Ebben a csoportban is hárrom függvényt mutatunk be: length egy lista hosszát adja eredményül, take egy lista elejéről vett adott számú elemből, drop egy lista elejéről adott számú elem elhagyásával kepez eredménylistáját.<sup>5</sup> Ha

akkor

```
length xs = n
take(xs,i) = [x0,x1,...,xi-1]
drop(xs,i) = [xi,xi+1,...,xn-1]
```

lengthnai változata a következő:

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

Egy példa a függvény alkalmazására:

<sup>4</sup>hd a head (fej), tl a tail (farok) szóból származik. null, hd és tl fenti definíciója csak illusztráció, ugyanis minden hárrom belső függvényt, take és drop pedig a List könyvtárban van definiálva az SML-ben.

<sup>5</sup>length belső függvény, take és drop pedig a List könyvtárban van definiálva az SML-ben.

```

- nlength [[1,2,3] [4,5,6]];
 > val it = 2 : int
nlength rossz hatékonyiségi, mert nem iteratív: az 1-esek a veremben csak gyűrűnek, gyűrűnek, amíg
a maradéklásak ki nem ürül. A függvény javított, iteratív változata:
(* length xs = xs elemeinek száma
length : 'a list -> int
*)
local
 fun addlen (n, _::xs) = addlen (n+1, xs)
 | addlen (n, []) = n
in
 fun length xs = addlen(0, xs)
end

```

A *nagyon gyakran használt* (pl. könyvtárott) függvények *hatékonyisége* és *robosztussága*<sup>6</sup> fontos, hogy ha kevésbé szépek, kevésbé olvashatók is. A speciális feladatokra írt, *rőlükben használt* függvények azonban legyenek *könnyen olvashatók*, és a *helyesséjüket* is *egyszerűen* lehessen belátani, bizonyítani.

take első változata:

```

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

```

A benne lévőt változtat az *if-then-else* alkalmazása miatt nem olyan elegáns, de robosztus: negatív i-re az üres listát adja eredményül. Majdnem ugyanez van minden szébben:

```

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

```

take második változata negatív i-re a *teljes listát* visszaadja. (Miért?) *Fügyelem:* ebben a definícióban az ágak sorrendje nem közömbös! (Miért nem?)<sup>7</sup>

Nézzünk egy példát take egyszerűsítésére (egyes trivialis lépéseket összevonunk):

```

take([9,8,7,6],3) ->9::take([8,7,6],2) ->9::8::take([7,6],1)
->...->9::8::7::[] ->9::8::[7] ->9::[8,7] ->[9,8,7]

```

<sup>6</sup>Tegy eljárás, függvény, program, stb. akkor robosztus, ha szélesítéges körülmenetek között is a specifikációjának megfelelően, megújításon, kiszámításon viselkedik. Szélsőséges körülmenetek számít például, ha egy függvény minden előforduló, extrem értékre alkalmazunk.

Az SML-ben, egy függvény robosztúságát, jelenti, hogy a függvény az értelmezési tartományba eső minden lehetséges argumentumra specifikálva van, és e specifikáció szerint viselkedik. Például a *belső id* és *tl* függvény, ha üres listára alkalmazzuk, meghatározott *kwedell* jelez az SML-ben. A kivételezhető *take* itt bemutatott, hogy a minden lehetséges értékre definíálva vanak: a rekurszió minden esetben visszatérítésre készítők, ha i negatív, az első az üres listát, a második az eredeti lista adja eredményül. Egyaránt ennek a robosztus megoldásnak haténya is van: valósult, hogy i < 0-ra szandikos ritkán fogják alkalmazni *take*-et, ha pedig valamilyen hiba folyamánkent lesz negatív az i, az eredeti lista majd csak jóval később keletkezik, és ezért nehezebb lesz felidéreni. Ezért, a könyvtári *List.take* (és *List.drop*) ik-ora *kwedell* jelez.

Az egyszerűsítés folyamatot bemutató példában a négyesponiot (:) nem lista létrehozására használjuk, mint a programokban, hanem olyan listakifejezéseket írnak fel vele, amelyeket az egyszerűsítés során az SML-errelnevezének ki kell értékelnie. A lista elemeit az SML-értelmező előbb egyszerű berakja a verembe, majd hátról visszafelé haladva megnézi először őket az eredmény-lista eltolásáshoz.

Következő kérdésünk az, hogy érdemes-e megírni *take* iteratív változatát? Próbáljuk meg: a részeredményeket gyűjtésük 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;
 (* drop(xs, i) = az xs első i db elemeket elhagyásával
 elölálló 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;

```

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

```

 (* drop(xs, i) = az xs első i db elemeket elhagyásával
 elölálló 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 megsötétítés és szereposére iteratív is. Az else ágban lévő x::xs lista ugyanaz, mint a drop második argumentuma; részibb az összetett rendezésű szövegben tömörebb jelölést – a réteges miniat –, amely ilyen esetekben alkalmazható.
```

## 6.6. Listák összefűzése és megfordítása

Ebben a szakaszban két függvényt, az *append*-et és a *rev*-et mutatjuk be. Az *append infix* változatát a @ jellegű jelöljük. A ket listákat csak előlről visszafele haladva frisszik az elemeket az ys-hez, ugyanis a listákat csak előlről tudjuk felírni.

```

 (* append(xs, ys) = xs összes eleme ys előli füzve
 append : 'a list * 'a list -> 'a list
 *)
 fun append ([], ys) = ys
 | append (x::xs, ys) = x::append(xs, ys);

```

*Infix* változat pl. így definíálhatjuk:

```

- infix 5 @;
- val @ = append;

```

Itt is, akarsak *take*-nel, a lista építésének költsége meghaladja a veremhasználat – ti, az adatok veremben való tárolásának – költségét, ezért nem érdemes iteratív változatot kidolgozni. A Pascal, a C explicit mutatókkal kezelni a lista-kat, ezért az egik lista régen a mutató átirányítója a másik listára. Az ilyen, ún. destruktív frissítések gyorsabb, mint a másoló frissítések. Csak hogy ez veszélyes lehet! Pl. mi van akkor, ha minden argumentum ugyanarra a lista mutat?

Most nézzük a listát megfordító rev egy naív megoldását:

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

és egy példái nrēv redukciójára:

```
nrēv([1,2,3,4]) -> nrēv([2,3,4])@[1] -> nrēv([3,4])@[2]@[1]
-> nrēv([4])@[3]@[2]@[1] -> nrēv([])@[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 megnéz elmenie kell bontani, majd összerakni 0, 1, 2, ...,  $n - 1$  lépésekben.  
nrēv nagyon rossz hatékonysság:  $O(n^2)$ . De emlékezzünk csak vissza rtake-re: ott megfordult a listaelemek sorrendje, bár nem akartuk. Most pontosan ezt akarjuk használniunk tehát segédargumentumot a revto segédfüggvényben:

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

revto lépésszámra árányos a lista hosszával. Segítségével rev definíciója (revto lokális is lehetne nyerhető):

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

Egy 1000 elemű listát rev 10000 lépében, míg a rev  $\frac{1000 \cdot 1001}{2} = 50500$  lépében fordít meg. Hatalmas a nyereség!

## 6.7. Listákból álló lista, párok ból álló lista

Ebben a szakaszban megnézük hármon függvényt definíciójunk. flat két széres mélységrű listából egyszeres mélységrűt készít; combine két azonos hosszúságú lista elemeiből egyetlen, párok ból álló listát állít elő; split pedig combine inverz függvénye.

flat specifikációja és definíciója:

```
flat([x1,x2,...,xm],[y1,y2,...,yn]) = [x1,x2,...,xm,y1,y2,...,yn]
(* flat xss = a két széres mélységrű xss lista részlistáinak
 elemeiből képet listá
*)
flat [] = []
| flat (ls@:lss) = ls @ flat lss;
```

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

```
combine([x1,x2,...,xm],[y1,y2,...,yn]) = [(x1,y1),(x2,y2),..., (xm,yn)]
```

(\* 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);
| combine (x::xs, y::ys) = (x,y)::combine(xs,ys);
| combine (x::xs, y::ys) = (x,y)::combine(xs,ys);
```

Az SML-én belmező figyelmeztet: nem fedtünk le minden esetet (ui. az argumentumunkat átadt listák különböző hosszúságúak lehetnek). Az ilyen esetek kezelésére a később ismertendő *kivételkezelés* alkalmas.

split-et combine inverz függvényeket definíáljuk:

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

Iteratív változata segédargumentumokban gyűjti külön-külön a két listát, de sajnos megfordítja:
(* 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);
| rsplit (x,y) = (x,y);
```



Hivásorozatot állítja elő. Ugyanakkor a véghajtása nagyon gyors, például a fibpair 44 hívás azonnal kiírja a (433494437, 701408733) számpárt, amelynek a második tagja az Int. Maximál nél megszűnenek nem nagyobb Fibonacci-szám. A keresett Fibonacci-számot a számpár második tagjának különbözősével kapjuk, például:

```
- #2(fibpair 44);
```

```
> val it = 701408733 : int
```

Szép, általánosított és gyors megoldást kapunk iterációval:

```
(* interfib(n,p,c) = a (p,c) Fibonacci-számpárt követő
 n-edik Fibonacci-szám (n>0)
 interfib : int * int * int -> int
 *)
 fun interfib (1, prev, curr) = curr
 | interfib (n, prev, curr) = interfib(n-1, curr, prev + curr)
```

Az else ágban a rekurzió terminális, fib interfib-et hívja meg az n-edik Fibonacci-szám elcáfításához:

```
(* fib n = az n-edik Fibonacci-szám
 fib : int -> int
 *)
 fun fib 0 = 0
 | fib n = interfib(n, 0, 1)
```

Nézzünk egyszerű példát fib redukciójára:

```
fib 7->interfib(7,0,1)->interfib(6,1,1)->interfib(5,1,2)
->...->interfib(1,8,13)->13
```

Az interfib függvényt célszerű lokálissá tenni fib-ben:

```
(* fib n = az n-edik Fibonacci-szám
 fib : int -> int
 *)
 local
 (* interfib(n,p,c) = a (p,c) Fibonacci-számpárt követő
 n-edik Fibonacci-szám (n>0)
 interfib : int * int -> int
 *)
 fun interfib (n, prev, curr) =
 if n = 1 then curr else interfib(n-1, curr, prev+curr)
 in
 fun fib 0 = 0
 | fib n = interfib(n, 0, 1)
 end
```

Ebben a példában a lokális deklaráció helyett lokális kifejezést is használhatnánk, de ez nem minden van igy.

### 7.3. Egész négyzetgyök közelítésséssel

Az SML-ben négyzetgyökvonásra a Math könyvtárbeli Math.sqrt függvény használható. Most egy egész szám egész négyzetgyökének közelítésére írunk SML-függvényt. Az n szám k egész négyzetgyöke kielégíti az alábbi egyenlőséget:

$$k^2 \leq n < (k+1)^2$$

Hogyan számíthatjuk ki a k-t rekurzióval? Még kell találnunk a megfelelő részfeladatot, amely az eredményt használja. A lineáris mellett a fejezéses a másik szokszor alkalmazható alapmódszer, probálkozzunk most az utóbbitival.

$\sqrt{4n} = 2\sqrt{n}$ , tehát ha n-et 4-szel osztjuk, egyszerűsítjük a feladatot. Nem biztos, hogy n osztatható 4-szel, ezért az  $n = 4m + v$  segíteni fogja használni, ahol  $v = 0, 1, 2, 3$  lehet. Mivel

$$\begin{aligned} m < n, m \text{ egész négyzetgyökét a megrándított függvény rekurzív alkalmazásával kereshetjük.} \\ i^2 \leq m < (i+1)^2 \\ m \text{ is, } i \text{ egész, leh\'at ha } m \text{-hez } 1\text{-et adunk, legfeljebb egyenl\'o lehet } (i+1)^2\text{-nel:} \\ m+1 \leq (i+1)^2 \end{aligned}$$

Vonjuk össze a fentiegyenlőségeket, és szorozzuk be minden tagját 4-szel (mivel  $n = 4m + v$  és  $v < 4$ , ezért  $n < 4m + 4 = 4(m+1)$ ):

$$(2i)^2 \leq 4m \leq n < 4m + 4 \leq (2i+2)^2$$

Egészségről léven 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, mert ha igen, akkor az eredménytől kapott értékhöz még 1-et kell adnia. Erre szolgál az increase függvénny:

```
(* increase(j, n) = j+1, ha n négyzetgyöke nem kisebb j+1-nál,
 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. Tanúság: a hatékony-ságomás oka általában a valászott algoritmusban, pontosabban annak szerkezetében, nem pedig rekurzív voltaiban keresendő.

### 7.4. Valós szám négyzetgyöke Newton-Raphson módszerrel

Ha a négyzetgyökének egy közeliése x, akkor  $\frac{x+x}{2}$  a négyzetgyök egy jobb közelítése. x kezdeti értéke legyen 1. A közelítés akkor ér véget, ha  $\left| \left( x - \frac{x+x}{2} \right) / x \right| < \varepsilon$  előre meghatározott ε értéknél, az előiről relatív pontosságnál kisebbé válik. A függvény negatív számnál nem hívjuk meg, de 0-ra még jó eredményt kell adnia. Egy megrándított SML-ben (az ε neve a programban eps):

```
(* findroot (a, x, eps) = a négyzetgyöke Newton-Raphson közeliéssel,
 eps relativ pontossággal, ahol x az előző
 közeliöt 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 mindenkitől globalis findroot számának.
2. Jobb, ha findroot kívülről nem látsszik. A javított változatban sqroot-on belül lokális eljárásnék definíljunk.

A javított változat:
(* sqroot a = a négyzetgyöke Newton-Raphson közelítéssel
 sqroot : real -> real
*)
fun sqroot 0.0 = 0.0
| sqroot a =
 let val eps = 1E-10
 (* findroot x = a négyzetgyöke eps relatív pontossággal,
 ahol x az előző közelítő érték és a > 0.0
 findroot : real -> real
 *)
 fun findroot x =
 let val nextx = (a/x + x) / 2.0
 in if abs(x - nextx) < eps * x
 then nextx
 else findroot nextx
 end
 in
 findroot 1.0
 end

```

## 7.5. $\pi/4$ közeliítő értéke kölcsönös rekurzióval

Végül lássunk egy példát olyan kölcsönösen rekurzív függvények használatára, amelyek  $\pi/4$  értékét határozzák meg (nem igazán hatékonynak módon) az alábbi közelítő polinom alapján:

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \dots + 1/(4k+1) - 1/(4k+3) + \dots$$

Pos-sal a sorozat pozitív, neg-gel pedig a negatív előjelű tagjait számítattuk ki. d-vel a soron következő tag nevezőjét jelölik.

(\* pos d = pi/4.0 közeliítő értékének pozitív előjelű,
 d nevezőjű tagja (d = 1.0, 5.0, 9.0, ...)

```

pos : real -> real
neg d = pi/4.0 közeliítő értékének negatív előjelű,
 d nevezőjű tagja (d = 3.0, 7.0, 11.0, ...)
neg : real -> real
*)
fun pos d = neg(d - 2.0) + 1.0/d
and neg d = if d > 0.0 then pos(d - 2.0) - 1.0/d else 0.0
pos és neg felhasználásával számítja ki π értékét a sum függvény:
(* sum n = pi n-edik közeliítő értéke (n>0)
 sum : int -> real
*)
fun sum n =
 let val d = real(2*n+1)
 in 4.0 * (if n mod 2 = 0 then pos d else neg d)
 end

Segédargumentum alkalmazásával a kölcsönösen rekurzív függvények sokszor egyetlen függvényrel helyettesíthetők:
(* sum n = pi n-edik közeliítő értéke (n>0)
 sum : int -> real
*)
local
 (* pi4(d, s)=pi/4.0 d nevezőjű, s előjelű közelítő értéke
 pi4 : real * real -> real
*)
 fun pi4 (d, s) =
 if d > 0.0 then pi4(d-2.0, ~s) + s/d else 0.0
 in
 fun sum n =
 let val d = real(2*n+1)
 in pi4 (d, s) = if n mod 2 = 0 then 1.0 else ~1.0
 4.0 * pi4(d, s)
 end
 end
end

```

## 7.6. A következő permutált

Addott egészek egy sorozata. Permutáljuk úgy a sorozatot, hogy az eredmény *lexikografikusan eggyel nagyobb* legyen az eredetinél! Tekintsük például a következő sorozatokat:

123 → 123 → 132 → 132 → 1423 → 1423 → 2134 → ... → 4321

Látható, hogy a jobb szélső kisebb helyiértékű elemek változnak gyakrabban (aláhúzással jelölik a helyiérték változtatott elemeket). 4321-nyel a permutálás véget ér, mivel minden nála lexikografikusan nagyobb sorozat.

Lexikografikusan nagyobb sorozatot permutálással úgy állíthatunk elő, hogy a sorozat egy elemét felcseréljük egy töle jobbra álló, nála nagyobb elemmel,

1243 → 1423.

Pontosan egyetlen nagyobb sorozat – a következő permutált – úgy állítható elő, ha a cserében a legkisebb helyiértékű elemhez, azaz a sorozat jobb széléhez lehetőleg kisebb álló elem vesz részt. Ezzel az elemmel kell felcsérni a hozzá legközelebbi, töle balra álló, nála kisebb elemet, pl.

$1243 \rightarrow 1312$

Ebben a példában a sorozat jobb székső elemével, a 3-nal cserélítik fel a hozzá legközelebbi, töle balra álló, nála kisebb elemet, a 2-t. Sajnos, az eredményül kapott 1322 sorozat nem jó, mert 1243 után 1324-jön lexicografikus sorrendben. De már látszik a megoldás: a lecsérült elemüket jobbra álló, monoton csökkenő részosorozatot meg kell fordítani, hiszen az így kapott monoton növekedő részosorozat lexicografikusan a lehető legkisebb, pl.

$1243 \rightarrow 1212 \rightarrow 1321$

Mivel a lista olyan szerkezetet, amelyet a fejtől kezdve lehet feldolgozni, a sorozatot úgy ábrázoljuk, hogy a legkisebb helyiértékű elem legyen a lista feje. A példában előforduló sorozatokat listaként írjuk ábrázoljuk:

$[4, 3, 2, 1] \rightarrow [3, 4, 2, 1] \rightarrow [4, 2, 3, 1] \rightarrow [2, 4, 3, 1] \rightarrow [3, 2, 4, 1] \rightarrow \dots$

Fogalmazzunk meg a következő permutáltat elvállító algoritmust!

1. Bontsuk három részre a listát:

- a lehető leghosszabb monoton növekedő részosorozatra ( $zs$ ) – itt a lehető legkisebb olyan elemet keressük, amelynek a bal oldalán nála nagyobb elem található;
- az ezt a sorozatot követő elemre ( $y$ ) – itt építen ezt az elemet akárjuk másra cserélni;
- a maradék listára ( $ys$ ) – amely ebben a lépésben nem változik.

Például

```
[2, 4 , 3 , 1]
zs y ys
```

1. Kereslik meg  $zs$ -ben azt a legkisebb  $z_i$ -t, amely  $y$ -nál azért nagyobb (ui. *eggyel nagyobb* sorozatot keresünk), és állítuk elő a  $z_i:ys$  részlistát (az előző példa esetén  $[4, 1]$ -et).

2. Fordításuk meg a  $[z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_n]$  maradéklistát, amely így csökkenő sorrendű lesz, és szűriuk bele  $y$ -t a megfelelő helyre (a példában a maradéklista a  $[2]$ ), és ha a 3-at beszünik,  $[3, 2]$ -t kapunk! Biztos, hogy lexicografikusan ennek kisebb részlista nincsen.

3. Állítsuk elő az eredménylistát a részlisták összeítésével (a példában  $[3, 2, 4, 1]$ -et kaphunk!)

Néhány megjegyzés a vázolt algoritmushoz:

a) Vegyük észre, hogy (3) egyik részfeladatát – ti.  $zs$  megfordítását – kényelmesebben elvégezhetjük

(1)-ben:  $zs$  t könnyebb elérve megfordítva elkállítani az eredeti listából; ez lesz az (1.) lépés. Így a módosított (3.) lépésben  $y$ -t csak be kell szurni a megfelelő helyre.

(1') E lépés eredményképpen telthet előáll

- a lehető leghosszabb monoton csökkenő részosorozat ( $zs$ ),
- a  $zs$ -t követő elem ( $y$ ),
- a lista változatlan maradványa ( $ys$ ).

Ezzel a feladat megoldása:

```
local
 fun next (zs, y:ys) =
 if hd zs <= y
 then next (y:zs, ys)
 else ...
```

A next függvény első részét  $zs$ ,  $y$  és  $ys$  előállítására írunk meg. Feltételezzük, hogy  $zs$ -ben kezdetben a jelenlegi lista első eleme van, és addig rakunk bele újabb elemeket ( $y:ys$ -ből, amíg  $zs$  monoton csökkenő marad).

(\* next(zs, ys) = a következő permutált

```
next : int list * int list -> int list
 *)

```

fun next (zs, y:ys) =

```
 if hd zs < y
 then next (y:zs, ys)
 else ...
```

ys+ most már változtatás nélküli használhatjuk fel a (3.) lépésben.

b) Most a (2), (3.) és (4) lépések megalosítsására írunk függvényt swap néven.

Adva van a monoton csökkenő  $zs = z_n, \dots, z_{i+1}, z_i, z_{i-1}, \dots, z_1$  sorozat. Elő kell állítani az ugyancsak monoton csökkenő  $z_n, \dots, z_{i+1}, y, z_{i-1}, \dots, z_1$  sorozatot.  $y$  és  $ys$  e függvény számára globális állandók.

Ha előállt az új sorozat, akkor a következő permutált is megvan:

$[z_n, \dots, z_{i+1}, y, z_{i-1}, \dots, z_1] @ [z_i] @ ys$

Rekurzív függvényt írunk.  $z:z'::zs$  nek kezdetben legalább két eleműnek kell lennie, hogy a sorozat monoton csökkenő legyen.

Ha már csak egyetlen elem maradt a feldolgozandó listában, annak biztosan kisebbnek kell lennie  $y$ -nál, mert ha nem lenne kisebb, akkor  $z>y=z'$  teljesítő volna az előző lépésben; így megvan az eredmény.

Ha valamely lépésben  $z>y=z'$  teljesül, a  $zs$  lista véget már nem kell megírásáho, ugyancsak megvan az eredmény.

Mindaddig, amíg  $z>y$  ( $i = n, \dots, 2$ ), a függvény saját magát hívja meg.

```
(* swap zs = a (2), (3) és (4) lépések eredményeként elállító sorozat
swap : int list -> int list
 *)
 fun swap [z] = y:z:ys
 | swap (z:z'::zs) =
 if z' > y (* z>z'>y, folytasd! *)
 then z::swap (z'::zs)
 else (* z>y=z' *)
 (y::z)::zs @ (z::ys);
```

Ezzel a feladat megoldása:

```
local
 fun next (zs, y:ys) =
 if hd zs <= y
 then next (y:zs, ys)
 else ...
```

```

else (* zs@[y]@ys, ahol zs monoton csökkenő sorozat *)
let fun swap [z] = y::z::ys (* z>y *)
| swap (z::z'::zs) =
 if z' > y
 then (* z>z'>y *)
 z::swap(z'::zs)
 else (* z>y>=z' *)
 (y::z'::zs) @ (z::ys)
 in
 swap(zs)
 end;
in
 (* nextperm zs = a következő permutált
 nextperm : int list -> int list
 *)
 fun nextperm (y::ys) = next([y], ys)
end;

```

Mielőtt tovább olvasná a jegyzetet, próbáljon meg önhallóan válaszolni az alábbi kérdésekre!

1. Megváltoztatható-e swap-ban az ágak sorrendje?
2. Mi van akkor, ha elérjük a legnagyobb permutált sorozatot, pl. elég áll az [1, 2, 3, 4] lista?
3. Mi van akkor, ha kezdetben length ys < 2?
4. Lefordítunk-e minden esetet a swap függvény definíciójában?

A válaszok:

- (A) Igen, hiszen kölcsönösen kizárták egy másikat kiválasztó feltételek.
- (B) Ilyenkor zs elemei monoton csökkennek, és már nincs következő elem (y: ys üres). Az eredménylistának monoton növekedő sorozatnak kell lennie, ezért zs-t meg kell fordítani. A next függvényt tehát az alábbiak szerint kell módosítani:

```

local fun next (zs, []) = rev zs
| next (zs, y::ys) = if hd zs < y ...

```

- (C) A kezdetben üres sorozatok kezelésére a nextperm függvény definícióját kell írni:

```

...in fun nextperm [] = []
| nextperm (y::ys) = next([y], ys)

```

end;

Kérdés: miért oldja meg ez a változatás a length ys < 2 eset kezelését?

- (D) Nem, hiszen [z] csak pontosan egy elemű, (z::z'::zs) pedig legalább két elemű listára illeszkedik. Az üres lista kezelését nem mondunk semmi. Csak azért hagyhatunk el a swap [] = [] változatot, mert a swap függvényt üres listával sohasem hívjuk meg.

A fenti javításokat is tartalmazó megoldás:

```

local
 fun next (zs, []) = rev zs
 | next (zs, y::ys) =
 if hd zs <= y
 then next(y::zs, ys)
 else (* zs@[y]@ys, ahol zs monoton csökken *)
 let fun swap [z] = y::z::ys(* z > y *)
 | swap (z::z'::zs) =
 if z' > y
 then (* z >= z' > y *)
 z::swap(z'::zs)
 else (* z > y >= z' *)
 (y::z'::zs) @ (z::ys)
 in
 swap(zs)
 end;
 in
 fun nextperm [] = [] (* ha a sorozat eleve üres *)
 | nextperm (y::ys) = next([y], ys)
 end;

```

## 8. fejezet

### Polimorfizmus

A polimorfizmus több váltojával alkalmazzuk a programozásban.

- Egy *polimorf név* egyetlen olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *parameteres polimorfizmus*.
- Egy *többszörösen terhelt név több* algoritmust azonosít: minden típusú argumentumra alkalmazható, amelyre fel; ez az ad-hoc vagy *többszörös terheléses polimorfizmus*.

- A polimorfizmus harmadik változatát *örökölődéses polimorfizmusnak* nevezzük. (Öröklődéses polimorfizmust használ az objektum-orientált programozás.)

### Egyenlőségvizsgálat polimorf függvényekben

Képzeljük el azt a függvényt, amelyik megvizsgálja, hogy egy ls lista minden eleme van-e egy bizonyos elem. Polimorf-e ez a függvény? A lista minden eleméről el kell tudni döntenı, hogy egyenlő-e vel. Csakholog 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  $\forall x \bullet f(x) = g(x)$ . Ezit általánosságban lehetetlen eldöníteni.

- Az absztrakt típusok közül az *abstype* deklarációval deklaráltakra csak az absztrakt típus-sal együtt definált 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 konstruktőrökön elvégzéhető az egyenlőségvizsgálat.<sup>1</sup>

Az egyenlőség telhát csak *körlátozott értelmemben* polinomf. *Egyenlőségi típusnak* (equality type) nevezzük az olyan típust, amelyen az egyenlőségviszgálat elvégezhető. Amint már említettük, az ilyen típusvállozókat az SML *két perjelzővel* ( prime-bél ) és egy betűből álló azonosítóval (' a, ' b, ' c stb.) jelílik. Pl:

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

Most már definíálhatjuk az *isMem* (eleme) függvényt, ill. operátort:

```
(* isMem(x, ys) = x = elem-e ys-nek
```

<sup>1</sup>Az *abstype* és a *datatype* deklarációiról később lesz szó a jegyzetben.

### 8.1. Polimorf halmazműveletek

```
isMem : "a * "a list -> bool
*)
fun isMem (x, ys) = [x] és ys orelse isMem (x, ys)
| isMem (_ , []) = false;
infix isMem;
```

A *newMem* függvény egy új elemet rak be egy listába, ha az ellen még nincs benne.

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

*newMem*, ha a sorrendől 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 elég rossz a hatékonysága. Szerencséssel, ha a halmazokat a megszokott halmazműveletekkel kezeljük. Most tövábbra is egyszerűbb listákat ábrázoljuk ötlet, de később valamiben hatékonyabb tárolást választhatunk pl. rendezett listát vagy bináris fát. Ót halmazműveletet definíálunk: unió (*union*, *SUT*), metszet (*inter*, *SUT*), részhalmaza (*isSubset*, *T ⊆ 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) =
 if x isMem ys then x :: inter (xs, ys) else inter (xs, ys)
| inter ([] , _) = [];
```

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

```
fun isSubset (x :: xs, ys) = (x isMem ys) andalso isSubset (xs, ys)
| isSubset ([] , _) = true;
infix isSubset;
```

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

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

A hatványhalmaz egy halmaz összes részhalmazának a halmaza, az eredeti halmazt és az üres halmazt is beleértve. Jelöljük S -sel az eredeti halmazt, S hatványhalmazat így állíthatjuk elő, hogy S -ből kiveszünk egy x elemet, és azután rekurzió módon elbocsátjuk az $S - \{x\}$ hatványhalmazát.
Ha térszöges T halmazra $T \subseteq S$ és $T \bigcup \{x\} \subseteq S$, így minden T minden x eleme S hatványhalmazának. A pws függvényben a base argumentum gyűjti a hatványhalmaz elérőit; kezdetben törleszek tell lennie.

(* pws(xs, base) = az xs halmaz hatványhalmazának és
 a base halmaznak az uniójá
 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 ugyanezzel rekurzív módon base-ben gyűjti az x elemeket, vagyis előállítja az összes olyan halmazi, amelyben x benne van. Halmazegyenlettel pws eredménye így adható meg:
pws(S,B) = {T ∪ BT ⊆ S}

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


```

Minden esetet le kell fedni minélával, különben hibaüzenetet kapunk. A minták letszegesen összetettek lehetnek (lehetnek benük ennek, listák, rekordok stb.). Például a `sirs` függvény az összes `Knight` nevű, összegyűjtő a `person` típusú személyek egy listájából:

```
(* sirs ps = az összes Knight nevénk listaja
 sirs : person list -> person 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` mintha nemcsak King-re, Peer-re és Peasant-ra illeszkedne (li. ezek helyett áll itt!). Hanem `Knight`-ra is.

Az összes diszjunkt eset felsorolása segíti az algoritmus helyességének felátlását, bizonyítását. Míért vontunk össze négy három esetet egyetlen változatban? Azért, mert a három eset részletezése hosszabbá tenné a program szövegét is, végrehajtását is. A bizonyítás sem olyan gondot, ha a harnadik sort *feltételek*nek tekintjük:

```
sirs(p::ps) = sirs ps if Vs p#Knight s
(* 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
```

## 9.2. A datatype deklaráció

Kezdjük egy példával!

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

Person néven új összetett típust hozunk létre. Az új típusnak négy *adtatkonstruktora* (röviden: konstruktora) van: King, Peer, Knight és Peasant; közülük King ún. *adtatkonstruktortörökötő*, a másik három ún. *adtatkonstruktorfüggvény*. Az adtatkonstruktorknak is van típusuk:

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

King (király) csak egy van, ezért definiálhattuk konstruktortállalunként. A Peer-t (főnemest) nemess cime (string), birtokának neve (string) és szózana (int), a Knight-ot (lovagot) és a Peasant-ot (paraszitot) csupán a neve (string)azonosítja. Példák:

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

Mintaillesztéssel választhatók szét az esetek, pl. egy függvényben:

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

Itt pl. a title (Peasant name) a *match* (pattern), és benne a name a *mintazonosító* (pattern identifier).

Gyakori, hogy egy név csak néhány különböző értékkel veleti fel (azaz a név által felvethető értékek halmozta száma nagy), ilyen esetben érdemes bevezetni a *felsorolás típus* (enumeration type). A datatype deklaráció használható felsorolás típus létrehozására is, pl. így

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

A felsorolás típusnak csak *konstruktorral* vannak. Az új típus alkalmazásához a person típus újra deklárálnunk kell:

```
datatype person = King
 | Peer of degree * string * int
 ...
degree típusú adatok feldolgozásakor külön-külön elemezniük az előforduló eseteket, pl.:
(* lady p = p főnemes hitvesének rangja
 lady : degree -> string
 *)
 fun Lady Duke = "Duchess"
 | Lady Marquis = "Marchioness"
 | Lady Earl = "Countess"
```

```
| Lady Viscount = "Viscountess"
| Lady Baron = "Baroness"

A belsejű bool típushoz hasonló Bool típushoz és hozzá a Not függvényt például így is deklarálhassánk:
ill. definíálhatnánk:
```

```
datatype Bool = True | False;
```

```
Not : Bool -> Bool
```

```
fun Not True = False
```

```
| Not False = True
```

### 9.3. Polimorf adattípusok

Láttuk, hogy List nem típus, hanem *posfix pozíciójú típusoperátor*, int list list, int list list, (string \* string) list stb. azonban már típusok. A datatype deklarációval *típusoperátor* is leírhatóunk.

A belső 'a' list típushoz hasonló 'a' List listát és vele együtt a Nil és a Cons *adatkonstruktőrököt* például így definíálhatnánk:

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

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

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

Bevezethetjük az *infix pozíciójú* :: : (hatospont) *adatkonstruktőropéárt*, hogy kényelmesebb jelést használhassunk:<sup>1</sup>

```
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 dobjat definiáltunk:

1. a kétargументű disun típusoperátor,

2. az In1 : 'a -> ('a, 'b) disun és

3. az In2 : 'b -> ('a, 'b) disun adatkonstruktőrfüggvényeket.

('a, 'b) disun az 'a és 'b típusok megkülönböztetett egyesítése. *Megkülönböztetettnek* nevezik az egységet, 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 × aláírak, ha x 'a típusú, és In2 'b típusú. Az In1 és In2 konstruktőrfüggvények olyan

<sup>1</sup> A :: : és az = közé szóköz kell rakeni, különben a fordítóprogram egyetlen írásjelékből álló jele sorozatot.

címkeinek tekinteléből, amelyek az 'a típusú megkülönböztetik a 'b típusú. (Megkülönböztetett egysétes például a Pascal variábilis rekordja is.)

A megkülönböztetett egysétes relatívén teszi, hogy különböző típusokat használunk ott, ahol egéréből csak egyetlen típusnak használhatunk (v.ö. az objektum-orientált programozással, ahol például egy *alakzat* osztálynak *tegely*, *háromszög* vagy *kör* nevű leszármazottai lehetnek). Az SML-ben megkülönböztetett egyséssel tudunk létrehozni például *különböző típusú elemekből álló listát*:

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

A lehetséges eseteket most is miniallesztéssel elnevezhetjük, pl.

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

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

Nézzük, mi a típusa az In1 "ú! Skócia" kifejezésnek!

Az In1 konstruktorfüggvény 'a -> ('a, 'b) disun típusú, ezért string típusú argumentumra alkalmazza (string, 'b) disun típusú értéket ad eredményül. Az In2 King kifejezés típusa nyilvánvalón ('a, person) disun.

Az (In2 King, In1 "Skócia") kifejezésben mindenekkel alapípus leköltjük, ezért ennek a két elemi listának a típusa a fent is látható ('string, person) disun) list. Ugranez lesz a háromelemű [In1 "ú!", In2 King, In1 "Skócia"] lista típusa is, hiszen az 'a típusváltatót az In1 konstruktőrfüggvénytel mindkét esetben stringnek adjuk meg.

Az [In2 "ú!", In2 King, In1 "Skócia"] kifejezés viszont nincsibjelezést eredményez, mert a 'b típusváltót nem lehet ugyanabban a kifejezésben egyszer így, mászor úgy lekötni.

### 9.4. A case-kifejezés

Szintaxisa a következő:

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

Az SML-értelemző – bárholt jobbra és fölöttiről lefelé haladva – megpróbálja E-t P1-re illeszteni, ha nem sikerült, P2-re stb. A case-kifejezés eredménye az E kifejezés illeszkedő első Pi mintához tartozó E1 kifejezés lesz. Például a Lady függvényt így is definíálhattuk volna:

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

Az a lehetséges lehet, hogy a fun függvénydefinícióban változatokat definiálhatunk, nem egyéb, mint rövidítés, szintaktikai *edészőszer*.

## 10. fejezet

### Részlegesen alkalmazható függvények

#### 10.1. Az fn jelölés

A 2.1.2 szakaszban már találkoztunk a (gyakran *lambdanak* ejtett) *fn* kulcsszóval. Névtelen függvény például *az fn x => E függvénykifejezés*, ahol *E*-nek (esetleg *x*-ről függ), kiértekelhető kifejezések kell lennie. Ha *x* által *E* pedig *'b* típusú érték, akkor a függvénykifejezés *'a -> 'b* típusú. Mivel ennek a függvénynek *nincs neve*, rekurzív függvény így minden nem hozható létre. Mintállal esetleg több változat is megadható:

```
fn p1 => E1 | p2 => E2 | ... | pn => En
Nézzük egy példát a függvénykifejezés alkalmazásáról!
```

A függvénykifejezést – precedenciaok 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-kalkultust örökölt fr-jelölések milyen kifejező ereje van! *fn*-jelöléssel színe az összes megszokott programozási jelölésnek milényen kifejező ereje van! *fn*-jelöléssel *edetísszer*.

A korábban láttott *lady* függvényt fr-jelöléssel is definíálhattuk volna:

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

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

#### 10.1.1. Függvény definíálása fun, val és val rec kulcsszóval

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

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

A fun kulcsszóval kezdődő függvénydefinició lehet rekurzív is:

```
(* replist(n, x) = n db x értékből álló lista
 replist : int * 'a -> 'a list
*)
fun replist (n, x) =
 if n = 0 then [] else x :: replist(n-1, x)
A val kulcsszóval kezdődő függvénydefinició csak akkor lehet rekurzív, ha ezet 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)
```

#### 10.2. Részlegesen alkalmazható függvények

Tudjuk, hogy az SML-hen egy függvénynek csak egyetlen argumentuma van, de ez egy pár, egy ennes, egy másik függvény sub. is lehet. Több argumentum függvényt olyan függvénytel mielőszintűtünk, amely függvényt ad eredményül.

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

Nézzük a következő függvénydefiniciókat:

```
(* 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, mindenkető a részlegesen alkalmazható prefix függvény definíciója. A függvény tipusát leíró *típuskifejezésű* kiolvasható, hogy ha a prefix függvényt string típusú

argumentumá alkalmazzuk, *függvényt*, ad eredményül, amely ugyancsak string típusú argumentuma alkalmazható string típusú értéket ad eredményül.

Egy részlegesen alkalmazható függvény alkalmazható csak az első, az első és a második stb. argumentumára. A részleges alkalmazás eredménye maga is *függvény*, *kétargumentumú függvényként* viselkedik. Nézzük néhány példát a használatára:

```

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

- val knightify = prefix "Sir"
- val dukeify = prefix "The Duke of"
- val lordify = prefix "Lord"
 > val prefix = fn : string -> (string -> string)

prefix fenti definíció nehezségek, az alábbi változat jóval olvashatóbb:
- fun prefix pre post = pre ^ post;
 > val prefix = fn : string -> (string -> string)

Természetesen a részlegesen alkalmazható függvények is lehetnek rekurzívak, pl.
(* replist n x = n db x értékkel 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 általánosítja a list típusú értéket ad eredményül. Gyűjtőargumentummal javíthatunk replist hatékonyságát:
```

```

fun replist n x =
 let fun rpl1 0 xs = xs
 | rpl1 n xs = rpl1 (n-1) (x :: xs)
 in rpl1 n []
 end
end
```

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

Mint tudjuk, az SML-értelmezők a kifejezéseket *balról jobbra haladva* értékelik ki. A függvényalkalmazás *erősen elő*, precondíciójára a lehető legnagyobb. A  $\rightarrow$  típusoperátor (a *laképzés* jele) *jelöl* *köt*, ezért például a  $\text{string} \rightarrow (\text{string} \rightarrow \text{string})$  típuskifejezés ekvivalens a  $\text{string} \rightarrow \text{string}$  típuskifejezettel (az SML-értelmezők a típuskifejezést redundáns zárójelökkel tüntetik ki a képernyőre).

A részleges *nem* alkalmazható függvényt *uncurried* függvénynek is nevezik. Ha egy részleges nem alkalmazható változatának a típusa  $'a \rightarrow ('b \rightarrow 'c)$ .

### 10.3. Függvény mint argumentum és mint eredmény

Nézzük a beszűró rendezést megalósító *inssort* függvényt!

(\* inssort LE ls = ls elemeinek az LE relació szerint rendezett lista)

```

inssort : ** vezesse le önállóan a függvény típusát! **
*)
fun inssort LE ls =
 let infix LE
 (* inssort(x, ys) = az ys eleméiből és az ys-be az LE relació szerint beszűrt x-ből álló, rendezett lista
 PRE: ys az LE relació szerint rendezve van
 *)
 ins : 'a * 'a list -> 'a list
 PRE: ys az LE relació szerint rendezett listája
 fun ins (x, []) = [x]
 | ins (x, y:ys) = if x LE y
 then x::y
 else y::ins(x, ys)
 sort : 'a list -> 'a list
 PRE: ys elemeinek LE színű rendezett listája
 fun sort [] = []
 | sort (x::xs) = ins(x, sort xs)
 in
 sort ls
 end
 sort végegny a rendezendő lista minden elemén, és ins segítségével egysével beilleszt az elemeket a helyükre. A nyugdíjában használt PRE szócska előtelített (precondition) jelő: az utána álló (logikai) kijelentések a függvény kiértékelés előtt teljesüdni kell ahhoz, hogy a függvény helyes értéket adjon eredményt. Az ls paraméter elhagyható a definícióból, elhagyásával mégsem javasoljuk, mert nélküle nehézből megérteni a függvény működését:
 fun inssort LE =
 let
 ...
 in
 sort
 end
 Mielőtt folytatná az olvasást, gyakorlásképpen vezesse le inssort típusát!
inssort : ('a * 'a -> bool) -> 'a list
inssort itt bemeneti változata generikus (általános), mert a rendezési relációit megalósító függvényt paraméterként adjuk át. Nezzük ki a példát inssort alkalmazására:
inssort op <= [5, 3, 7, 5, 9]
inssort op >= [5, 3, 7, 5, 9]
Mint tuduk, az op szócska és a minyeleti jel közötti szóköz elmaradhat. inssort újabb alkalmazásához új relációt definiálunk füzetek lexikografikus rendezésére:
(* leStrPair(s, t) = igaz, ha s lexikografikusan nem nagyobb t-nél
 leStrPair : (string * string) * (string * string) -> bool
*)
 fun leStrPair ((a, b), (c : string, d : string)) =
 a < c orelse (a = c andalso b <= d)
inssort leStrPair [(("Kovács", "Tibor"), ("Bihari", "Tamás"))]
```

```

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

(* recip r = az r valós szám reciproka
recip : real -> real
*)
val recip = secl 1.0 op/
val halve r = az r valós szám fele
halve : real -> real
*)
val halve = secr op/ 2.0

```

## Magasabb rendű függvények

### 11. fejezet

#### 11.1. Magasabb rendű függvények

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

##### 11.1.1. secl és secr

Címkákat hasznos, ha egy *infix* operátor egik operandusát rögzítjük, a másikat szabadon hagyjuk, például

```
(* "Sir" `~) ekvivalens a knightify függvényel,
(/ 2.0) olyan függvény, amely 2.0-vel oszt.
```

Sajnos, ezek az jelölések így nem használhatók az SML-nen, secl és secr segítségével azonban éppen ilyen függvényeket definíálhatunk. A nevet záró l, ill. r betű arra utal, hogy a bal (*left*), ill.

a jobb (*right*) operandust kijelöl le.

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

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

secl és secr paramétereinek nevét és sorrendjét úgy választottuk meg, hogy egységesen utaljanak a paraméterek szerepére és pozíciójára, másrészt tegyük lefelőve secl és secr részleges alkalmazását. Mielőtt folytatnánán az olvasást, vezessé le önmállóból a két függvény típusát!

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

Végül bemutatunk néhány példát a két függvény alkalmazására.

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

#### 11.1.2. Kombinátorok

A kombinátoroknak elsősorban *elő* szempontból nagy jelentősége (v.ő.  $\lambda$ -kalkulus).

##### 11.1.2.1. Két függvény kompozíciója

Két függvény kompozícióját általában a operátorral jelölik, mi az o betű foglal használati.

```
(* 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 a alkalmazására, írunk összegező függvényt a  $\sum_{i=0}^{m-1} f(i)$  kifejezés kisszámítására! A kifejezésben az m és az f ún. szabad változók, az i ún. kötött változó, a 0 és az 1 pedig állandók.

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

A sum segédfüggvény a hatékonysegöt javítja, mert *iteráció*: a z argumentumban gyűti az eredményt. Most alkalmazzuk a függvényt a  $\sum_{k=0}^{m-1} \sqrt{k}$  kifejezés kisszámítására!

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

Talán nem kell sokat bizonyni, hogy az sqrt o real kompozíció alkalmazásával a felirat kifejezés jobban kifejezi a lényeget, és olvashatóbb is, mint a summa(sqrt(real 10)) alak, többek között azért, mert jobban hasonlít a matematikai megszokott jelölésnövére.

<sup>1</sup> sqrt gyakran vagy a *7-4* szakaszban definiált sqrtot, vagy a Math könyvtárbeli Math.sqrt függvény használatát.

### II.1.2.2. Az S, a K és az I kombinátor

Az I kombinátor a közismert *identitásfüggvény*.

```
fun I x = x

A K kombinátor egy állandóra alkalmazva azt, konstansfüggvényt állít el: olyan függvényt, amelyet bármilyen argumentumra alkalmazunk is, minden ezt az állandót adjja eredményül.
```

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

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

summa (K 7) 5

Az S kombinátor neve: *általános kompozíció*. SML-definiciója:

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

A  $\lambda$ -kalkulus szerint minden függvény felirható *kizárolag* K és S alkalmazásával, változók nélküli! Ez K és S igazi jelentősége, gyakorlati szempontból nem olyan fontosak. Az érdekkesség kedvét megmutatjuk, milyen egyszerű I definíciója K-val és S-vel:

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

Nézzük egy példát S K K 7 egyszerűsítésére:

```
S K K 7 → K 7 (K 7) → 7
```

### II.1.3. map és filter

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

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

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

Lássunk néhány példát az alkalmazásukra!

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

Válaszoljon önmállan a következő kérdésekre: mi az alábbi függvénykifejezések típusa, és mi a kiterételestik eredménye, ha mindenkorral az [[“abc”, “def”], [“mnopqr”, “xyz”]] listára alkalmazzuk?

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

Két halmaz *metszele*(S  $\cap$  T) például így definíálható filter-rel (ss-ben S, ts-ben T elemet tároljuk):

```
(* inter(ss, ts) = az ss és ts halmazok metszete
inter : ''a list * ''a list -> ''a list
*)
fun inter (ss, ts) = filter (secr (op isMem) ts) ss
(*)
PRE: $\forall i, j \bullet i \neq j, s_i \in ss, s_j \in ts, s_i \neq s_j, i \bullet i \neq j, t_i \in T \bullet t_i \neq t_j$
```

Az isMem függvényt a 8 szakaszban *infis* operátorról definiáltuk.

### II.1.4. takewhile és dropwhile

Korábban take-kelet és drop-pal találkoztunk: *take* egy lista elejéről vett addott számú elemből, drop egy lista elejéről addott számú elem elhagyásával kész listát. Néha olyan függvényekre van szükség, amelyek egy lista elejéről vett, addott predikátnuot kielégítő elemeket listával adott predikátnuot kielégítő elemek elhagyásával készítenek. Ilyen függvényeket írnunk most takewhile, ill. dropwhile néven.

```
(* takewhile p xs = az xs elejéről vett, p-t kielégítő elemek listája
takeWhile : ('a -> bool) -> 'a list -> 'a list
*)
fun takewhile p [] = []
| takewhile p (x::xs) = if p x then x :: takewhile p xs else []
(* dropwhile p xs = xs elejéről a p-t kielégítő elemek elhagyásával
dropWhile : ('a -> bool) -> 'a list -> 'a list
*)
fun dropwhile p [] = []
| dropwhile p (x::xs) = if p x then dropwhile p xs else x :: xs
dropwhile-ban releges mintái is alkalmazhatunk:
fun dropwhile p [] = []
| dropwhile p (x::xs as x::xs) = if p x then dropwhile p xs else xs
```

### II.1.5. exists és forall

exists és forall a logikából jól ismert *kontinuátor* ( $\exists, \forall$ ) megalósítása 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 kielégítíti p-t
forall : ('a -> bool) -> 'a list -> bool
*)
fun forall p [] = true
| forall p (x::xs) = p x andalso forall p xs
```

A korábban már definált isMem függvényt újrafelülbíráljuk 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 segítsével definíálhatjuk a halmozok diszjunkt voltáit 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 elszárának megérteni. Próbálunk 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 valamely – e függvény
számára tűlső és töviseit – x értékkel. Ezt a függvényt a zárójelben belüli forall hívás az összes
számos xs-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 kiliső forall az
összes xs-beli értékkel meghívja ezt a függvényt, és akkor igaz az eredménye, ha minden ys-beli
érték, amely egyenlő lenne valamely xs-beli értékkel.
```

### 11.1.6. foldl és foldr

foldl balról jobbra (left to right), foldr jobbról balra (right to left) haladva egy *kétargumentumú prefix* függvényi (*pl. op<sup>r</sup>, op<sup>l</sup>*) alkalmaz egy lista minden elemére. A két függvény specifikációja *infix* operátorral ( $\oplus$ ) írja fel, mert *nfiz* jelössel könnyebb megérteni a működésüket. ( $\oplus$  tetszőleges infix operátor helyettesít.)

```
foldl op \oplus [x1, x2, ..., xn] = ((...((e \oplus x1) \oplus x2) $\cdot\cdot\cdot$ \oplus xn)
foldr op \oplus [x1, x2, ..., xn] = (x1 \oplus (x2 $\cdot\cdot\cdot$ \oplus (xn \oplus e)) $\cdot\cdot\cdot$)
```

Ilt e az ún. egységelem. Látható, hogy ha elvégzések kiejtését kifejezi, mindenket függvény egyszerűsítik az eredeti kifejezést. Kifejezéseket általában – de nem minden! – jobbról balra haladva működnek.

Associatív műveletek esetén azonban az *egységelem kölcsönös* függ a dolog: ha  $e \oplus x = x$ , az egységelem bal oldali, ha  $x \oplus e = x$ , az egységelem jobb oldali. Ha mindenki egyenlősen fennáll – és asszociatív egyszerűsítési török, ezért foldr-néha reducere néven definiálják.

Közismert, hogy az összeadás és a szorzás asszociatív műveletek, a kivonás és az osztás azonban nem, ezért nem minden, hogy foldl-t vagy foldr-t alkalmazzuk: *nemasszociatív* műveletek esetén ez a helyetől – egyszerűen egyszerűen műveletből beszélünk.

E kis példából az is látható, hogy a kivonásnak – és hozzá hasonlónak – jobb oldali egységeleme van.

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

```
foldl op \oplus e [] = e és foldr op \oplus e [] = e
Jól látszik, hogy e valóban az op \oplus művelet egységeleine, hiszen op \oplus -t az üres listára alkalmazza
et kapjuk eredményül. Most már definíálhatjuk foldl és foldr SML-változatát.2
```

```
(* foldl f e xs = az xs elemeire balról jobbra haladva alkalmazzott,
 kétoperandusú, e egységlemei 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
```

<sup>2</sup>Mindkettő belső függvény az SML-ben.

```
(* foldr f e xs = az xs elemeire jobbról balra haladva alkalmazzott,
 kétoperandusú, e egységlemei 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. Vagyük észre, hogy foldl *iteratív* függvény: második argumentumá, amely kezdetben az egységeleml, gyűjti össze a részeredményeket, és majd csak a lista külön-külön hajtja végre a kijelölt műveleteket. Sejthet, foldr a verbenben gyűjti a részeredményeket a két definíciót levezetni a tipust!

Számos függvény irányt fel foldl és foldr alkalmazásával, ha megadjuk a lista sziszédes elemein végrehajtandó műveletet és az egységeleml. Lássunk néhányat:

```
(* sum xs = xs elemeinek összege
 sum : int list -> int
*)
fun sum xs = foldl op+ 0 xs
(* prod xs = xs elemeinek szorzata
 prod : int list -> int
*)
fun prod xs = foldl op* 1 xs
(* flat xs = az xs részlistáinak konkatenálásával előálló lista
 flat : 'a list list -> 'a list
*)
fun flat xs = foldl op@ [] xs
(* length xs = az xs lista hossza
 length : 'a list -> int
*)
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
A length függvény egy iteratív változata (inc olyan kétargumentumú segédfüggvény, amelyik nem
használja a második argumentumát!):
```

```
(* append xs ys = az xs ys listákat előállító lista
 append : 'a list list -> 'a list
*)
fun append xs ys = foldr op@ [] ys xs
A beszűró rendezés egy újabb változata foldr-re:
```

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

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

11.1.7. További rekurzív függvények

Egy függvény n -edik hatványát így specifikálhatjuk: $f^n = f(\dots f(f(x))\dots)$, ha $n \geq 0$ (f n -szer ismétlőlik). Definíljunk SML-függvényt illeten ismélésük felirására.


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


Sok függvény fejezető ki repeat segítségével. Példák:


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

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


```

### 11.1.8. curry és uncurry

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

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

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

```

### 11.1.9. map újratelvezetése foldr-rel

map definíálásához mintául szolgálhat a listaelenek összegét képező sum függvény definíciója pl. foldr-rel:

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

```

Idézzük fel 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 kompozíciójuktat használni. Csaknugy op:: részlegesen nem alkalmazható, argumentumként pár változó függveny, az egy argumentumú f-fel pedig csak részlegesen alkalmazható változatát komponálhatjuk: (curry op :: o f). Igen ám, de foldr első argumentuma argumentumként pár változó, 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 szembe az uncurry(curry op :: o f) kifejezést (ahol f még lekötötten azonosító)! Argumentumána egy pár, e pár első taga valamilyen második tagja egy lista, az eredménye pedig egy ugyanilyen típusú lista. A függvény a pár első tagján elvégzi az f transzformációt, majd a kapott értéket a pár második tagjához fűzi:

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

```

Mit tagadjuk, uncurry(curry op :: o f) elég osztynácska kifejezés, ríjuk 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

```

kulcsszóval kezdődő kifejezések képesek kezelni, általános értelemben nem tekinthetők értéknék az SML-ben. Az `exn` típus „közvetti” a kivételesonagok és más SML-értékek között.

A kivételkezelés során az SML-értétek a kivételesonagokat az érték szerinti paraméteráradás szabályai szerint adjja tovább. Ha egy E kifejezés eredménye egy kivételesonag, akkor tetszőleges főre f(E) eredménye is egy kivételesonag, f(raise E) ekvivalens raise E-val (pl. raise (badvalue raise Failure)).

Tudjuk, hogy az SML minden kifejezést bárhová lefelé haladva értékel ki. Ezért ha E1 eredménye egy kivételesonag, akkor az (E1, E2) párban E2 kírótérére nem is kerül sor. Ha E2 eredménye a kivételesonag, E1-é pedig valamitlen más érték, akkor az (E1, E2) párból eredménye a kivételesonag lesz.

Az `if E then E1 else E2` feltétes kifejezésben nemcsak E1 és E2, hanem E kírótérének is lehet kivételesonag az eredménye. Ha a `let val P = E1 in E2 end` kifejezésben E1 eredménye egy kivételesonag, akkor az egész let-kifejezés eredménye is ez a kivételesonag.

### 12.2.1. Belső kivételek

Az SML legfontosabb belső kivételekonstruktoraikat az alábbi táblázatban soroljuk föl.

| Megnevezés | Művelet, amely a kivételek kiváltása                      |
|------------|-----------------------------------------------------------|
| Bind       |                                                           |
| Chr        | chr pred succ                                             |
| Div        | / div mod                                                 |
| Domain     |                                                           |
| Empty      | hd tl last                                                |
| Fail       | compile load loadOne                                      |
| Interrupt  |                                                           |
| Lo         |                                                           |
| 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           |

### 12.3. Kivétel feldolgozása a handle kulcsszóval

A kivétel feldolgozásaa 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, mintaháttérben továbbája az eredményt. De ha E kivételesonagot eredményez, akkor a tartalmát az SML-futtatórendszer megröbálja a megadott mintákra illeszténi. Ha az első illeszkedő minta a Pi ( $i = 1, 2, \dots, n$ ), akkor a kivételkezelő eredménye az Ei kifejezés eredménye lesz. Ha egyetlen minta sem illeszthető a kivételesonagra, akkor a kivételkezelő továbbasszolia a kivételesonagot az előző hivási szintre.

### 12.4. Néhány példa a kivételkezelésre

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

## 12. fejezet

### Kivételkezelés

**Kivételkek** (exception) nevezik a különleges elbánást igénylő eseteket: a különféle futási hibák (0-val való osztás, túlsordulás, lista kiürítése, nemlétező állomány megnyitása stb.) felépését, a programmegszakítást stb. A kivételkezeléshez harom szintaktikai elem – exception, raise, handle – jelentésével és használával kell megismernedünk.

Az SML-ben a kivétel a függvények mindenadig továbbhalaszolják az öket hívó függvényeknek, végső esetben az SML kererendszernek, amíg egy kivételkezelő fel nem isméri, 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.

### 12.1. Kivétel deklarálása az exception kulcsszóval

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

```
exception Failure
deklaracio a Failure kivételellandóval (különleges típuskonstruktőrökkel) bővíti az exn típusú értékek (a kivételekonstruktőrök) halmazt. A következő példa két kivétfüggvényet (különleges típuskonstruktorfüggvényet) bővíti a konstruktőrhöz:
```

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

FailedBecause típusa: string -> exn, BadValue-é int -> exn.

Kivétel tölélisan is lehet dekláráni, de nem célszerű, hiszen ha pl. ugyanazon a néven több kivétel is jelezhet a futtatónak, az nehezíti a liba lokalizálását, a program megerősítését.

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

### 12.2. Kivétel jelzése a raise kulcsszóval

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

Ha az E kifejezés kírójelése exn típusú értéket ad eredményül, akkor a raise E kifejezés az értéket tartalmazó kivételesonagot eredményez. Az ilyen kivételesonagot csak a handle

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

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

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

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

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

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

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

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

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

Az első példában a demo1 és demo3 is meglévő ható kivételkezelő alkalmazása nélküli, de mindenkor a program végrehajtása kivétel fellépésekor felbeszakad. Tegyük föl például, hogy a test30.sml állományban az alábbi program van:

```
fun test30 (x,s) = demo3(x,s);
- use "test30.sml";
[Opening file "test30.sml"]
> val test30 = fn : int * string -> int * string
> val it = (5,"five") : int * string
! Caught exception:
! Demo3 "Not five but three"
[Closing file "test30.sml"]
```

Anikor az SML-értelemező a use "test30.sml" függvényalkalmazás hatására ezt a programot hozzársa, a feldolgozás a test30(3, "three") kifejezés kiértékelése közben a kivétel fellépése miatt felbeszakad, az s név deklarárára sohasem kerül sor.

```

*)
 fun ins cmp (x, ys) =
 let infix cmp
 fun ins0 (y::ys) = if x cmp y then x::ys else y::ins0 ys
 | ins0 [] = [x]
 in
 ins0 ys
 end

```

Ezzel inssort egy újabb változat:

```

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

inssort eddig bennalévo változatban előbb elemre szedik a rendezendő listát, majd határoló
visszafele haladva, rendezés közben építik fel az újat. Jobbrekunroz és gyűjtőargumentumot használó
változatának kisséb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra
haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét a 13.3
szakaszban hasonlítjuk össze).

```

```

fun inssort2 cmp xs =
 (* sort xs zs = az xs már feldolgozott elemeinek a cmp
 * reláció szerint rendezett listája zs
 * sort : 'a list -> 'a list -> 'a list
*)
let fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
| sort [] zs = zs
in sort xs []
end

```

### 13.1. Beszűró rendezés

Az ins segédfüggvény az x elemet a megtérülő helyre rakja be az ys listában:

```

(* ins (x, ys) = az x értékkel a <= reláció szerint bővített ys
 ins : real * real list -> real list
 PRE: ys a <= reláció szerint rendezve van
*)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
| ins (x : real, []) = [x]

inssort-ral rekurzív rendezést a lista maradványát; végrehajtási ideje $O(n^2)$:

```

```

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

inssort az inssort generikus meghívása:

```

#### 13.1.1. Generikus meghívások

Ha a következő elemet a helyére rakt f függvényt paraméterként adjuk át, az inssort téteszleges
tipusú adatok rendezésére használható:

```

(* inssort f xs = az xs elemeinek az f segítségével rendezett listája
 inssort : ('a * 'b list -> 'b list) -> 'a list -> 'b list
*)
fun inssort f (x::xs) = f(x, inssort f xs)
| inssort [] = []

Még jobb, ha magát az ins függvényt tesszük generikussá:

```

```

(* ins cmp (x, ys) = az x értékkel a cmp reláció szerint bővített ys
 ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
 PRE: ys a cmp reláció szerint rendezve van

```

### 13.1.2. Beszűró rendezés foldr-rel és foldl-rel

A második argumentumunkat gyűjtőargumentumként használó foldl sokkal kisebb vermet használ,
mint foldr, ezért inssort2 hosszabb listák rendezésére alkalmas. A futási időkről a 13.1.3 szakaszban lesz szó.

```

fun inssort cmp = foldr (ins cmp) []
fun inssort2 cmp = foldl (ins cmp) []

```

### 13.1.3. A futási időök összehasonlítása

Véletlenszerűen előállított listák, illetve eredetileg éppen fordított sorrendű listák rendezéséhez
szükséges futási időt mérünk. Véletlen először2 hosszabb listák állít elő a Random könyvtárbeli rangeList
függvény. Növekvő sorrendű egyszírást állít elő a -- operátor:

```

infix --;
fun fm -- to =
 let fun upto to zs = if to < fm then zs else upto (to-1) (to:zs)
 in upto to []
end;

```

A futási idők mércélez 2000 elemet tartalmazó listákat állítunk el:

```

val xs = Random.rangeList (1, 100000) (2000, Random.newgen());
illetve
val xs = 1 -- 2000;
in
 val qs = qs xs
end;

A futási időket az alábbi programnal mérjük meg és írjuk ki:
app load ["Int", "Random", "Time", "Timer"];
let val starttime = Timer.startCPU();
val zs = insert op> xs
val {usrt=tim,...} = Timer.checkCPU();
val startime = "Integer sort with insert: " ~
 Int.toString(length xs) ~ " sec\n";
val starttime = Timer.startCPU();
val zs = insert2 op> xs
val {usrt=tim,...} = Timer.checkCPU();
val t2 = "Integer sort with insert2: " ~
 Int.toString(length xs) ~ " sec\n";
val t1 = Time.fmt 2 tim ~ " sec\n";
in
 print(t1 ~ t2)
end;

```

A futási időket az alábbi programmal mérjük meg és írjuk ki:

app load ["Int", "Random", "Time", "Timer"];
let val starttime = Timer.startCPU();
val zs = insert op> xs
val {usrt=tim,...} = Timer.checkCPU();
val startime = "Integer sort with insert: " ~
 Int.toString(length xs) ~ " sec\n";
val starttime = Timer.startCPU();
val zs = insert2 op> xs
val {usrt=tim,...} = Timer.checkCPU();
val t2 = "Integer sort with insert2: " ~
 Int.toString(length xs) ~ " sec\n";
val t1 = Time.fmt 2 tim ~ " sec\n";
in
 print(t1 ~ t2)

több mint 5-sig, a gyűjtőargumenntumos insert2-ben utóbbi változatával viszont csak 0.01-0.025ig tart (linux alatt, 233 MHz-es Pentium processzor). Hatalmas a nyereség, nem beszélve arról, hogy gyűjtőargumenntum nélkül a verem hamar megnelik! De eltérniük különbség a két félé változat között, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk: a futási idő bármielyik változat esetén kb. 2 s.

## 13.2. Gyorsrendezés

A gyorsrendezés a növekvő sorrendben rendezendő sorozatot három részre osztja: az  $m$  mediárra, a mediánnál nem nagyobb, ill. nagyobb elemekre:

$$\boxed{\quad} \leq \boxed{\quad} m \boxed{\quad} < \boxed{\quad}$$

Mediánnak az adott lépéshben rendezendő lista fejét választjuk.

```

(* quicksort cmp xs = az xs elemeinek cmp szerint rendezett listája
quicksort : (a * `a -> bool) -> a list -> a list
*)
fun quicksort cmp xs =
 let (* qs : `a list -> `a list
 *)
 fun qs (m:ys) = partition ([], [], m, ys)
 | qs ys = ys
 (* partition : `a list * `a list * `a * `a list -> `a list
 *)
 and partition (ls, rs, m, xs) =
 if cmp(x, m)
 then partition (x::ls, rs, m, xs)
 else partition (ls, x::rs, m, xs)
 in
 qs
 end

```

```

 | partition (ls, rs, m, []) = qs ls @ (m::qs rs)
 in
 qs xs
 end

partition iteráv függvény, amely két eredménylistát épít fel, ls-t és rs-t.
quicksort a 2000 elemet tartalmazó, eredetileg fordított sorrendű listát majdnem 14 s alatt
rendezi. Az ugyanilyen hosszú, de véletlenszerűen előállított lista rendezéséhez ugyanakkor csak
0.1 s-ra van szüksége.
A @ művelet kiküszöbölikő újabb segédfüggvényt a bevezetésével. Az algoritmus hatékonyiséga
tovább javítható pl. úgy, hogy az m értékű elemeket egy hármasik listában gyűjti juk, amikor a
listát két részre bontjuk.
```

## 13.3. Összefeszítő rendezés

Az összefeszítő rendezéshez szükséges link van egy olyan függvényre, amely két listát növekvő sorrendben egészít:

```

(* merge(xs, ys) = xs és ys elemeinek <= szerint egészített listája
merge : int list * int list -> int list
*)
fun merge (xs::xs, y::ys) = if x <= y
 then xs::merge(xs, y::ys)
 else y::merge(xs::xs, ys)
| merge ([] , ys) = ys
| merge (xs, []) = xs

```

vagy ún. *réteges minna* (...as...) alkalmazásával:

```

fun merge (xs as x::xs, ys as y::ys) = if x <= y
 then x::merge(xs, y::ys)
 else y::merge(xs::xs, ys)
...

```

Hatékonyágrólást okoz, hogy a részrendezéket itt is a veremben tároljuk. Sajnos, ez nem
lehet segíteni, mert iteratív megoldás esetén vissza kellene fordítani az eredménytől kapott listát.

### 13.3.1. Fölülről lefelé haladó összefeszítő rendezés

A fölülről lefelé haladó (*top-down*) összefeszítő rendezés akkor hatékony, ha közelazonos hosszúságú
az a két lista, amelyekre a rendezendő listát szétszedjük. (A tmsort név elején a t betű utal a
*top-down* rendezésre.)

```

(* tmsort xs = az xs elemeinek a <= reláció szerint rendezett listája
tmsort : int list -> int list
*)
fun tmsort xs = let val h = length xs
 fun tmsort xs = let val k = h div 2
 in
 if h > 1
 then merge(tmsort(take(xs, k)), tmsort(drop(xs, k)))
 else xs
 end

```

A legrosszabb esetben  $O(n \cdot \log n)$  lépésre van szükség. A módszer egyszerű és elég gyors.

### 13.3.2. Alulról fölfelé haladó összefésülő rendezés

Az alulról fölfelé haladó (*bottom-up*) összefésülő rendezés leggyorsabb változata az eredeti *l* hosszságú listát *l* darab egyműemelő lista bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő. A megoldás egyszerű, de pazarló.

R. O’Keefe algoritmusá (1982) lépései futtatja össze az egymára hosszú részlistákat, de csak az utolsó lépésben rendeli az összeset. Az alábbi példában az összefuttatott részlistákat aláháttal jelölik:

```
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
```

Vagyis az algoritmus először az A-t futtatja össze a B-val, majd a C-t a D-val, ezt követően pedig az AB-t a CD-val, hiszen most már ezek hossza is egyforma. Ezután E és F, G és H, EF és GH, majd ABCD és EFGH összefuttatása következik s.t.

Most bmsort néven definíljuk az összefésülő rendezés alulról fölfelé haladó változatát, amely a quicksort-tal kb. azonos idő alatt rendez (a név elég betűje, b utal a *bottom-up* rendezésre):

```
(* bmsort xs = az xs elemeinek a <= reláció szerint rendezett listája
```

```
bmsort : int list -> int list
```

```
fun bmsort xs = sorting(xs, [], 0)
```

az *xs* a már rendezett részlisták listája. Ennek első argumentuma a rendezendő lista, második argumentumában a már rendezett részlistákat gyűjti (kezdőérte []), harmadik argumentuma pedig az adott elemet összefuttatandó elem sorszáma (kezdetben 0).

Ha a rendezendő lista még nem fogott el, soron következő eleméből sorting egyelemű listát (*[x]*) képez, és ezt a már rendezett részlisták listájába (lss) elé fizet megújva a *mergepairs* segédfüggvényt. *mergepairs*, annint látom fogjuk, az argumentumunk általadott lista két, azonos hosszságú bal oldali részlistáját fűzi egyre, feltéve persze, hogy vannak ilyenek. *k* az építen átadott elem sorszáma. Ha a rendezendő lista kiürül, sorting a készintű lista egyellen elemét, a rendezett listát adja eredményül.

```
(* sorting(xs, lss, k) = a még rendezetlen xs lista elemeit berakja
```

```
a k elemet tartalmazó, már rendezett lss listába
```

```
sorting : int list * int list -> int list
```

```
PRE: k >= 0
```

```
*)
fun sorting (x::xs, lss, k) = sorting(xs, mergepairs([x]:lss, k+1), k+1)
| sorting ([] , lss, k) = hd(mergepairs(lss, 0))
```

*mergepairs* egyellen listában gyűjti a már összefuttatott részlistákat. Nem a listák hosszát hasonlítja össze, hanem az épén átadtott elem k sorzámból dönti el, hogy mit kell csinálni a következő részlistával.

```
(* mergepairs(lss, n) = az n elemet tartalmazó, már rendezett lss lista
első két részlistáját, ha egymára a hosszuk,
```

```
összefuttatja
```

```
mergepairs : int list list -> int list list
PRE: n >= 0
```

\*)  
 fun mergepairs (lss as lss1::lss2::lss, n) =
 (\* legalább két elemű a lista \*)
 if n mod 2 = 1
 then lss
 else mergepairs(merge(lss1, lss2)::lss, n div 2)
 | mergepairs (lss, \_) = lss (\* egyelemű a lista \*)

Ha n páratlan, mergepairs a listát változtatás nélkül adja vissza, ha pedig páros, akkor az lss lista elején álló két, egyformára hosszú listát egyetlen rendezett listára futtatja össze. n=0-ra mergepairs az összes listák olyan listáját olyan listára futtatja össze, amelynek egyetlen eleme maga is lss.

A függvények működését az alábbi példán követhetjük. A kezdő hívás legyen  
 bmsort [1,2,3,4,5,6,7,8,9] =  
 sorting ([1,2,3,4,5,6,7,8,9], [], 0)

Amig sorting első argumentuma a nem üres x :: xs lista, sorting saját magát hívja meg. A rekurzív hívás első argumentuma a lépésekben egyre rövidülő xs lista, harmadik argumentuma (k+1) amián feloldogozott listaenek száma, második argumentuma pedig a *mergepairs* ([x]:lss, k+1) függvény alkalmazás eredménye, ahol kezdetben lss = [].

A következő táblázatos elhendezés *mergepairs* mindenkit argumentumát, valamint a rekurzív sorting hívás itt j-vel jelölt harmadik argumentumát, k+1-et és hármas számként k+1 mutatja lépésről lépésre. (Né fedejni, hogy mergepairs-nek listák listájára az első argumentumra.) A sorting függvény hívja *mergepairs*-t azoknak a soroknak, amelyekben a j-ijétet levez föl, a többi helyen *mergepairs* hívása rekurzív. A táblázat ujolsz oszlopja a vonalközö magyarázatra hivatkozik.

Vegyük észre, hogy kapcsolat van az lss első eleme utáni listaelemek hossza és a k bitje között! Ha k valamelyik bitje 1, akkor (balról jobbra haladva) az 11:ss megfelelő listaelemének a hossza az adott bit helyiértékkel egyenlő. A 0-értékű biteknek megfelelő listaelemek „hiányoznak” lss-ban.

|                                                         |    |   |      |
|---------------------------------------------------------|----|---|------|
| lss                                                     | n  | j | k    |
| [[]]                                                    | 1  | 1 | 0    |
| [[],[]]                                                 | 2  | 2 | 1    |
| [[],[],[]]                                              | 1  |   | m3   |
| [[],[],[],[]]                                           | 3  | 3 | 0    |
| [[],[],[],[],[]]                                        | 4  | 4 | 11   |
| [[],[],[],[],[],[]]                                     | 2  |   | m2   |
| [[],[],[],[],[],[],[]]                                  | 1  |   | m3   |
| [[],[],[],[],[],[],[],[]]                               | 5  | 5 | 100  |
| [[],[],[],[],[],[],[],[],[]]                            | 6  | 6 | 101  |
| [[],[],[],[],[],[],[],[],[],[]]                         | 3  |   | m3   |
| [[],[],[],[],[],[],[],[],[],[],[]]                      | 7  | 7 | 110  |
| [[],[],[],[],[],[],[],[],[],[],[],[]]                   | 8  | 8 | 111  |
| [[],[],[],[],[],[],[],[],[],[],[],[],[]]                | 4  |   | m2   |
| [[],[],[],[],[],[],[],[],[],[],[],[],[],[]]             | 2  |   | m2   |
| [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]          | 1  |   | m3   |
| [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]       | 9  | 9 | 1000 |
| [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]    | 0  |   | m4   |
| [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]] | 11 |   |      |

### Megjegyzések.

- m1: Az argumentumként átadott listának egyetlen eleme van (ez maga is egy lista), ezért az argumentumot `mergepairs` második klózva változtatás nélkül visszaadja az öt hívó `sorting-nak`.
- m2: n páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme egyforma hosszú lista, amelyeket `merge` egyetlen rendezett listává futtat össze, majd az eredménytől `mergepairs` első klóz meghívja saját magát.
- m3: n páratlan, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú lista, ezért az argumentumot `mergepairs` első klózra változtatás nélkül visszaadja az öt hívó `sorting-nak`.
- m4: n=0, az összes listák listáját olyan listává kell összefuttatni, amelynek egyetlen lista az eleme.
- A 2000 elemből álló listákat `bmsort` kevesebb mint 0.1 s alatt rendez minden eredetileg fordított sorrendű, mint véletlenszerűen előállított listák esetén.

### 13.4. Simarendezés

A simarendezés (*smooth sort*) végrehajtási ideje  $O(n)$ , azaz árányos az elemek számával, ha a benne lévő lista csakis rendezi van, és a legrosszabb esetben is legfeljebb  $O(n \cdot \log n)$ .

Az applikatív simarendezés O'Keefe algoritmusa O'Keefe alól fölfelé haladó rendezéséhez hasonló, de nem egyszerű listával, hanem növekvő futamokat (*run*) állít el.

Ha a futamok száma  $n$ -tel fogynak, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje  $O(n)$ .

```
(* nextrun(run, xs) = ????
nextrun : int list * int list * int list
*)
fun nextrun (run, x::xs) =
 if x < hd run
 then (rev run, x::xs)
 else nextrun(x::run, xs)
| nextrun (run, []) = (rev run, [])

nextrun eredménye egy párt. A pár első tagja a futam (egy növekvő szánsorozat), a második tagja pedig a rendezendő lista maradéka. Mivel a futam csökkenő sorrendben bővíti, kilépéskor a futamot meg kell fordítani.
```

`smsorting` a futamokat ismétlen előállítája és összefuttatja:

```
(* smsorting (xs, lss, k) = ???
smsorting : int list * int list list * int -> int list
*)
fun smsorting (x::xs, lss, k) =
 let val (run, tail) = nextrun(lx, xs)
 in
 smsorting(tail, mergepairs(run::lss, k+1), k+1)
 end
| smsorting ([] , lss, k) = hd(mergepairs(lss, 0))

(* smsort xs = az xs elemeinek a <= reláció szerint rendezett listája
smsort : int list -> int list
```

\*)  
 fun smsort xs = smsorting(xs, [], 0);  
 A simarendezés egy változata sort néven megalátható a Listsort könyvtárban. A függvény specifikációja a következő:  
 sort ord  $\lambda$  =  $\lambda$  xs  $\lambda$  ord szerint nem csökkenő sorrendben  
 (Richard O'Keefe applikatív simarendezése)  
 val sort : ('a \* 'a -> order) -> 'a list -> 'a list  
 Az ord olyan függvény, amelynek egy párt az argumentuma, és e pár két elemének az összehasonításával kapott order típusú érték az eredménye:  
 datatype order = LESS | EQUAL | GREATER  
 Az order típus a General könyvtár deklarálja. compare néven többek között a Char, az Int, a Real, a String, a Word és a Word8 könyvtában található olyan függvény, amely sort első argumentumaként használhat. A 2000 elemből álló listákat Listsort.sort Int.compare kevesebb mint 0.1 s alatt rendez minden eredetileg fordított sorrendű, mint véletlenszerűen előállított listák esetén.

## 14. fejezet

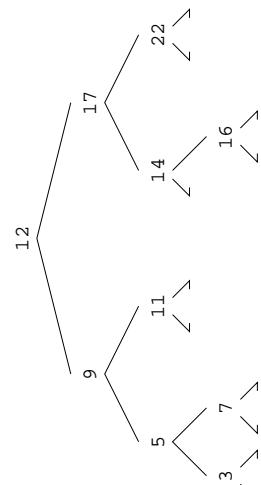
### Bináris fák

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

Eloször olyan bináris fát dekláralunk, amelynek a levelei üresek, a csomópontjaiban pedig előbb a bal részfa, majd az 'a' típusú érték, és végül a jobb részfa van:

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

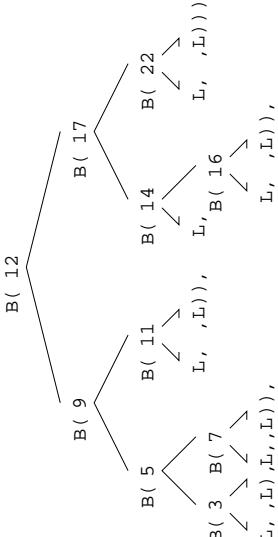
Tekintsük például az alábbi fát:



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

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

A fastruktúra szöveges leírását megkönyűítő, ha az ábrába beijlik a megfelelő adatkonstruktorokat:



Ezt bizony egleg nehez átlátni! A leírás attékinthetőbbé 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);
```

Ternézesítésen másfélle fastruktúrákat is deklarálhattunk, pl. kezdhettük az 'a' típusú értékkel, majd folytathattuk előbb a bal, azután a jobb részfa megragadásával. Felhasználhatjuk a levelet is értékek tárolására, vagy előiránytuk, hogy csak a leve�ben lehet érték szb. A

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

deklaráció például abban különbözik a korábban már látott

deklaráció 'a tree = L | B of 'a tree \* 'a \* 'a tree

deklarációtól, hogy a bináris fa leveleiben is tárolunk értéket, az értéket nem tároló üres csomókot pedig Evel jelöljük.

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

```
datatype 'a tree = B of 'a badtree * 'a * 'a badtree
```

```
datatype 'a tree = L of 'a badtree | B of 'a badtree * 'a * 'a badtree
```

### 14.1. Egyeszerű műveletek bináris fákon

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

```
datatype 'a tree = L | N of 'a * 'a tree * 'a tree
nodes egy fa csomóponjait számítja meg. Ehhez hasonlóan számolhatók meg a fa levelei.
```

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

Jobbkiszűrő változata jobb hatékonyságú.

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

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

depth0 jöbbkiszűrő változata:

```
(* depth0 f = az f fa mélysége
depth0 : 'a tree -> int
*)
fun depth0 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
```

depth egy fa mélységét (más szóhasználattal a magasságát) határozza meg. A fa gyökérrelől a leveleihöz vezető úton az élek számát (az ut hosszát) az adott levél szintjének is nevezik. A szintek között a legnagyobbat a fa mélységének hívjuk.

```
(* depth0 f = az f fa mélysége
depth0 : 'a tree -> int
*)
fun depth0 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 2n-ig beszánzza. Egy
```

teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levélnek ugyanezz a színe.

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

reflect a fát a függeléges tengelyre mentén tükrözi.

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

## 14.2. Lista előállítása bináris fa elemeiből

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

preorder először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát. inorder

először bejárja a bal részfát, majd kiveszi az értéket, és végül bejárja a jobb részfát. postorder

először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.

Az alábbi megalapítások egyszerűek, érthatók, de nem elég hatékonyak (a @ operátor használatára 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]
(* gyűjtőargumentum használata miatt nehezebbén érhetőek, de halékonyabbak a jobbnelkuszív változatok. (A függvények „elhagyásával” most eltekintünk.)
*)
fun preord(f, vs) = az f fa elemeinek postorder sorrendű listája
| preord (N(v,t1,t2)) = postorder t1 @ postorder t2 @ [v]
A gyűjtőargumentum használata miatt nehezebbén érhetőek, de halékonyabbak a jobbnelkuszív változatok. (A függvények „elhagyásával” most eltekintünk.)
```

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

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

### 14.3. Bináris fa előállítása lista elemeiből

Lista *kiegyensúlyozott binánris formájában* alakítanak a következő függvények; a különböző közöttük most is a bejárási sorrendben van.

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

 (* take'ndrop (k, xs) = olyan párr, amelynek első tagja xs első k db eleme,
 vége a lista első felén. Irunk take'ndrop néven olyan függvényt, amelynek egy egészben (k) és
 egy listából (xs) 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.
 *)
 fun take'ndrop (k, xs) =
 let fun td (0, xs, ts) = (rev ts, xs)
 | td (k, xs, ts) = td (k-1, xs, x::ts)
 | td (_, [], ts) = (rev ts, [])
 in
 td (k, xs, [])
 end
```

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

```
(* take'ndrop egy párt ad eredményül, ezért balpreorder-t módosítani kell.
 take'ndrop xs = az xs lista elemeiből álló, preorder bejárású,
 kiegyensúlyozott fa
 balpreorder: 'a list -> 'a tree
 *)
 fun balpreorder (x :: xs) =
 let val k = length xs div 2
 in N(x, balpreorder ts, balpreorder ds)
 end
 | balpreorder [] = L
```

Hatékonytáblázat okoz az is, hogy balpreorder minden meghívásakor kiszámítja xs aktuális hosszát. Ennek elkerülésére balpreorder-t egy segédfüggvényel egészítjük ki, amely k-t paramétert kapja.

```
fun balpreorder xs =
 let fun bpo (x:xs, k) =
 let val (ts, ds) = take'ndrop (k, xs)
 val k' = (k - 1) div 2
 in N(x, bpo(ts, k), bpo(ds, k))
 end
 in bpo ([], 0) = L
 in bpo(xs, (length xs - 1) div 2)
end
```

balinorder take'ndrop-pal való definíálását gyakorló feladatként az olvasóra bízzuk.

```
(* balinorder xs = az xs lista elemeiből álló, inorder bejárású,
 balinorder: 'a list -> 'a tree
 *)
 fun balinorder (x :: xs) =
 fun kiegyensúlyozott fa
 in N(y, balinorder(take(k, xs))), balinorder ys
 end
 | balinorder [] = L

 (* balinorder (x :: xs as x :: xs) =
 let val k = length xs div 2
 in N(y, drop(k, xs)) = drop(k, xs)
 in N(y, balinorder(take(k, xs))), balinorder ys
 end
 | balinorder [] = L

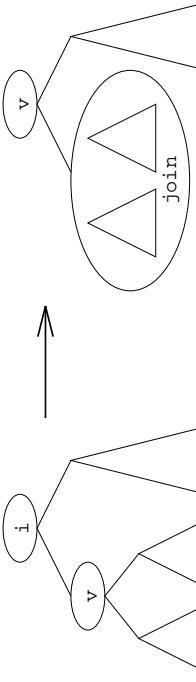
 (* balpostorder xs = az xs lista elemeiből álló, postorder bejárású,
 kiegyensúlyozott fa
 balpostorder: 'a list -> 'a tree
 *)
 fun balpostorder (y :: ys) =
 fun kiegyensúlyozott fa
 in N(y, balinorder(rev xs))
 end
 | balpostorder [] = L
```

```
(* balpostorder xs = az xs lista elemeiből álló, postorder bejárású,
 balpostorder: 'a list -> 'a tree
 *)
 fun balpostorder (y :: ys) =
 fun kiegyensúlyozott fa
 in N(y, balinorder(rev xs))
 end
 | balpostorder [] = L
```

**14.4. Elemtörlesé bináris fába**

Bináris fában adott értékű elemet rekurzív módszerrel *megkeresni* egyszerű feladat. Új elemet beszámít sem lehet: rekurzív módszerrel keressük egy levélét, és ennek helyére berakjuk az új értéket. Ha a fa rendeze van, ügyelünk kell arra, hogy a rendezettség megmaradjon.

Bináris fárból adott értékű elemet vagy elemeket rekurzív módszerrel kihordhatunk valamivel nehezebb: ha a törleendő érték az éppen vizsgált részfa gyökérérében van, a két részne széteső fa részfáit valamilyen módon *egyesíteni* kell, minután a törést a két részfán már vegrehajtottuk.



A vázolt műveletre mutat példát az alábbi remove függvény: rendezetten bináris fából törli az i. értékű és tékű elem összes előfordulását. A join segédfüggvényel egyesítjük a fölés hatására létrejövő két részfát, mígpedig úgy, hogy a bal fát lebonjuk, és közben az elemeit berakjuk a jobb fába.

```
(* bupdate(f, (b,y)) = az f fa, a b kulcszhou 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)

(* join(b, j) = a b és a j fák egységesével 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 törljük az adott értékű elemet. Ez a feladatot gyakorlásként az olvasóra bízzuk.
```

## 14.5. Bináris keresőfák

Ebben a szakaszban *bináris keresőfákon* alkalmazható műveleteket definiálunk. Rendszerint adott kulcsú elemet keresünk, ehhez értékeket kell összehasonlítanunk egymással; a keresett kulcsnak telált *egyenlőségi típusának* kell lennie. A példákhán a string típus használjuk, de a típus természetesen tetszőleges más egyenlőségi típus is lehet. Szebb lenne, ha *generikus függvényeket* írnánk; ez a feladatot gyakorlásképpen megihagyunk az olvasónak. A függvények *kivételek helyezetet* jeleznek, ha a keresett kulcsú elem nincs a keresőfában.

exception Bsearch of string

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

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

A binsert függvény egy új kulcsú elemet rakt be egy rendezett bináris fába, ha még nincs benne:

```
(* binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa
binsert : (string * `a) tree * (string * `a) -> (string * `a) tree
*)
fun binsert (L, (b,y)) = N((b,y), L, L)
| binsert (N((a,x), t1, t2), (b,y)) =
 if b < a then N((a,x), binsert(t1, (b,y)), t2)
 else if a < b then N((a,x), t1, binsert(t2, (b,y)))
 else (* a=b *) raise Bsearch("INSERT: " ~ b)
```

A bupdate függvény megfelelő kulcsú elembe új értéket ír be egy rendezett bináris fában:

eredményét  $xq$ -val jelöliük, akkor  $\text{consq}(x, E)$  kiértékelése a fenti definíció szerint  $\text{Cons}(x, \text{fn } () \Rightarrow xq \text{ függvény } \text{fn } () \Rightarrow xq)$ -t eredményez. A  $\text{consq}$ -beli  $\text{fn } () \Rightarrow xq$  függvény *nen késlelteti* a farok (a példában  $E$ ) kiértékelését  $\text{consq}$  alkalmazásakor. Az SML-ben a lista kiértékelés érdekeben a hivások is a  $\text{Cons}(x, \text{fn } () \Rightarrow E)$  alakot kell használnunk,  $\text{consq}(x, E)$  nem jó. Az  $\text{explicit fn } () \Rightarrow E$  késlelteti a kiértékelést, és ezzel szükséges szerinti hivatalosítási valósítás meg.

Példaként a korábban megnézett from es take függvények lista változatait mutatjuk be. A  $\text{fromq}_k$  sorozat egészek k-tól induló végében sorozata:

```

- fun fromq _k = Cons(_k, fn () => fromq(_k+1));
 > val fromq = fn : int -> int seq;
 - head (fromq 1);
 > val it = 1;
 - fromq 1;
 > val it = Cons (1, fn) : int seq;
 - tail it;
 > val it = Cons (2, fn) : int seq;
 - tail it;
 > val it = Cons (3, fn) : int seq;
 > val takeq(n, xq) az xq sorozat elsőn eleméből képzett listát adjja vissza:
 - fun takeq (0, xq) = []
 | takeq (n, Cons (x, xf)) = x :: takeq(n-1, xf());
 - tail it;
 > val takeq = fn : int * `a seq -> `a list

```

Nézzük egy példát takeq egyszerűsítésére!

```

takeq(2, fromq 30) ->
takeq(2, Cons(30, fn () => fromq(30+1))) ->
30::takeq(1, fromq(30+1)) ->
30::takeq(1, Cons(31, fn () => fromq(31+1))) ->
30::31::takeq(0, fromq(31+1)) ->
30::31::takeq(0, Cons(32, fn () => fromq(32+1))) ->
30::31::[] -> [30::31]

```

A 32-t az SML ugyan kiszámítja, de sohasem használja fel.  
Az `a seq típus nem egész és lista kiértékelési egy neműres sorozat fejét a rendszer minden feldolgozza, és ezt az SML-ben csak körülmenyes lehet elkerülni.<sup>2</sup>

## 15.1. Elémi feldolgozási műveletek sorozatokkal

A kiszámíthatóság érdekelben egy függvény eredményének telzőleges, véges része az argumentum véges részéről független csak. Amikor az eredményre szükség van, akkor ez az igény vályú ki az argumentum feldolgozását.

Nézzük egy példát, amelyben egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény faraka - egy függvény - alkalmazza a squareq függvényt az argumentum fájára.

```

- fun squareq Nil: int seq = Nil
 | squareq (Cons (x, xf)) = Cons(x * x, fn () => squareq(xf()));
 - squareq (fromq 1);
 > val it = Cons(1, fn) : int seq
 - takeq(10, it);

```

<sup>2</sup>Lásd L.C. Paulson: SML for the Working Programmer, Cambridge Press, 1991, 168. o.

## 15. fejezet

### Lusta lista

Az SML alapvetően mohó kiértékelésű, de lehet vele *lusta lista* tételni: a lista *farka* függvény, ezáltal *készleltetjük* a kiértékelését. Ily módon *végén* lista *elemeinek* összegét, nem kereshetjük meg benne a *legkisebbet*, nem előnyös tulajdonságai mellett hátrányaiak, veszélyeiak is vannak, pl.

- egy lista lista *bármely részlet* megjeleníthetjük, de *sóhasem az egészet*;
- két lista lista elemeiből páronként képezhetünk egy harmadikat, de például *nem számíthatjuk ki* egy lista lista *elemeknek összegét*, nem kereshetjük meg benne a *legkisebbet*, nem fordíthatjuk meg az *elemek sorrendjét*;
- úgy kell rekurziót definíálnunk, hogy *nincs alapelvej*;
- egy program befejeződése helyett csak azt igazolhatunk, hogy az eredmény *tetszőleges véges része véges idő után elérhető*.

Az SML-ben a lista listákat kezelő függvények bonyolultabbak, mint egy lista kiértékelést alkalmazó nyelvben, mert a lista faríkában *explicit* hivatalozunk kell a függvényre.

A lista listáit *sorozatnak* (sequence) fogunk nevezni, és a *seq* típusoperátor használjuk a létrehozására.<sup>1</sup>

datatype `a seq = Nil | Cons of `a \* (unit -> `a seq)

Egy sorozat fejét, ill. farkát adják eredményül az alábbi *head*, ill. *tail* függvények ; mindenkitől abortál, ha üres sorozatra alkalmazzák.

```

- fun head (Cons(x, _)) = x;
 > val head = fn : `a seq -> `a

```

A sorozat farika unit -> `a seq típusú függvény, erre illesztjük az xf minitát; tail törszében xf-et a () argumentumra kell alkalmazni:

```

- fun tail (Cons(_, xf)) = xf();
 > val tail = fn : `a seq -> `a seq

```

consq(x, x)-et berakja az xq sorozatba:

```

- fun consq (x, xq) = Cons (x, fn () => xq);
 > val consq = fn : `a * `a seq -> `a seq

```

Ha a consq függvényt alkalmazunk, mondtuk, az (x,E) argumentumra, a consq(x, E) kifejezést az SML *nem lustan* értékelni ki, hiszen az SML alapvetően minden lista *nil* és a lusta lista *Nil* konstruktora nem azonosak. A unit típus, mint tudjuk, a típusműveletek *egységelem*; egyetlen eleméről jelölik.

```

> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] : int list
> fun addq (Cons (x, xf), Cons(y, yt)) =
 Cons(x+y, fn () => addq(xf(), yt()));
| addq : int seq = Nil;
> val addq = fn : (int seq * int seq) -> int seq
- addq(f,fromq 1000, squareq(fromq 1));
> val it = Cons(1001, fn) : int seq
- takeq (5, it);
> val it = [1001, 1005, 1011, 1019, 1029] : int list

```

Az appendq függvény addig nem nyúl yq-hoz, amíg xq ki nem ürül - vagyis csak akkor nyúl hozzá, ha xq véges. Véges sorozatot consq-val készíthetünk. Most érhetjük meg, hogy miért kellett a típusdefinícióban a Nil konstruktőr állandó definiálni.

```

- fun appendq (Cons (x, xf), yq) = Cons(x, fn () => appendq (xf(), yq))
| appendq (Nil, yq) = yq;
> val appendq = fn : 'a seq * 'a seq -> 'a seq
- val finited = consq(25, consq(10, Nil));
> val it = Cons(25, fn) : int seq
- appendq(finiteq, fromq 1526);
> val it = Cons(25, fn) : int seq
- takeq (4, it);
> val it = [25, 10, 1526, 1527] : int list

```

## 15.2. Magasabb rendű függvények sorozatokra

Két jó ismert magasabb rendű függvény - map és filter - lusztá változatát definíáltuk ebben a szakaszban.

```

- fun mapq f (Cons (x, xf)) = Cons(f x, fn () => mapq f (xf()))
| mapq f Nil = Nil;
> val mapq = fn : ('a -> 'b) -> 'a seq -> 'b seq
- fun filterq p (Cons (x, xf)) = if P x
then Cons(x, fn () => filterq p (xf()))
else filterq p (xf());
| filterq p Nil = Nil;
> val filterq = fn : ('a -> bool) -> 'a seq -> 'a seq

```

Az előző szakaszban definált squareq sokkal egyszerűbben definíálható mapq-val; f-ként egészek esetén sq 'i-t, valósak esetén sq 'r-et kell használni, pl.!!!!!!

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

```

- filterq (fn n => n mod 10 = 7) (fromq 50);
> val it = Cons(57, fn) : int seq
- takeq(8, it);
> val it = [57, 67, 77, 87, 97, 107, 117, 127] : int list

```

Az ugyancsak magasabbrendű iterateq függvény a fromq egyáltalánosítása, az  $[e, f(x), f(f(x)), \dots, f^k(x), \dots]$  sorozatot állítja elő (v.ő. a korábban definált repeat-tel).

```

- fun iterateq f x = Cons(x, fn () => iterateq f (f x));
> val iterateq = fn : ('a -> 'a) -> 'a -> 'a seq
- iterateq (seqr op / 2.0) 1.0;
> val it = Cons (1.0, fn) : real seq
- takeq (5, it);
> val it = [1.0, 0.5, 0.25, 0.125, 0.0625] : real list

```

fromq-t iterateq-val úgy kaphatjuk meg, hogy f-kent a succ függvényt alkalmazzuk:

- fun succ k = k + 1;
- val fromq = iterateq succ

## 15.3. Néhány összetett példa

### 15.3.1. Álvéletlen-számok

A hagyományos álvéletlenszám-generátorok olyan eljárások, amelyek egy változóban tárolják a seed (mag) értéket - ebből állítják el a következő hivatalnál a következő álvéletlenszámot. Ha sorozathatóan valósítjuk meg az álvéletlenszámokat, akkor a következő álvéletlenszám csak szűkség esetén áll elő.

```

- local val a = 16807.0 and m = 2147483647.0
 fun nextrandom seed =
 let val t = a * seed
 in t - real(floor(t/m)) * m
 end
 in
 fun randseq s = mapq (seqr op / m) (iterateq nextrandom (real s))
 end;
> val randseq = fn : int -> real seq

```

Ha nextrandom-ot 1.0 és 2147483647.0 közötti seed-értékkre alkalmazzuk, ugyanebbel a tartományba eső más értéket állít elő az a \* seed mod m művelettel. A valós számokat csak a tüleszdulás elterületére használjuk.

A sorozat előállítására iterateq-t nextrandomra és seed valós számánál alkotott kezdőértékkel alkalmazzuk. mapq gondoskodik arról, hogy a sorozatban minden értéket elhosszunk m-relnél, és így randseq 0.0-nál nem kisebb és 1.0-nél kisebb értékeket adjon eredményül. Látható, hogy a sorozat a megalosztás részletét szépen elrijti a felhasználó elől.<sup>3</sup>

Az előállított álvéletlenszámok tehát 0.0-nál nem kisebb és 1.0-nél valós számok; mapq-val alkothatjuk át őket 0 és 1 közötti egészekké:

```

- mapq (floor o seqr 10.0 op*) (randseq 1);
> val it = Cons(0, fn) : int seq
- takeq(5, it);
> val it = [0, 0, 1, 7, 4] : int list

```

<sup>3</sup>Ez az algoritmust Park és Miller dolgozta ki 1988-ban. Ilyes működéshez a mantisszák emel rendszertől jóval rövidebb. Kevésbé jó statisztikai tulajdonságú álvéletlen-számokat kapunk, ha a -nak cs -nek kisebb relatív prímelek választunk.

### 15.3.2. Prímszámok

Következő SML programunk az eratosztenész szíta megrálosítása. A jól ismert algoritmust egy példa bemutatásával idézzük föl:

- 1. Végül az egészek 2-vel kezdődő sorozatát: [2, 3, 4, 5, 6, 7, ...]
- 2. Töröljük az összes 2-val osztatható számot: [3, 5, 7, 9, 11, ...]
- 3. Töröljük az összes 3-mal osztatható számot: [5, 7, 11, 13, 17, 19, ...]
- 4. Töröljük az összes 5-tel osztatható számot: [7, 11, 13, 17, 19, ...]
- 5. Töröljük az összes ...

Látható, hogy a sorozat első eleme mindenkor a következő prím. A sorozatban azok a számok maradnak le, amelyek az eddig elkövült prímekkel nem osztatják.

```
- fun sift p = filterq (fn n * n mod p <> 0);
- fun sift' p = Cons(p, fn () => sieve(sift p (nf())));
- val primes = sieve (fronq 2);
- val primes' = Cons(2, fn() : int seq
 - takeq(25, primes));
- val it = [2, 3, 5, 7, *, 83, 89, 97]: int list
```

### 15.3.3. Numerikus számítások

A példa legyen a négyzetgyökvonal Newton-Raphson módszerrel.

```
- fun nextapprox a x = (a/x + x)/2.0;
nextapprox xk-ból xk+1-est számítja ki az $x_{k+1} = \frac{x_k + x}{2}$ képlet alapján. A befejeződés megállapítására egszerű lesz írni:
- fun within (eps: real) (Cons (x, xf)) =
 let val Cons (y, yf) = xf()
 in if abs (x-y) <= eps then y else within eps (Cons (y, yf))
 end;
```

$A(\text{Cons } (y, yf))$  és  $a(xf)$  sorozat ugyanaz: az else-ágban azért használjuk megis az előbbiöt, hogy elkerüljük  $xf()$  különöséssel meghívását.

```
- fun qroot a = within 1E-6 (iterateq (nextapprox a) 1.0);
 - qroot 5.0;
 - val it = 2.236067977 : real
```

A fenti példa világosan különbözik a leállásvizsgálatot (termination test) a következő jelölőtől: elállásától.  
A példában az *abszolút különbséget* ( $|x - y| < \varepsilon$ ) vagy az  $(|x| + |y|)/2 + 1 < \varepsilon$  feltételt. A feladat többi

részre foggal lenne teljesítve, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást. minden egyes leállásvizsgálat telthát egy-egy függvény legyen olyan, amely egy sorozatból egy valós számot állít elő. Egy másik függvény állítja majd elő real seq -> real seq típusú függvényt, kérve a következő jelöltet. Ha később integrálni, differenciálni stb. akarunk, csak a megfelelő integráló, differenciáló stb. függvényeket kell megnini, a leállásvizsgálatot végző függvényet felhasználhatjuk.

Most tehát a következő jelölt elkövült-sorozására funk függvényt, és ezzel elérhetjük a részleteket:

```
fun approx a =
 let fun nextapprox x = (a/x + x) / 2.0
 in iterateq nextapprox 1.0
 end;
```

Most már könnyű megnini a qroot függvény egy tiszább változatát:

```
val qroot = within 1E-6 o approx;
```

### 15.4. Sorozatok sorozata és egymásba ékelése

Legyen  $xg$  és  $yg$  egy-egy sorozat. Képezzünk új sorozatot  $az(x_i, y_i)$  párokból, ahol  $x_i \in xg$  és  $y_i \in yg$ .<sup>1</sup> A feladat megoldása során látni fogunk, hogy a végelőn listák kezelése milyen különleges problémákat vet fel. Megkönytött a megoldás kidolgozását, ha először véges listára oldjuk meg ugyanez a feladatot map és pair alkalmazásával.

#### 15.4.1. Keresztszorzatokból álló lista

Legyen tehát  $xs$  és  $ys$  egy-egy lista. Képezzünk listát az  $(x_i, y_i)$  párokból, ahol  $x_i \in xs$  és  $y_i \in ys$ . map-et, pair-t és flat-et alkalmazva juthatunk el a keresett függvényhez. pair két értékbeli képez párt, flat listák listáját simítja listává. Idezük föl a definíciójukat:

```
- fun pair x y = (x, y);
- val pair = fn : 'a * 'b -> ('a * 'b)
- fun flat xs = foldl op@ ([], xs);
- val flat = fn : 'a list list -> 'a list
```

Most már felirhatjuk a feladat megoldását véges listá esetére. A részleges (pair x) függvény a rögzített x értékből és az argumentumából képez párt. Ha ezt a függvényt map az ys lista elemeire alkalmazzuk:

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

akkor olyan párokból álló listát kapunk eredményül, amelyben a párok első tagja a rögzített x érték, második tagja pedig az ys egy-egy eleme. Hogyan érhetjük el, hogy x végigfusson az xs lista összes elemén? Eloször is az előző szabad x-et kössük le egy függvény argumentumaként:

```
fn x => map (pair x) ys
```

majd pedig alkalmazzuk újabb map-ot erre a függvényre és xs-re:

```
map (fn x => map (pair x) ys) xs
```

Igen ám, de most listák listáját kapunk eredményül, hiszen a belső map már listát adott vissza, amelynek minden elemből újabb listát képezünk a különböző map-pel. Nevezük ezt a függvényt pairssnek (a nevet záró két s utal arra, hogy az eredmény listák listája):

```
- fun pairss xs ys = map (fn x => map (pair x) ys) xs;
```

```
> val pairss = fn : 'a list -> 'b list -> ('a * 'b) list list
flat elvégzi a szükséges simítást, és ezvel a feladatot megoldó pairs függvény definíciója:
```

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

#### 15.4.2. Keresztszorzatokból álló sorozat

Legyen  $xq$  és  $yq$  egy-egy sorozat. Képezzük új sorozatot az  $(x_i, y_i)$  párokból, ahol  $x_i \in xq$  és  $y_i \in yq$ . Terjűünk viszont az eredeti feladathoz, vagyis legyen  $xq$  és  $yq$  egy-egy sorozat! Képezzünk új sorozatot az  $(x_i, y_i)$  párokbol, ahol  $x_i \in xq$  és  $y_i \in yq$ !

A pairs-hez hasonlóan állíthatjuk elő páros sorozátnak sorozatát (a nevet záró két **q** utal arra, hogy az eredmény sorozatok sorozata):

```
- fun pairqq xq yq = mapq (fn x => mapq (pair x) yq) xq ;
> val pairqq = fn : 'a seq -> 'b seq -> ('a * 'b) seq seq
```

Az eredmény véges része kiiratható takeq-val, amely a bal felső saroktól számított első  $m$  sorból és n oszlopból álló *téglalapot* jelentíti meg az  $xqq$  sorozatból:

```
- fun takeqq ((m, n), xqq) = map (secl n takeq) (takeq(m, xqq)) ;
```

Állítsunk elő például páros sorozatából álló olyan sorozatot, amelyben a páros első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:<sup>4</sup>

```
- pairqq (fromq 30, primes);
> val it = Cons (Cons ((30, 2), fn), fn) : (int * int) seq seq
- takeqq ((3, 5), it);
> val it = [[(30, 2), ... (30, 11)], [(31, 2), ... (31, 11)], [(32, 2), ... (32, 11)]] : (int * int) list list
```

Az előző szakaszban alkalmazott flat listákhoz álló lista elemeit fűzi egymelen listába. Mi a helyzet végtelen sorozatok esetén? Ha  $xq$  végtelen, appendq ( $xq$ ,  $yq$ ) =  $xq$ , flat-hez hasonló függvénytel tehet nem megünk semmis! Ellenben két sorozat elemei *prónkent egymáshoz ékelhetők* (interleave):

```
fun interleaveq (Nil, yq) = yq
| interleaveq (Cons (x, xf), yq) = Cons(x, fn () => interleaveq(yq, xf()))
| takeq (0, interleaveq (fromq 0, fromq 50))
> val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
```

Vagyük észre, hogy interleaveq a *rekurzív hívásban* váltogatja a két sorozatot, egyik sorozat sem zára ki a másikat! Pl.

Most ennek néven olyan függvényt definiálunk, amely sorozatok sorozatából egymellen sorozatot állít elő. Legyen a két szélességi sorozat feje  $xq$  és a farka  $xf$ ; alkalmazzunk enumerate-et rekurzíván  $xqq$ -re, majd az eredményt ékeljük  $xq$ -ba:

```
- fun enumerate Nil = Nil
| enumerate (Cons (xq, xf)) = interleaveq (xq, enumerate(xf()))
| enumerate (Cons (Cons (x, xf), xqf)) =
 Cons(x, fn () => interleaveq (enumerate (xqf()), xf ()))
```

<sup>4</sup>Sajnos, az SML-terhelmező nem az általánosított olvasási alakra tördeli a kiírt szöveget, hanem csak a képernyőn kilogik sorokat vágja egy vagy több darabba.

Csakhogy ez a "megoldás" nem jó, mert ha a sorozat, amelyre alkalmazzunk, végeiben, a rekurió is végezen lesz! Nem lenne az luster nyelv esetén, az SML-ben azonban, amint van eredmény, a rekuriív hívást késlelteti kell, ezért több esetet kell megkülönöztetniünk:

```
- fun enumerate Nil = Nil
| enumerate (Cons (Nil, xqf)) = enumerate (xqf ());
| enumerate (Cons (Cons (x, xf), xqf)) =
 Cons(x, fn () => interleaveq (enumerate (xqf()), xf ()));
```

Vagyis ha a bemeneti sorozat üres, készteni vágunk. Ha nem üres, meg kell vizsgálni a sorozat fejét: ha ez üres, akkor folytatni kell a rekuriív hívást, de ha nem üres, akkor az explicit fn () => ... függvénydefiníciójával késleltetni kell a rekuriót.

Állítsunk elő például a pozitív egészekből álló páros egy sorozatát!

```
- val posintqq = Pairqq (fromq 1, fromq 1);
> val posintqq = Cons (Cons ((1, 1), fn), fn) : (int * int) seq seq
- takeq (15, enumerate posintqq);
> val it = [(1,1), (2,1), (3,1), (1,3), (2,2),
(1,4), (4,1), (2,3), (1,6), (3,2),
(1,7), (2,4), (1,8)] : (int * int) list
```

Az eredmény véges része kiiratható takeq-val, amely a bal felső saroktól számított első  $m$  sorból és  $n$  oszlopból álló téglalapot jelenti meg az  $xqq$  sorozatból:

Az előző szakaszban alkalmazott flat listákhoz álló lista elemeit fűzi egymelen listába. Mi a helyzet végtelen sorozatok esetén? Ha  $xq$  végtelen, appendq ( $xq$ ,  $yq$ ) =  $xq$ , flat-hez hasonló függvénytel tehet nem megünk semmis! Ellenben két sorozat elemei *prónkent egymáshoz ékelhetők* (interleave):

```
fun interleaveq (Nil, yq) = yq
| interleaveq (Cons (x, xf), yq) = Cons(x, fn () => interleaveq(yq, xf()))
| takeq (0, interleaveq (fromq 0, fromq 50))
> val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
```

Vagyük észre, hogy interleaveq a *rekurzív hívásban* váltogatja a két sorozatot, egyik sorozat sem zára ki a másikat! Pl.

Most ennek néven olyan függvényt definiálunk, amely sorozatok sorozatából egymellen sorozatot állít elő. Legyen a két szélességi sorozat feje  $xq$  és a farka  $xf$ ; alkalmazzunk enumerate-et

<sup>5</sup>Sajnos, az SML-terhelmező nem az általánosított olvasási alakra tördeli a kiírt szöveget, hanem csak a képernyőn kilogik sorokat vágja egy vagy több darabba.

A rekurzió befejezőlését korábban *teljes indukcióval* igazoltuk, rekurzív adattípusok, pl. listák és fák esetén *struktúrális indukcióval* bizonyítjuk.

A *teljes indukciót*, az egész számok halmozán értelmezniük, és azon alapul, hogy minden egész után egy nála egyvel nagyobb egész következik. Az INT típus így is lehetne deklaráció: datatype INT = 0 | Succ of INT. A struktúrális indukció a teljes indukció általánosítása rekurzív adattípusokra; szembetűnő a hasznoság az INT típus deklarációja és például a datatype 'a List = Nil | Cons of 'a List deklaráció között.

Az addott esetben a kiértékelés biztosan véget ér, mert ua-t vagy rekurzív módon alkalmazzuk az aktuális B vagy C fa egy részétől, amely biztosan rövidebb az aktuális fánál, vagy befeljződik a hívás, mert az aktuális fa A.

```
fun ugyanannyi f = let val (b, c, _) = ua f in b = c end
```

## 2. megoldás

A második paraméterként átadott számlálót egyel növeljük, ha B csomópontnak van A gyermeké, és csökkentjük, ha C csomópontnak van A gyermeké. (ua és ba, ill. ua és ca kölcsönösen rekurzív függvények.)

```
(* ua : 'a fa * int -> int
 ua(f, num) = num, ha f = A; vagy num + az f-beli B-k A gyermekéinek száma
 - az f-beli C-k A gyermekéinek száma
*)
fun ua(A, num) = num
| ua(B(x, y), num) = ba(x, ba(y, num))
| ua(C(x, y, z), num) = ca(x, cay, 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
```

## 1. megoldás

Összeszámoljuk, hogy a B és a C csomópontok hány A gyérneke van külön-külön, majd negezzük, hogy a két szám egyenlő-e.

```
(* ua : 'a fa -> int * int
 ua f = hármas: 1., ill. 2. tagja a B, ill. a C csomópontok A leveleinek a
 száma; 3. tagja 1, ha az aktuális csomópont A, egyébként 0
*)
fun ua (B(b1, b2)) =
 let val (b11, c11, a11) = ua b1
 val (b21, c21, a21) = ua b2
 in (b11 + b21 + a11 + a21, c11 + c21, 0)
 end
| ua (C(c1, c2, c3)) =
 let val (b11, c11, a11) = ua c1
 val (b21, c21, a21) = ua c2
 val (b31, c31, a31) = ua c3
 in (b11 + b21 + b31, c11 + c21 + c31 + a11 + a21 + a31, 0)
 end
| ua A = (0, 0, 1)
```

## 3. megoldás

A 2. megoldás egyszerűsített változata egy újabb paraméterrel használ a növekmény átadására. Ennek erére B csomópont esetén +1, C csomópont esetén pedig -1. (Seregi Péter megoldása.)

```
local
 (* ua : 'a fa * int * int -> int
 ua (f, num, incr) = num + incr, ha f = A; vagy num + incr + az f-beli
 B-k A gyermekéinek száma - az f-beli C-k A gyermekéinek száma
 *)
 fun ua (C(c1, c2, c3), num, incr) =
 ua(c1, ua(c2, ua(c3, num, incr)) =
```

```
| ua (B(b1, b2), num, incr) = ua(b1, ua(b2, num, 1), 1)
| ua (A, num, incr) = num + incr
in
 fun ugyanannyi f = ua(f, 0, 0) = 0
end
```

## 16.2. Fa adott tulajdonságú részfáinak száma (bea)

Tehetsük az alábbi adatípus-deklarációt:

```
datatype fa = A | B of fa * fa | C of fa * fa * fa
fjon bea néven olyan SML-függvényt, amely megszánálja egy fa típusú fában azokat a B csomópontokat, amelyeknek minden részfája B vagy A (de nem C), és ezeknek a számát adja eredményül!
Segedfüggvényt definíálhatunk.
```

```
bea f = azoknak az f-beli B-knek a száma, amelyeknek csak B vagy A
részfájuk van
bea : fa -> int
```

Példák:

```
bea A = 0;
bea(B(B(A,A),C(A,A,A))) = 1;
bea(B(C(B(A,A),C(A,A,A),B(A,A)),B(B(A,A),C(A,B(A,A),A)))) = 4;
```

### 1. megoldás

Aba segédfüggvényt az olyan B-ket számlálja meg, amelyeknek egyetlen utódja sem C.  
 ba : fa -> int \* bool  
 ha f-ben van C

\*

```
fun ba A = (0, false)
| ba (C(bf, kf, jf)) =
 let val (bb, _) = ba bf
 val (kb, _) = ba kf
 val (jb, _) = ba jf
 in
 (bb+kb+jb, true)
 end
| ba (B(bf, jf)) =
 let val (bb, bc) = ba bf
 val (jb, jc) = ba jf
 val b = bc orelse jc
 in
 (bb + jb + (if b then 0 else 1), b)
 end
```

Ha az aktuális fa A, a jó B-k száma nem változik, az össék között pedig lehetnek jó B-k (ezért false az eredménypár második tagja). Ha az aktuális fa C, a részfai és ezek utódai között lehetnek jó B-k, de az összi között egyetlen B sem lehet jó (ezért true az eredménypár második tagja). Ha az

### 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
 in
 (* szamol f = f jó B csomópontjainak száma
 szamol : fa -> int
 *)
 fun szamol A = 0
 | szamol (B(I, 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
 end
```

### 16.3. Fa adott tulajdonságú részfáinak száma (testverE)

Ijón testverE néven olyan SML-függvényt, amely a datatype 'fa = E | N of 'a fa \* 'a fa \* 'a fa deklarációval meghozott fában megláthatózza azoknak az E levelének a számát, amelyeknek legalább egy testvérük van! Egy E levél testvéreinek az ugyanahoz az N csomóponthoz tartozó másik E levélét nevezük.  
 A függvény specifikációja:

```
testverE f = az E testvérek száma az f fában
testverE : 'a fa -> int

Példák:
testverE E = 0;
testverE (N(E,E,E)) = 3;
testverE (N(E,N(E,E,E),NN(N(E,E,E),E,E))) = 8;
testverE (N(E,N(E,E,E),N(E,E,E))) = 6;
```

### Megoldás

A feladat és a megoldása nagyon egyszerű. Ügytünk arra, hogy csak a valóban megkölönböztetendő esetekre írunk fel változatokat. Figyelj meg, hogy az  $N(f1, E, E)$  eseteket visszavezetünk az  $N(E, E, f3)$  esetbe. Ezel ugyan egy lépéssel mélyítettük a rekurziót, de ha később a program adott ágai 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
```

### 16.4. Fa adott eleméinek összegzése (szint0ssz)

Feliratunk szint0ssz néven olyan SML-függvényt, amely egy bináris fában tárolt értékek szintenkénti összegéből alkotott listát ad eredményül! A lista első eleme az első szinten lévő gyökérélelm értéke, második eleme a második szinten tárolt, legfeljebb két elem összege stb. A fa típusa:

```
datatype itree = L of int | N of itree * int * itree
```

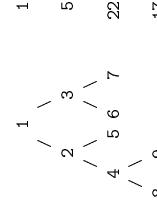
A függvény specifikációja:

```
szint0ssz t = a t-beli elemek szintenkenti összegének lista
szint0ssz : itree -> int list
```

Példák:

```
szint0ssz (L 1999) = [1999];
szint0ssz (N(N(L 4, 2, L 5), 1, N(N(L 8, 6, L 9), 3, L 7))) = [1, 5, 22, 17];
```

A második példában használt fa és a szintenkenti összegek ábrázolása:



### 1. megoldás

listaoosszeg két, esetleg különböző hosszságú lista eleméinek páronkénti összegéből álló listát ad eredményül. (A rövidebb listából hiányzó elemeket "pölöljük") Jobbtekurzív változata az elemek sorrendjét megfordítaná, ezért az eredeti sorrendet `rev`-vel helyre kellene állítani.

```
local
 (* lista0sszeg (xs, ys) = az xs és ys elemeiből páronként képzett
 összegek lista)
 lista0sszeg : int list * int list -> int list
 *)
 fun lista0sszeg (x::xs, y::ys) = x+y:lista0sszeg(xs, ys)
 | lista0sszeg ([] , ys) = ys
 | lista0sszeg (xs, []) = xs

```

### 2. megoldás

A szint0ssz függvény az N csomópontban előírja a bal, ill. a jobb részfa szintenkenti összegéinek listáját, majd a két lista elemét páronként összeadják. Az L level egyetlen elemből egyelemű listát képezi.

```
local
 (* s0 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 eleménél felel meg; ahogy egyre
 mélyebbre haladunk a fában, úgy építjük a listát, ill. haladunk jobbról bárhá a már felépült listában.
 *)
 fun s0(t, xs) = az egyes szinteken lévő t-beli és a megfelelő xs-beli
 elemek összegének a lista
 s0 : itree * int list -> int list
 *)
 fun s0 (L, v, []) = [v]
 | s0 (L, v, x::xs) = x+v::xs
 | s0 (N(1, v, r), []) = v::s0(1, s0(r, []))
 | s0 (N(1, v, r), x::xs) = x+v::s0(1, s0(r, xs))
 in
 fun szint0ssz t = s0(t, [])
 end
```

### 3. megoldás

Végül őszre, hogy a 2. megoldásban az s0 segédfüggvény két-két klózra alig különbözik egymástól. A hasonlóságot még jobban kiemeljejjük:

```
fun s0 (L, v, xs as []) = 0+v::xs
| s0 (L, v, x::xs) = x+v::xs
| s0 (N(1, v, r), xs as []) = 0+v::s0(1, s0(r, xs))
| s0 (N(1, v, r), x::xs) = x+v::s0(1, s0(r, xs))
in
 fun szint0ssz t = s0(t, [])
end
```

Az egymáshoz hasonló klózokat összevonhatjuk (Szeredi Péter megoldása):

```
local
 (* feje : int list -> int
 feje xs = hd xs vagy 0, ha xs = []
 *)
 fun feje [] = 0
 | feje (x::_) = x
 fun farka : 'a list -> 'a list
 farka xs = tl xs vagy [], ha xs = []
 *)
 fun farka [] = []
 | farka (_::xs) = xs

```

```
(* s0(t, xs) = az egyes szinteken lévő t-beli és a megfelelő
 xs-beli elemek összegének a listája
 s0 : itree * int list -> int list
*)
fun s0 (L v, xs) = feje x + v :: farka xs
| s0 (Nl, v, r), xs) = feje x + v :: s0(l, s0(r, farka xs))
in
 fun szint0ssz t = s0(t, □)
end
```

## 16.5. Kifejezésfa egyszerűítése (egyszerusít)

Az alábbi adattípus-definiciók olyan kifejezést írnak le, amelynek a levelei egész számok, a gyökerébeni jellek:

```
datatype oper = ++ | -- | * | /
```

```
datatype Expr = Lf of int | Br of oper * Expr * Expr
```

Felirat olyan SML-függvényt egyszerűít néven, amely egy kifejezésfában az  $m+n$  alakú részki-fejezések összes előfordulását az összegül, az  $m+n$  alakú részkifejezések összes előfordulását a szorozatukra cseréli, a többi részkifejezést pedig változtatnál hagyja ( $m$  és  $n$  egész számok)!

Gondoljon a redukció során keletkező, hasonló alakú részkifejezések helyettesítésére, de kerülje el a végtelen rekurziót!

A függvény specifikációja:

```
egyszerusít kf = kf egyszerűített változata, amelyben m+tn,
ill. m*xn összes előfordulással helyen m és n összege,
ill. szorzata van
```

```
egyszerusít : Expr -> Expr
```

Példák:

```
egyszerusít(Br(++,Lf 1,Lf 2)) = Lf 3;
egyszerusít(Br(/, Br(++, Br(*,Lf 3,Lf 4), Br(++,Lf 5,Lf 6)),Lf 7)) =
Br(/,/Lf 23,Lf 7);
egyszerusít(Br(/,/Br(/,/Lf 3,Lf 4),Lf 5)) =
Br(/,/Br(/,/Lf 3,Lf 4),Lf 5);
egyszerusít(Br(--,Br(--,Br(*,Lf 3,Lf 4), Br(++,Lf 5,Lf 6)),Lf 7)) =
Br(/,, Br(--,Lf 12,Lf 11),Lf 7);
```

## 1. megoldás

A  $++$  és a  $**$  műveleti jeleket tartalmazó részkifejezések kezelésére két-két változatot kell írni: eggyel-egyel a  $++$ , ill.  $**$  műveleti jelből és pontosan két levelből (jelöljük  $Lf$   $b$ -vel és  $Lf$   $j$ -vel)

Az első két esetben a kiejelölt művelet elvezethető, az eredményen az  $Lf(b+j)$  level. A két utóbbi esetben egyeszerűítés karanzsír először is a bal és a jobb részről

egyszerűsítőjük. Előfordulhat, hogy mindenkető levél vár, ezért meg kell próbálni, hárha további rekurzív hívásokat lehet egyszerűsíteni az adott műveleti jelből és a redukált részfákból összerakott

fáj is.

Ha az aktuális fa gyökérében más műveleti jel van, egyszerűsítrekurzív hívásával csak a bal és a jobb részről egyszerűsítjük, további redukcióra nincs lehetőség. Nem lehet egyszerűsíteni a kifejezést akkor sem, ha az aktuális fa level.

## 2. megoldás

Három változat (klöz), összevonásával, valamint a közös részek kiemelésével a megoldás rövidebbé tehető.

```
fun egyszerusít (Br(++, Lf b, Lf j)) = Lf(b+j)
| egyszerusít (Br(**, Lf b, Lf j)) = Lf(b*j)
| egyszerusít (Br(++, bf, jf)) =
 egyszerusít (Br(**, bf, jf)) =
 egyszerusít (Br(**, bf, jf)) =
 egyszerusít (Br(mj, bf, jf)) =
 egyszerusít (Br(mj, bf, jf)) =
 Br(mj, egyszerusít bf, egyszerusít jf)
| egyszerusít (kf as Lf v) = kf
```

## 16.6. Kifejezésfa egyszerűítése (coeff)

Tekintse az alábbi típust és adattípust:

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

```

Egy term típusú párt, egy egész együtthatós és egy char típusú változónév szorzatának, egy expr típusú kifejezést term típusú tagok és Z (zérus) állandók összegének tekintünk.  
Írjon SML-függvényt coeff nevén, amelynek expr típusú kifejezésből és char típusú változónév ből álló pár az argumentuma, és az eredményne az adott változó együtthatónak az összege az adott kifejezésben! Hatékony, jobbkurziv programot írjon! Segédfüggvényt definíthet.

A függvény specifikációja:

```
coeff (e, v) = v egütthatónak az összege e-ben
coeff : expr * char -> int
```

Példák:

```
coeff(Z ++ (2,#"a") ++ (3,#"b") ++ (~5,#"a") ++ (4,#"c"), #"a") = ~3;
coeff(Z ++ (2,#"a") ++ (3,#"b") ++ (~5,#"a") ++ (4,#"c"), #"x") = 0;
```

## Megoldás

Figyelem meg, hogy a Z az expr típusú kifejezések *hololaki egysége*: Z maga is expr típusú kifejezés, az expr típusú kifejezésekben pedig csak a bal oldali állhat expr típusú kifejezés.

A cf segédfüggvény az n argumentumban gyűjti az e-beli v-k együtthatónak 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

 pp : char list * int list * (int * int) list ->
 (int * int) list
 *)
 fun pp ("":cs, i, bs, ps) = pp(cs, i+1, i:bs, ps)
 | pp (#":cs", i, b:bs, ps) = pp(cs, i, b:bs, ps)
 | pp (_:cs, i, bs, ps) = pp(cs, i+1, bs, ps)
 | pp ([], _, _, ps) = rev ps
 in
 pp(explode s, 1, [], [])
 end

```

### 16.7. Szövegfeldolgozás (parPairs)

Főjön SML-függvényt parPairs néven, amely az argumentunként kapott füzérben található, egymáshoz tartozó kerek nyitó- és csukzárobjectek pozícióiból alkotott párok listáját adja eredményül, tetszőleges sorrendben! A füzér karakterei 1-től számoznak. Segédfüggvényt definiálhat.

A függvény specifikációja:

```

parPairs s = az s füzérbeli, egymáshoz tartozó, kerek nyitó- és
 csukzárobjectek pozícióból alkotott párok listája
parPairs : string -> (int * int) list
Példák:
parPairs "Zárójelmentes." = [];
parPairs ")" = [];
parPairs "(" = [];
parPairs "real(3*4) + (sin(0..5) - (11..4+3..4)) * 1..2" =
 [(5, 11), (19, 23), (27, 38), (15, 39)];

```

#### Megoldás

Az alábbi megoldásban kihasználjuk, hogy a lista verenként, azaz LIFO-tárként használható: amiit legutoljára rakunk bele, azt vesszük ki belőle legkiselebb.

A füzetet az explode függvény karakterlistára alakítja. A pp segédfüggvény, ha kerek nyitózárobjectet talál, az indexét (azaz helyénnek sorrendjét az eredeti füzérben), berakja a bs verenbe. Ha csukzárobjectet talál, és a bs verem nem üres, kiveszi a bs-ból a megfelelő nyitózárobject indexét, és a (b, i) párt berakja ps-be. Ha egyik karaktert talál, egyszerűen torzítja a lisfában. I. értelek két minden egész lépésben 1-gyel megnöveli. Ha a lista elfogy, a indexpárok ps-ben összegyűjtött listáját az eredeti sorrendbe rakva (azaz rev-et alkalmazva) adja eredményül.

**Megjegyzés:** A pp segédfüggvényt nem lenne könnyű deklarációjában módosan specifikálni, ezért megelőzünk a műveleti szemléltető specifikációval.

```

fun parPairs s =
 let (* pp (cs, i, bs, ps) =
 cs = a feldolgozandó karakterek listája;
 i = a cs első karakterének az indexe
 (= helye az eredeti füzérben);
 bs = a még le nem zárt nyitózárobjectek indexének
 fordított sorrendű lista;
 ps = az egymáshoz tartozó nyitó- és csukzárobjectek
 indexeiből álló párok listája

```

```

fun osszefuttat (x::xs, y::ys, zs) = osszefuttat(xs, ys, y::zs)
| osszefuttat ([], [y], zs) = rev(y :: zs)
| osszefuttat ([x], [], zs) = rev(x :: zs)
| osszefuttat (_ , _ , zs) = rev zs (* lehetetlen eset *)

```

```

val is = String.tokens (not o joKar) s
val os = String.tokens joKar s
in
 if String.isPrefix (hd is) s
 then osszefuttat (ls, os, [])
 else osszefuttat (os, is, [])
end;

```

## 2. megoldás

Ez a megoldás nem használja sem a String.tokens, sem a String.isPrefix függvényt.

```

exception Mezok;
fun mezok ("", _, _) = []
| mezok (s, c1, c2) =
 let (* joKar c = igaz, ha c a [c1, c2] zárt intervallumba esik
 joKar : char -> bool *)
 fun joKar c = c >= c1 andalso c <= c2
 (* mezok0(cs, js, rs, ts) =
 mezok0 : *)
 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)
 in
 mezok0 ([], js as j::js, [], ts) = rev j:js :: ts
 | mezok0 ([], [], rs as r::rs, ts) = rev rs :: ts
 | mezok0 _ = raise Mezok (* lehetetlen eset *)
 end
 val (js, rs) = if joKar c then ([c], []) else ([], [c])
 in
 map implode (rev mezok0(cs, js, rs, []))
 end;

```

## 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 meghatározásra megvizsgáljuk, hogy melyik lista első elemével kezdődik az eredeti füzér.

```

fun mezok(s, c1, c2) =
let (* joKar c = igaz, ha c a [c1, c2] zárt intervallumba esik
 joKar : char -> bool *)
 fun joKar c = c >= c1 andalso c <= c2
 (* osszefuttat(xs, ys, zs) = az xs és az ys elemei váltakozva
 a zs elé fűzve
 PRE : |length xs - length ys| <= 1
 osszefuttat : 'a list * 'a list * 'a list -> 'a list
 *)

```

## 17.2. Füzér adott tulajdonságú elemei (baseName)

írjon olyan SML-függvényt basename néven, amely egy füzérként megadott állománynev utolsó névűres komponensét adja eredményül! A névben egy más után öbbször szereplő #"/" karaktereket egyetlen #"/" karakternek vége! Legalább egyet alkalmazzon a String.fields, String.tokens, List:nth, List.length, List.last, List.take függvények között!

A függvény specifikációja:

# Példaprogramok: füzérek és listák kezelése

## 17. fejezet

### 17.1. Füzér adott tulajdonságú elemei (mezok)

Feljön mezok néven olyan SML-függvényt, amelynek egy füzér maximális hosszúságú, nemüres részeiből álló lista az eredményel! E lista elemei az eredeti sorrendben felsorolt olyan füzérek, amelyek vagy csak a megadott, zárt intervallumban tartozó, vagy csak az ezen kívül eső karakterekből állnak. Használhatja a String.tokens és a String.isPrefix magasabb rendű függvényeket. Segédfüggvényt definíálhat (pl. két lista összehatárolására).

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
 intervallumban, vagy azon kívül esik
 mezok : string * char * char -> string list

```

Példa:

```

mezok ("Ali Baba + a 40 rablo", #"a", #"n") =
 ["A", "Li", " ", "babá", " " + " ", "a", " ", "40 r", " ", "ab1", " ", "o"]

```

## 1. megoldás

Egy-egy listába szétválogatjuk a füzér adott intervallumba eső, ill. azon kívüli karakterekből álló szakaszait, majd a két lista elemeit a megfelelő sorrendben összefuttatjuk. A megfelelő sorrend meghatározásra megvizsgáljuk, hogy melyik lista első elemével kezdődik az eredeti füzér.

```

fun mezok(s, c1, c2) =
let (* joKar c = igaz, ha c a [c1, c2] zárt intervallumba esik
 joKar : char -> bool *)
 fun joKar c = c >= c1 andalso c <= c2
 (* osszefuttat(xs, ys, zs) = az xs és az ys elemei váltakozva
 a zs elé fűzve
 PRE : |length xs - length ys| <= 1
 osszefuttat : 'a list * 'a list * 'a list -> 'a list
 *)

```

```

basename s = az s állománynév utolsó neműres komponense
PRE: s <> ""
 string -> string
basename : string -> string

Példák:
basename "dr-1/dr-2/file.ext" = "file.ext";
basename "dr-1//dr-2//file.ext" = "file.ext";
basename "dr-1/file.ext" = "file.ext";
basename "dr-1/file.ext//" = "file.ext";
basename "///" = "";
basename "/" = "";
```

### Megoldás

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

```

fun basename s = let val ts = String.tokens (fn c => c = #"/"') s
 in
 if null ts then "" else List.last ts
 end;
```

### 17.3. Füzér adott tulajdonságú elemei (rootname)

Függetlenül a rootname néven, amely egy fizérként megadott állománynév első neműres, esetleg #"/" jelű kezdő komponensét adja eredényül! A névben egymás után több ször szereplő #"/" karaktereket egyetlen #"/" karakternek vége! Legalább egyet alkalmazzon a String.fields, String.tokens, List.nth, List.length, List.last, List.take, List.drop függvények között!

A független specifikációját:

```

rootname s = az s állománynév első neműres komponensee
PRE: s <> ""
 string -> string

Példák:
rootname "dr-1/dr-2/file.ext" = "dr-1";
rootname "dr-1/file.ext///" = "/dr-1";
rootname "///file.ext///" = "/file.ext";
rootname "/" = "/";
rootname "///" = "/";
```

### Megoldás

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

```

fun rootname s = let val ts = rev(String.tokens (fn c => c = #"/"') s)
 in
 if null ts then "/"
 else (if String.sub(s, 0) = #"/"
 then "/"
 else "") ~ List.last ts
 end
```

### 17.4. Lista adott tulajdonságú részlistái (szomsor)

Függetlenül a rootname néven, amely elgállíja egy adott egészszám-lista olyan (általában nem folytonos, de a sorrendet megőrző) részlistáinak listáját, amelyben a rész-listák legalább kételőnél maximális (egykirályban sem kiterjeszhető), egysével növekvő számtani sorozatot alkotnak! A listáról feltételezi, hogy csupa különbözöző egészszámok áll.

```

(* szomsor ns = ns legalább kételőnél, maximális (egykirályban sem kiterjeszhető), egysével növekvő számtani sorozatot alkotó részlistáinak listaia (feltételező, hogy ns minden eleme különböző)
 szomsor : int list -> int list list
 *)
```

Sugallás: Irjon segédfüggvényt, amely az egészlista első elemével kezdődő számtani sorozatot alkotó listát és a számtani sorozatban fel nem használt elemek listáját adja vissza párhuzamosan, majd vizsgálja meg, hogy van-e még mit felelőzni. Csak legalább két elemű sorozatokat tüntessen az eredménylistához!

Példa:

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

### Megoldás

Készüljük a segédfüggvényt!

```

(* szomsor ns ss ms = pár, amelynek első tagja az ns első elemével
 kezdődő, a felületének megfelelő számtani sorozat,
 második tagja ns fel nem használt elemeinek a listaja,
 mindenkitőz az eredeti sorrendben
 szomsor ns ss ms : int list * int list -> (int list * int list)
 *)
 fun szomsor [] ss ms = (rev ss, rev ms)
 | szomsor (n::ns) [] ms =
 | szomsor (n::ns) (sss as s::ss) ms =
 if n=s+1
 then szomsor ns (n::sss) ms
 else szomsor ns sss (n::ms);
 fun szomsor (ns as n1::n2::ns) =
 let val (ss, ms) = szomsor ns [] []
 in if length ss >= 2
 then ss :: szomsor ms
 else szomsor ms
 | szomsor _ = [];
```

### 17.5. Bináris számok inkrementálása (binc)

Függetlenül a rootname néven SML-függvényt egy listaként ábrázolt bináris szám inkrementálására! A bináris számjegyeket az I és 0 adattípusokkal jelölik, típusuk:

```
datatype bin = I | 0
```

A listák feje a legnagyobb helyiértékű bináris számjegy, amely sohasem 0. A független specifikációja:

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

Példák:

```
binc [0] = [1]
binc [1] = [1,0]
binc [1,1,1,1,1,1,1] = [1,0,0,0,0,0,0,0,0];
```

### Megoldás

Először olyan segédfüggvényt írunk, amely fordított sorrendű bináris számokat inkrementál, és a bináris számjeleket egy gyűjtőargументumban gyűjti.

```
local
 (* binc0 (bs, c, rs) = rs a bs fordított sorrendű bináris szám normál
 * sorrendű inkrementáltja (a bs lista füje
 * a legkisebb helyiértéktől bit, c a növekmény)
 binco : (bin list * bin * bin list) -> bin list
 *)
 fun binc0 (b::bs, 0, rs) = binco(bs, 0, b::rs)
 | binc0 (0::bs, I, rs) = binco(bs, 0, I::rs)
 | binc0 (I::bs, I, rs) = binco(bs, I, 0::rs)
 | binc0 ([], 0, rs) = rs
 | binc0 ([], I, rs) = I::rs
in
 (* binc ns = az ns bináris szám inkrementáltja
 * binc : bin list -> bin list
 *)
 fun binc ns = binc0(rev ns, I, [])
end
```

Kiegészítésképpen íjunk egy-egy segédfüggvényt bin list típusú bináris számok és int típusú egészek egymáshára alakítására b2i, il. i2b néven!

```
(* b2i bs = a bs bináris szám egész számként, a vezető 0-k nélkül
b2i : bin list -> int
*)
fun b2i bs =
 case Int.fromString(implode(map (fn 0 => #"0" | I => #("1" ^ bs)) of
 SOME i => i
 | NONE => 0;
 end
```

```
app load ["Int"];
(* i2b i = az i egész bináris megfelelője (0 -> 0, nem 0 -> 1),
 a vezető 0-k nélkül
i2b : int -> bin list
*)
fun i2b i = map (fn #"0" => 0 | _ => 1) (explode(Int.toString i));
```

### 17.6. Mátrix transzponáltja (trans)

Főjön trans néven olyan SML-függvényt, amely elbállítja egy mátrix transzponáltját! Egy mátrixot sorok listájákról adunk meg, ahol a sorok a mátrix elemek listái. (Egy a [n..m] nem-es mátrix transzponálja az a b[m..n] méréses mátrix, ahol b[k..l] = a[l..k].) Rövidtási függvényeket használhat, de saját segédfüggvényt ne definíáljan!

A függvény specifikációja:

```
trans mss : az mss mátrix transzponáltja
trans : `a list list -> `a list list
```

Példa:

```
trans [[1,2,3], [4,5,6], [7,8,9], [0,0,0]] =
 [[1,4,7,0], [2,5,8,0], [3,6,9,0]]
```

#### 1. megoldás

A mátrix sorainak – a részlistáknak – az első elemét ill. a többi elemét rekurzív módon egy-egy listába (ms1, mss1) gyűjtjük, amíg csak vannak feldolgozható elemek. Nem elég a teljes lista nem üres voltát vizsgálni, azt is meg kell nézni, hogy az egyes részlisták nem üresek: az utóbbit ellenőri List exists.

```
fun trans mss =
 if (not o null) mss andalso List.exists (not o null) mss
 then
 let val (ms1, mss1) = (map hd mss, map tl mss)
 in
 ms1 :: trans mss1
 end
 else []
end;
```

#### 2. megoldás

Valamivel hatékonyabb az alábbi megoldás, mert a teljes lista üres voltát csak egyetlen egyszer ellenőri. Ha nem volt üres kezdetben, nem is válhat azzá menet közben: csak a részlisták végeire.

```
fun trans [] =
 trans mss = if List.exists null mss then []
 else let val (ns, nss) = (map hd mss, map tl mss)
 in ns :: trans nss
 end;
```

#### 3. megoldás

Talán ez a lehetséges legtömörebb megoldás: az egyik map kifagyítja 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

Kévesbé hatékony a tabulate függvényt használó, két egymásba ágyazott ciklusra építő megoldás:

```
fun trans [] = []
| trans mss = List.tabulate(List.length(hd mss),
 fn r => List.tabulate(List.length(mss,
 fn c => List:nth(List.nth(mss, c), r))));
```

## 18. fejezet

### Az SML szintaxisa

#### 18.1.2. Infix operátorok

Egy név az infix vagy az infix direktívával *infix* helyzetűnek deklarátható. Ha az *id* név infix helyzeti, akkor az *exp1 id exp2* kifejezés, szükség esetén zárójelk között, minden olyan helyen használható, ahol az *id(exp1, exp2)* vagy az *id{I-exp1, 2-exp2}* kifejezések egyébként használhatók. Infix helyzetű nevek minimálban színén használhatók.

Egy minősített névet, vagy egy olyan nevet, amelyet az op szócska előz meg, sohasem lehet infix helyzethen alkalmazni. Az infix, infixr és nonfix direktívák szintaxisa a következő ( $n \geq 1$ ):

|        |                   |                   |            |                   |
|--------|-------------------|-------------------|------------|-------------------|
| infix  | $<\mathbf{d}>$    | $id_1 \dots id_n$ | balra köt  | left associative  |
| infixr | $<\mathbf{d}>$    | $id_1 \dots id_n$ | jobbra köt | right associative |
| nonfix | $id_1 \dots id_n$ | neni köt          |            | non-associative   |

A d decimalis számjegy opcionális, és a nevek precedenciáját adja meg; alapértelmezés szerinti értéke 0. Nagyobb szám magasabb precedenciájú. Az infix, infixr és nonfix direktívák érvényességi tartománya a szokásos, azaz a let kifejezésen és a local deklaráció belül lokális érvényűek.

Azonos precedenciájú, de különféle balra kötő operátorok balra, azonos precedenciájú, de különféle jobbra kötő operátorok pedig jobbra kötők. Típus azonos precedenciájú, de különféle kötésű operátorokat használhatnak ugyanabban a kifejezésben.

#### 18.1.3. Jelölések

- minden szintaktikai osztályt változatok listájájaként adunk meg, mégpedig soronkent egy váltózatot. Üres sor türes kifejezést fognak közre.

- $A < \& s a >$  csúcos 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):

|            |                   |                  |                    |
|------------|-------------------|------------------|--------------------|
| $xseq ::=$ | $x$               | egyelémű sorozat | singleton sequence |
|            | $x_1, \dots, x_n$ | türes sorozat    | empty sequence     |
|            | sorozat           |                  |                    |

- A kifejezés változatokat precedenciájuk csökkenő sorrendjében adjuk meg.

- A típusok szintaxisa erősen köt, mint a kifejezések.

- A megfelezés oszlopban az L betű balra kötő műveletet jelöli.

- minden ismétlődő konstrukció (pl. a *klasszorozat*) a lehető legtovább terjeszkedik jobbra. Ezért egy case-kifejezést egy másik case-, fn- vagy fun-kifejezésen belül esetleg zárójelbe kell rakni.

- Az különféle típusú állandók jelölését a 2.1.3.1 fejezetben ismertettük.

A szintaxisleírás a Moscow ML Language Overview c. kézikönyv alapján készült.

#### 18.1.1. Fogalmak és jelölések

##### 18.1.1.1. Nevezések

Egy név lehet

- alfánumérikus, azaz betűk, számjegyek, percjek és aláhúzásjelek olyan sorozata, amely betűvel vagy percjellel kezdődik;
- szimbolikus, azaz az alábbi jelek tetszőleges, nem üres sorozata:

$! \quad \% \quad \& \quad \$ \quad # \quad + \quad - \quad / \quad : \quad < \quad = \quad > \quad ? \quad @ \quad \backslash \quad \sim \quad \cdot \quad | \quad *$

Névként nem használhatók a következők, ún. fenntartott szavak vagy kulcsszavok:

```
abstype and also as case do datatype else end
exception fn fun handle if in infix inixr let local
nonfix of open orelse raise rec sig signature
struct structure then type val with withtype while
() [] { } , : ; ... - | = => #
```

A nevezéket különféle osztályokba soroljuk:

|               |                      |                      |                      |
|---------------|----------------------|----------------------|----------------------|
| <i>var</i>    | értékváltozó         | értékváltozó         | értékváltozó         |
| <i>con</i>    | adatkonstruktör      | adatkonstruktör      | adatkonstruktör      |
| <i>excon</i>  | kivételekkonstruktör | kivételekkonstruktör | kivételekkonstruktör |
| <i>tynar</i>  | típusváltozó         | típusváltozó         | típusváltozó         |
| <i>tycon</i>  | típuskonstruktör     | típuskonstruktör     | típuskonstruktör     |
| <i>lab</i>    | mezőnév              | mezőnév              | mezőnév              |
| <i>unitid</i> | modulnév             | modulnév             | modulnév             |

- 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 long-X pária. A long-X osztályba tartozó névek 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:

## 18.2. Az SML Core language szintaxisa

### 18.2.1. Kifejezések és klózsorozatok

|               |                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                  |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>atexp</i>  | ::= <i>infexp</i><br><i>exp : ty</i><br><i>exp1 andalso exp2</i><br><i>exp1 orelse exp2</i><br><i>exp handle match</i><br><i>raise exp</i><br><i>if exp1 then exp2 else exp3</i><br><i>while exp1 do exp2</i><br><i>case exp of match</i><br><i>fn match</i>                                                                                                       | típusmegkötés (L)<br>sequential konjunkció<br>sequential disjunction<br>handle exception<br>raise exception<br>conditional expression<br>iteration<br>case analysis<br>function expression                                                                                                                                                                       |
| <i>infexp</i> | ::= <i>appexp</i><br><i>infexp1 id infexp2</i>                                                                                                                                                                                                                                                                                                                     | infix alkalmazás                                                                                                                                                                                                                                                                                                                                                 |
| <i>appexp</i> | ::= <i>atexp</i><br><i>appexp atexp</i>                                                                                                                                                                                                                                                                                                                            | (prefix) alkalmazás                                                                                                                                                                                                                                                                                                                                              |
| <i>atexp</i>  | ::= <i>scon</i><br><i>&lt;op&gt; longar</i><br><i>&lt;op&gt; longcon</i><br><i>&lt;op&gt; longazon</i><br><i>f &lt; exprow &gt; J</i><br><i># lab</i><br><i>()</i><br><i>(exp1, ..., expn)</i><br><i>[exp1, ..., expn]</i><br><i># [exp, ..., expn]</i><br><i>(exp1; ...; expn)</i><br><i>let dec</i><br><i>in exp1; ...; expn</i><br><i>end</i><br><i>( exp )</i> | állandó<br>értékéről<br>adattípuskonstruktör<br>kivételekkonstruktör<br>rekord<br>rekordszelektör<br>nullas<br>ennas, $n \geq 2$<br>lista, $n > 0$<br>vektor, $n \geq 0$<br>kifejezessorozat, $n \geq 2$<br>sorozat, $n \geq 2$<br>list, $n > 0$<br>vektor, $n \geq 0$<br>sorozat, $n \geq 2$<br>lokális kifejezés, $n \geq 1$<br>lokális expression, $n \geq 1$ |
| <i>exprow</i> | ::= <i>lab = exp &lt; , exprow &gt;</i>                                                                                                                                                                                                                                                                                                                            | kifejezessor                                                                                                                                                                                                                                                                                                                                                     |
| <i>match</i>  | ::= <i>mrule &lt;   match &gt;</i>                                                                                                                                                                                                                                                                                                                                 | klózsorozat, változat-sorozat                                                                                                                                                                                                                                                                                                                                    |
| <i>mrule</i>  | ::= <i>pat =&gt; exp</i>                                                                                                                                                                                                                                                                                                                                           | klóz, változat                                                                                                                                                                                                                                                                                                                                                   |

*exprow* :: expression row

*match* :: expression row

*mrule* :: match rule

### 18.2.2. Declaraciók és kötések

|                 |                                                                                                                                                                                                                                                                                                                                          |                                                                                                        |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <i>dec</i>      | ::= <i>val tyvarseq valbind</i><br><i>fun tyvarseq funbind</i><br><i>type typhbind</i><br><i>datatype datbind</i><br><i>&lt; whtype typhbind &gt;</i><br><i>abstype datbind</i><br><i>&lt; whtype typhbind &gt;</i><br><i>with dec end</i><br><i>exception ebind</i><br><i>local dec1 in dec2 end</i><br><i>open unitid1 ... unitidn</i> | értékdeklaráció<br>függvénydeklaráció<br>típusdeklaráció<br>datatype declaration                       |
| <i>typhbind</i> | ::= <i>kivételekkelánc</i><br><i>lokális deklaráció</i><br><i>open-deklaráció,</i><br><i>n ≥ 1</i>                                                                                                                                                                                                                                       | abstype declaration<br>exception declaration<br>local declaration<br>open declaration,<br><i>n ≥ 1</i> |
| <i>valbind</i>  | ::= <i>deci &lt; ; &gt; dec2</i><br><i>infix &lt; d &gt; id1 idn</i>                                                                                                                                                                                                                                                                     | üres deklaráció<br>deklaráció-sorozat                                                                  |
| <i>ebind</i>    | ::= <i>infixr &lt; d &gt; id1 idn</i><br><i>nonfix id1 ... idn</i>                                                                                                                                                                                                                                                                       | infix-direktívá,<br><i>n ≥ 1</i>                                                                       |
| <i>datbind</i>  | ::= <i>pat = exp &lt; and valbind &gt;</i><br><i>rec valbind</i>                                                                                                                                                                                                                                                                         | értékkötés<br>rekurzív kötés                                                                           |
| <i>typhbind</i> | ::= <i>&lt; op &gt; var alpat1 ... alpatn &lt; : ty &gt; = exp1</i><br><i>  &lt; op &gt; var alpat21 ... alpat2n &lt; : ty &gt; = exp2</i><br><i>  ...</i><br><i>  &lt; op &gt; var alpatm1 ... alpatmn &lt; : ty &gt; = expm</i><br><i>&lt; and fealbind &gt;</i>                                                                       | <i>m, n ≥ 1</i>                                                                                        |
| <i>valbind</i>  | ::= <i>tyvarseq tycon = ty &lt; and typhbind &gt;</i>                                                                                                                                                                                                                                                                                    | value binding                                                                                          |
| <i>fealbind</i> | ::= <i>tyvarseq tycon = combind &lt; and datbind &gt;</i>                                                                                                                                                                                                                                                                                | recursive binding                                                                                      |
| <i>combind</i>  | ::= <i>&lt; op &gt; con &lt; of ty &gt; &lt;   combind &gt;</i>                                                                                                                                                                                                                                                                          |                                                                                                        |
| <i>exbind</i>   | ::= <i>&lt; op &gt; excon = op longercon &lt; and exbind &gt;</i>                                                                                                                                                                                                                                                                        |                                                                                                        |

Megjegyzés. *fealbind* fenti definíciójában, ha *var* infix helyzetűnek van deklárálva, akkor vagy meg kell elbírnia az *op* szöveknek, vagy infix helyzetben kell használni. Ez az, teljesen, hogy a közök elején *op var* (*atpat*, *atpat*) helyett (*atpat var atpat*) írható. A zártjelek elhagyhatók, ha *atpat* után közvetlenül *: ty* vagy = áll.

### 18.2.3. Tipuskifejezések

|         |     |                                                                                                            |                                                                                             |
|---------|-----|------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| $ty$    | ::= | $tyour$<br>$\{ < tyrow > \}$<br>$tyseq longcon$<br>$ty_1 * \dots * ty_n$<br>$ty_1 \dashv ty_2$<br>$( ty )$ | típusváltozó<br>rekordtípus<br>típuskonstrukció<br>ennes típus, $n \geq 2$<br>függvénytípus |
| $tyrow$ | ::= | $lab : ty < , tyrow >$                                                                                     | típuskifejezés-sor                                                                          |

### 18.2.4. Minták

|          |     |                                                                                                                                                                                                                   |                                                                                                                                                              |
|----------|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $atpat$  | ::= | $\_$<br>$scan$<br>$< op > var$<br>$< op > longcon$<br>$< op > longcon$<br>$\{ < patrow > \}$<br>$()$<br>$( pat_1 , \dots , pat_n )$<br>$[ pat_1 , \dots , pat_n ]$<br>$\# [ pat_1 , \dots , pat_n ]$<br>$( pat )$ | mindenjel<br>állandó<br>változó<br>adatkonstruktör<br>kivetelkonstruktör<br>rekord<br>nullas<br>ennes, $n \geq 2$<br>lista, $n \geq 0$<br>vektor, $n \geq 0$ |
| $patrow$ | ::= | $\dots$<br>$lab = pat < , patrow >$<br>$lab < :ty> < as pat >$<br>$< , patrow >$                                                                                                                                  | mindenjel<br>mintásor<br>címke mint változó                                                                                                                  |
| $pat$    | ::= | $atpat$<br>$< op > longcon atpat$<br>$< op > longcon atpat$<br>$pat_1 con pat_2$<br>$pat_1 excon pat_2$<br>$pat : ty < :ty> as pat$<br>$< op > var < :ty> as pat$                                                 | atomic pattern<br>adatkonstruktő<br>kivetelkonstruktő<br>infix adatkonstruktő<br>infix kivetelkonstruktő<br>minta típusmegkötéssel<br>réteges minta          |
|          |     |                                                                                                                                                                                                                   | wildcard<br>pattern row<br>label as variable                                                                                                                 |

### 18.2.5. Szintaktikai korlátozások

- Egy  $var$  osztálybeli névre egnél többször nem illeszhető minta. Egy  $lab$  osztálybeli mezőnévre egnél többször nem illeszthető kifejezéssor, mintasor vagy típuskifejezés-sor.
- Egy név csak egyféléképpen köthető le egy  $valbind$ ,  $typbind$ ,  $datbind$  vagy  $exbind$  deklarációján. Ugyanez érvényes az adatkonstruktorkra egy  $datbind$  deklarációján.

- Egy  $tyour$  osztálybeli típusváltozó nem fordulhat elő egnél többször egy  $typarseq$  sorozatban egy  $typbind$  vagy  $datbind$  deklaráció bal oldali  $typarseq$  részében. minden olyan  $tyvar$  osztálybeli típusváltozónak, amelyik előfordul a jobb oldalon, szerepetne kell  $typarseq$ -ben.
- A  $rec$ -et követő minden  $pat = exp$  feréklőtőlben az  $exp$ -nek, szükség esetén zárójelek között,  $fn$   $match$  alakúnak kell lennie, ahol a  $match$ -eklelez egy vagy több típusnegkötés és társható.
- $true, false, nil, ::$  és  $refine$  kapthat új értéket egy  $valbind$ ,  $datbind$  vagy  $exbind$  deklarációból. Az itt név nem kapthat új értéket egy  $datbind$  vagy  $exbind$  deklarációból.

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

A Listsort könyvtárból\* sort, sorted.

A Math könyvtárból\*. real, pi, e, sqrt, sin, cos, tan, atan, asin, acos, atan2, exp, pow, ln, log10, sinh, cosh, tanh.

Az Option könyvtárból\*. Option, getOpt, isSome, valOf, filter, map, app, join, compose, mapPartial, composePartial.

A Random könyvtárból. generator, newGen, random, randomList, range, rangeList.

A Real könyvtárból\*. ~, +, -, \*, /, abs, min, max, sign, compare, fromInt, floor, ceil, trunc, round, <, <=, >, >=, toString, fromString.

A Regex könyvtárból. regex, Regex of string, cflag, eflag, regcomp, tokens, fields, map, app, fold, regexecBool, regexecBool, replace, substitute.

A String könyvtárból\*. string, maxSize, size, sub, substring, extract, concat, ^, str, implode, explode, map, translate, tokens, fields, isPrefix, compare, collate, <, <=, >, >=.

A StringCvt könyvtárból. radix, realfmt, padLeft, padRight.

A Substring könyvtárból. substring (tipus), substring (függvény), extract, all, string, base, isEmpty, sub, size, slice, concat, explode, tokens, fields, isPrefix, compare, collate, drop, dropR, takel, raker.

A TextIO könyvtárból. instream, outputStream, openIn, closeIn, input, inputAll, toMicroseconds, fromSeconds, fromMilliseconds, fromReal, toReal, toTimeString, fromString, +, -, <, <=, >, >=, compare.

A Timer könyvtárból. cpu\_timer, real\_timer, startCPUtimer, totalCPUtimer, checkCPUtimer, startRealTimer, totalRealTimer, checkRealTimer.

A Vector könyvtárból. ~a vector, maxLen, fromList, tabulate, length, sub, extract, concat, app, map, foldl, foldr.

A Word könyvtárból. word, wordSize, orb, andB, xorB, notB, <<, >>, ^, -, \*, div, mod, >, <=, <, >=, compare, min, max, toString, fromString, toInt, toIntX, fromInt.

A Word8 könyvtárból. word, word8, wordSize, orb, andB, xorB, notB, <<, >>, ^, -, \*, div, mod, >, <, >=, <=, compare, min, max, toString, fromString.

Az IntSet könyvtárból. intset, empty, singleton, add, addList, isEmpty, equal, isSubset, member, delete, union, intersection, difference, listItems, app, revApp, foldr, foldl, find.

## A. Függelék

### Válogatás az SML '97 könyvtáraiból

A zárthelyin és a vizsgán a \*-gal megjelölt szakaszokban felsorolt típusok, értékek, jelölések, konstruktorok, kivételek és függvények ismertetői várjuk el.

Belső típusok\*. bool, char, exn, int, `a list, `a option, order, real, string, substring, unit, `a vector, word, word8.

Belső kivételek\*. Bind, Chr, Domain, Div, Fail, Interrupt, Io, Match, Option, Ord, Overflow, Size, Subscript.

Belső függvények a kezdeti könyvezetben\*. `::, ::, @, =, <, >, <=, >=, abs, app, ceil, concat, div, explode, false, floor, foldl, foldr, hd, implode, length, map, mod, not, null, o, ord, print, real, rev, round, size, str, tl, true, trunc, vector.

Csak interaktív módon használható belső függvények\*. compile, load, loadOne, printVal, printDepth, printLength, quit, system, use.

A Binaryset könyvtárból. `item set, empty, singleton, add, addList, isEmpty, equal, isSubset, member, delete, union, intersection, difference, listItems, app, revApp, foldr, foldl, find.

A Bool könyvtárból\*. bool, not, toString, fromString.

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

A General könyvtárból\*. A General könyvtár definíálja a belső típusokról, a belső kivételekről, valamint a belső függvényekről szóló szakaszokban felsorolt neveket.

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

Az IntSet könyvtárból. intset, empty, singleton, add, addList, isEmpty, equal, isSubset, member, delete, union, intersection, difference, listItems, app, revApp, foldr, foldl, find.

Az összefoglalot a Moscow ML 1.44 szignatúrái alapján készítettük.

## A.1. Structure Binaryset

```

(* Sets, implemented by ordered balanced binary trees.

* Modified for Moscow ML 1995-04-22 from SML/NJ lib 0.2 file ordset-sig.sml.
* COPYRIGHT (c) 1993 by AT&T Bell Laboratories.
* See file mosml/copyright.art for details.
* Original implementation due to Stephen Adams, Southampton, UK.
*)

type 'item set

exception NotFound

exception 'item option

val empty : ('item * 'item > order) -> 'item set
val singleton : ('item * 'item > order) -> 'item set
val add : item set * 'item -> 'item set
val addlist : item set * 'item list -> 'item set
val retrieve : item set * 'item -> 'item option
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
val find : ('item -> bool) -> 'item set -> 'item option
val emptyOrd : unit -> 'item set

(* This unit implements sets of ordered elements. Every set is
equipped with an ordering relation; the ordering relation is used
in the representation of the set. The result of combining two sets
(of the same type but with different ordering relations is undefined.
The implementation uses ordered balanced binary trees.

[empty Ord] creates a new empty set with the given ordering relation.

[singleton Ord 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.
)

```

## A.2. Structure Bool

```
type bool = bool

val not : bool -> bool

val toString : bool -> string
val fromString : string -> bool option
val scan : (char, 'a) StringCvt.reader -> (bool, 'a) StringCvt.reader
(*
 [not b] is the logical negation of b.

 [toString b] returns the string "false" or "true" according as b is
 false or true.

 [fromString s] scans a boolean b from the string s, after possible
 initial whitespace (blanks, tabs, newlines). Returns (SOME b) if s
 has a prefix which is either "false" or "true"; the value b is the
 corresponding truth value; otherwise NONE is returned.

 [scan getc src] scans a boolean b from the stream src, using the
 stream accessor getc. In case of success, returns SOME(b, rst)
 where b is the scanned boolean value and rst is the remainder of
 the stream; otherwise returns NONE.
*)

val toLower : char -> char
val toUpper : char -> char
```

## A.3. Structure Char

```
type char = char

val minChar : char
val maxChar : char
val maxOrd : int

val chr : int -> char
val ord : char -> int
val succ : char -> char
val pred : char -> char
(* may raise Chr *)

val isLower : char -> bool
val isUpper : char -> bool
val isDigit : char -> bool
val isAlpha : char -> bool
val isHeadDigit : char -> bool
val isAlphaNum : char -> bool
val isAlphaNum : char -> bool
val isDigit : char -> bool
val isAlpha : char -> bool
val isPrint : char -> bool
val isSpace : char -> bool
val isPunct : char -> bool
val isGraph : char -> bool
val isAscii : char -> bool
val isControl : char -> bool
(* contains "abcdefghijklmnopqrstuvwxyz")
(* contains "BCDEFGHIJKLMNOPQRSTUVWXYZ")
(* contains "0123456789")
(* contains "t\r\n\f")
(* isUpper or else isLower
 (* isDigit or else contains "abcdefABCDEF")
 (* isAlpha or else isDigit
 (* any printable character (incl. "#"))
 (* contains "\t\r\n\f")
 (* printable, not space or alphanumeric
 (* (not isSpace) andalso isPrint
 (* ord c < 128
 (* control character
 (*
 [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)
```

[minChar] is the least character in the ordering <.

[maxChar] is the greatest character in the ordering <.

[maxOrd] is the greatest character code; equals ord(maxChar).

*[chr i]* returns the character whose code is *i*. Raises `Chr` if *i*<0 or *i*>`maxOrd`.

*[ord c]* returns the code of character *c*.

*[succ c]* returns the character immediately following *c*, or raises `Chr` if *c* = `maxChar`.

*[pred c]* returns the character immediately preceding *c*, or raises `Chr` if *c* = `minChar`.

*[isLower c]* returns true if *c* is a lowercase letter (a to z).

*[isUpper c]* returns true if *c* is a uppercase letter (A to Z).

*[isDigit c]* returns true if *c* is a decimal digit (0 to 9).

*[isAlpha c]* returns true if *c* is a letter (lowercase or uppercase).

*[isHexDigit c]* returns true if *c* is a hexadecimal digit (0 to 9 or a to f or A to F).

*[isAlphNum c]* returns true if *c* is alphanumeric (a letter or a decimal digit).

*[isPrint c]* returns true if *c* is a printable character (space or visible).

*[isSpace c]* returns true if *c* is a whitespace character (blank, newline, tab, vertical tab, new page).

*[isGraph c]* returns true if *c* is a graphical character, that is, it is printable and not a whitespace character.

*[isPunct c]* returns true if *c* is a punctuation character, that is, graphical but not alphanumeric.

*[isCntrl c]* returns true if *c* is a control character, that is, if `not (isPrint c)`.

*[isAscii c]* returns true if  $0 \leq \text{ord } c \leq 127$ .

*[toLower c]* returns the lowercase letter corresponding to *c*, if *c* is a letter (a to z or A to Z); otherwise returns *c*.

*[toUpper c]* returns the uppercase letter corresponding to *c*, if *c* is a letter (a to z or A to Z); otherwise returns *c*.

*[contains s c]* returns true if character *c* occurs in the string *s*; false otherwise. The function, when applied to *s*, builds a table and returns a function which uses table lookup to decide whether a given character is in the string or not. Hence it is relatively

expensive to compute `val p = contains s` but very fast to compute `p(c)` for any given character.

*[notContains s c]* returns true if character *c* does not occur in the string *s*; false otherwise. Works by construction of a lookup table in the same way as the above function.

*[fromString s]* attempts to scan a character or ML escape sequence from the string *s*. Does not skip leading whitespace. For instance, `fromString "\\"065"` equals `#"A"`.

*[toString c]* returns a string consisting of the character *c*, if *c* is printable, else an ML escape sequence corresponding to *c*. A printable character is mapped to a one-character string; bell, backspace, tab, newline, vertical tab, form feed, and carriage return are mapped to the two-character strings "\a", "\b", "\t", "\n", "\v", "\f", and "\r"; other characters with code less than 32 are mapped to three-character strings of the form "\~Z", and characters with codes 127 through 255 are mapped to four-character strings of the form "\ddd", where *ddd* are three decimal digits representing the character code. For instance, `toString "#"A"` equals "A"  
`toString "#"\\"` equals "\\\\"  
`toString "#"\\"065"` equals "\\\\"065"  
`toString (chr 0)` equals "\\\\"0"  
`toString (chr 1)` equals "\\\\"A"  
`toString (chr 6)` equals "\\\\"F"  
`toString (chr 7)` equals "\\\\"a"  
`toString (chr 8)` equals "\\\\"b"  
`toString (chr 9)` equals "\\\\"t"  
`toString (chr 10)` equals "\\\\"n"  
`toString (chr 11)` equals "\\\\"v"  
`toString (chr 12)` equals "\\\\"f"  
`toString (chr 13)` equals "\\\\"r"  
`toString (chr 14)` equals "\\\\"m"  
`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 "#"065"` equals "A"

## A.4. Structure General

```

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"

{J compares character codes. That is, c1 < c2 returns true
if ord(c1) < ord(c2), and similarly for <=, >, >=.

[Compare (c1, c2)] returns LESS, EQUAL, or GREATER, according as c1 is
precedes, equals, or follows c2 in the ordering Char.< .
*}

exception Overflow
exception Bind
exception Match
exception Interrupt
exception Subscript
exception Size
exception Fail of string
exception Domain
exception Div
exception IntOverflow

val = : ,>,a * ,>,a -> bool
val <> : ,>,a * ,>,a -> bool

(* Below, numtxt is int, Word.word, Word8.word, real, char, string: *)
val < : numtxt * numtxt -> bool
val <= : numtxt * numtxt -> bool
val > : numtxt * numtxt -> bool
val >= : numtxt * numtxt -> bool

(* Below, realint is int or real:
 val ~ : realint -> realint (* raises Overflow *)
 val abs : realint -> realint (* raises Overflow *)
*)

(* Below, num is int, Word.word, Word8.word, or real:
 val + : num * num -> num (* raises Overflow *)
 val - : num * num -> num (* raises Overflow *)
 val * : num * num -> num (* raises Overflow *)
 val / : real * real -> real (* raises Overflow *)
*)

(* Below, wordint is int, Word.word or Word8.word:
 val div : wordint * wordint -> wordint (* raises Div, Overflow *)
 val mod : wordint * wordint -> wordint (* raises Div *)
*)

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

```

```

val trunc : real -> int (* round towards zero *)
val round : real -> int (* round to nearest even *)
val o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
val ignore : 'a -> unit
val before : 'a * 'b -> 'a
val ! : 'a ref -> 'a
val := : 'a ref * 'a -> unit
val vector : 'a list -> 'a vector
(* Non-standard types and exceptions *)
datatype 'a frag = QUOTE of string | ANTIQUOTE of 'a
exception Io of function : string, name : string, cause : exn
exception graphic_failure of string
exception out_of_memory

```

### A.5. Structure Int

```

type int = int

val precision : int option
val minInt : int option
val maxInt : int option
val ~ : int -> int (* Overflow *)
val * : int * int -> int (* Overflow *)
val div : int * int -> int (* Div, Overflow *)
val mod : int * int -> int (* Div, Overflow *)
val quot : int * int -> int (* Div, Overflow *)
val rem : int * int -> int (* Div, Overflow *)
val + : int * int -> int (* Overflow *)
val - : int * int -> int (* Overflow *)
val > : int * int -> bool
val >= : int * int -> bool
val < : int * int -> bool
val <= : int * int -> bool
val abs : int -> int (* Overflow *)
val min : int * int -> int
val max : int * int -> int
val sign : int -> int
val sameSign : int * int -> bool
val compare : int * int -> order
val toInt : int -> int
val fromInt : int -> int
val toLarge : int -> int
val fromLarge : int -> int
val toString : int -> string
val fromString : string -> int option (* Overflow *)
val scan : StringCvt.reader -> (int, 'a) StringCvt.reader
 -> (char, 'a) StringCvt.reader -> (int, 'a) StringCvt.reader
val fmt : StringCvt.radix -> int -> string
(*

[precision] is SOME n, where n is the number of significant bits in an integer. In Moscow ML n is 31 in 32-bit architectures and 63 in 64-bit architectures.

[minInt] is SOME n, where n is the most negative integer.

[maxInt] is SOME n, where n is the most positive integer.

[~, *, div, mod, +, -, abs] are the usual operations on integers. They raise Overflow if the result is not representable. If q = i div d and r = i mod d then it holds that qd + r = i, where either 0


```

$\leq r < d$  or  $d < r \leq 0$ . Evaluating  $i \text{ div } 0$  or  $i \text{ mod } 0$  raises Overflow. In other words, div rounds towards minus infinity. Note that  $i \text{ div } -1$  raises Overflow if  $i$  is the most negative integer, whereas  $i \text{ mod } -1$  is 0.

$\lfloor quo(i, d) \rfloor$  is the quotient of  $i$  by  $d$ , rounding towards zero (instead of rounding towards minus infinity, as done by div). Evaluating  $quo(i, 0)$  raises Div. Evaluating  $quot(i, -1)$  raises Overflow if  $i$  is the most negative integer.

$\lfloor rem(i, d) \rfloor$  is the remainder for quot. That is, if  $q = \text{quot}(i, d)$  and  $r = \text{rem}(i, d)$  then  $d * q + r = i$  where either  $0 \leq r < d$  or  $d < r \leq 0$ . If made infix, the recommended fixity for quot and rem is infix 7 quot rem

$\lfloor min(x, y) \rfloor$  is the smaller of  $x$  and  $y$ .

$\lfloor max(x, y) \rfloor$  is the larger of  $x$  and  $y$ .

$\lfloor sign x \rfloor$  is  $-1$ ,  $0$ , or  $1$ , according as  $x$  is negative, zero, or positive.

$\langle , \rangle$ ,  $\geq$ ,  $\leq$  are the usual comparisons on integers.

$\lfloor compare(x, y) \rfloor$  returns LESS, EQUAL, or GREATER, according as  $x$  is less than, equal to, or greater than  $y$ .

$\lfloor sameSign(x, y) \rfloor$  is true iff  $\text{sign } x = \text{sign } y$ .

$\lfloor toInt x \rfloor$  is  $x$  (because this is the default int type in Moscow ML).

$\lfloor fromInt x \rfloor$  is  $x$  (because this is the default int type in Moscow ML).

$\lfloor toLarge x \rfloor$  is  $x$  (because this is the largest int type in Moscow ML).

$\lfloor frontLarge x \rfloor$  is  $x$  (because this is the largest int type in Moscow ML).  
 $\lfloor fmt radix i \rfloor$  returns a string representing  $i$ , in the radix (base) specified by radix.

| radix | description                                  | output format |
|-------|----------------------------------------------|---------------|
| BIN   | signed binary<br>(base 2)<br>? [01] +        |               |
| OCT   | signed octal<br>(base 8)<br>? [0-7] +        |               |
| DEC   | signed decimal<br>(base 10)<br>? [0-9] +     |               |
| HEX   | signed hexadecimal (base 16)<br>? [0-9A-F] + |               |

$\lfloor toString i \rfloor$  returns a string representing  $i$  in signed decimal format.  
 Equivalent to (fmt DEC i).

$\lfloor frontString s \rfloor$  returns SOME( $i$ ) if a decimal integer numeral can be scanned from a prefix of string  $s$ , ignoring any initial whitespace;

returns NONE otherwise. A decimal integer numeral must have form, after possible initial whitespace:  
 $[+ -]?[0-9]+$

$\lfloor scan radix getc charsrl \rfloor$  attempts to scan an integer numeral from the character source `charrc`, 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( $i$ ,  $\text{rest}$ ) where  $i$  is the value of the number scanned, and  $\text{rest}$  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   | $[+ -]?[0-9a-fA-F]+$ |

\*)

## A.6. Structure Intset

```

(* Applicative sets of integers.

* Modified for Moscow ML from SML/NJ library version 0.2, which is
* COPYRIGHT (c) 1993 by AT&T Bell Laboratories.
* See file mosml/copyright.art for details.
* Original implementation due to Stephen Adams, Southampton, UK.
*)

type intset

exception NotFound

val empty : intset
val singleton : int -> intset
val add : intset * int -> intset
val addList : intset * int list -> intset
val isEmpty : intset -> bool
val equal : intset * intset -> bool
val isSubset : intset * intset -> bool
val member : intset * int -> bool
val delete : intset * int -> intset
val numItems : intset -> int
val union : intset * intset -> intset
val intersection : intset * intset -> intset
val difference : intset * intset -> intset
val listItems : intset -> int list
val app : (int -> unit) -> intset -> unit
val revapp : (int * 'b -> 'b) -> intset -> 'b
val foldr : (int * 'b -> 'b) -> 'b -> intset -> 'b
val foldl : (int * 'b -> 'b) -> 'b -> intset -> 'b
val find : (int -> bool) -> intset -> int option
(*
 [*] [empty] is the empty set of integers.
)

singleton i is the singleton set containing i.

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

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

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 intersection of 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
set in decreasing order.

[find p s] returns SOME i, where i is an item in s which satisfies
p, if one exists; otherwise returns NONE.

*)

```

## A.7. Structure List

```

type 'a list = 'a list

exception Empty (* Subscript and Size *)

val null : 'a list -> bool (* Empty *)
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 (* Subscript *)

val rev : 'a list -> 'a list (* Subscript *)

val @ : 'a list * 'a list -> 'a list (* Subscript *)

val concat : 'a list * 'a list -> 'a list (* Subscript *)
val revAppend : 'a list * 'a list -> 'a list (* Subscript *)

val app : ('a -> unit) -> 'a list -> unit (* Subscript *)
val map : ('a -> 'b) -> 'a list -> 'b list (* Subscript *)
val mapPartial : ('a -> 'b option) -> 'a list -> 'b list (* Subscript *)

val find : ('a -> bool) -> 'a list -> 'a option (* Subscript *)
val filter : ('a -> bool) -> 'a list -> 'a list (* Subscript *)
val partition : ('a -> bool) -> 'a list -> ('a list * 'a list) (* Subscript *)

val foldr : ('a * 'b -> 'b) -> 'a list -> 'b (* Subscript *)
val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b (* Subscript *)

val exists : ('a -> bool) -> 'a list -> bool (* Subscript *)
val all : ('a -> bool) -> 'a list -> bool (* Subscript *)

val tabulate : int * (int -> 'a) -> 'a list (* Size *)
val getItem : 'a list -> ('a *, 'a list) option (* Size *)

(* [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.

[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 xs] returns the list which is the concatenation of all the lists in xs.
[revAppend(xs, ys)] is equivalent to rev xs @ ys, but more efficient.
[app f xs] applies f to the elements of xs, from left to right.
[map f xs] applies f to each element x of xs, from left to right, and returns the list of f's results.
[mapPartial f xs] applies f to each element x of xs, from left to right, and returns the list of those y's for which f(x) evaluated to SOME y.
[find p xs] applies f to each element x of xs, from left to right until p(x) evaluates to true; returns SOME x if such an x exists otherwise NONE.
[filter p xs] applies p to each element x of xs, from left to right, and returns the sublist of those x for which p(x) evaluated to true.
[partition p xs] applies p to each element x of xs, from left to right, and returns a pair (pos, neg) where pos is the sublist of those x for which p(x) evaluated to true, and neg is the sublist of those for which p(x) evaluated to false.
[foldr op% e xs] evaluates x1 % (x2 % (... % (xn-1) % (xn % e)) ...))
where xs = [x1, x2, ..., xn], and % is taken to be infix.
[foldl op% e xs] evaluates xn % (x(n-1) % (... % (x2 % (x1 % e))))
where xs = [x1, x2, ..., xn-1], and % is taken to be infix.
[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.
[nth(xs, i)] returns the i'th element of xs, counting from 0.

```

*[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=xr::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
```

\*)

**A.8. Structure Listsort**

*[all sort : ('a \* 'a -> order) -> 'a list -> 'a list]*  
*[val sorted : ('a \* 'a -> order) -> 'a list -> bool]*

(\*      *[sort order xs]* sorts the list xs in nondecreasing order, using the given ordering. Uses Richard O'Keefe's smooth applicative merge sort.

*[sorted order xs]* checks that the list xs is sorted in nondecreasing order, in the given ordering.

\*)

## A.9. Structure Math

*[exp x]* is e to the x'th power.

```
type real = real
val pi : real
val e : real
val sqrt : real -> real
val sin : real -> real
val cos : real -> real
val tan : real -> real
val atan : real -> real
val asin : real -> real
val acos : real -> real
val atan2 : real * real -> real
val exp : real -> real
val pow : real * real -> real
val ln : real -> real
val log10 : real -> real
val sinh : real -> real
val cosh : real -> real
val tanh : real -> real
```

(\*)

*[pi]* is the circumference of the circle with diameter 1:  
3.14159265358979323846.

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

*[sqrt x]* is the square root of x. Raises Domain is x < 0.0.

*[sin r]* is the sine of r, where r is in radians.

*[cos r]* is the cosine of r, where r is in radians.

*[tan r]* is the tangent of r, where r is in radians. Raises Domain if r is a multiple of pi/2.

*[atan t]* is the arc tangent of t, in the open interval ] -pi/2, pi/2 [. Raises Domain if abs x > 1.

*[acos t]* is the arc cosine of t, in the closed interval [ 0, pi ]. Raises Domain if abs x > 1.

*[atan2(y, x)]* is the arc tangent of y/x, in the interval ] -pi, pi ], except that atan2(y, 0) = sign y \* pi/2. The quadrant of the result is the same as the quadrant of the point (x, y). Hence sign(cos(atan2(y, x))) = sign x and sign(sin(atan2(y, x))) = sign y.

*[exp x]* is x 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) 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.  
\*)

## A.10. Structure Meta

(\* Functions for use in an interactive Moscow ML session \*)

```

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

val use : string -> unit
val compile : string -> unit
val load : string -> unit
val loadOne : string -> unit
val loaded : unit -> string list
val loadPath : string list ref

val quietdec : bool ref
val verbose : bool ref

val exnName : exn -> string
val exnMessage : exn -> string

val quotation : bool ref
val valuepoly : bool ref

val quit : unit -> 'a
val system : string -> int

```

(\*  
*[printVal e]* prints the value of expression e to standard output exactly as it would be printed at top-level, and returns the value of e. Output is flushed immediately. This function is provided as a simple debugging aid. The effect of *printVal* is similar to that of ‘print’, in Edinburgh ML or Umeåa ML. For string arguments, the effect of SML/NJ print can be achieved by the function *TextIO.print*: *string -> unit*.

*[printDepth]* determines the depth (in terms of nested constructors, records, tuples, lists, and vectors) to which values are printed by the top-level value printer and the function *printVal*. The components of the value whose depth is greater than *printDepth* are printed as ‘#’. The initial value of *printDepth* is 20. This value can be changed at any moment, by evaluating, for example,

```
printDepth := 17;
```

*[printLength]* determines the way in which list values are printed by the top-level value printer and the function *printVal*. If the length of a list is greater than *printLength*, then only the first *printLength* elements are printed, and the remaining elements are printed as ‘...’. The initial value of *printLength* is 200. This value can be changed at any moment, by evaluating, for example,

```
printLength := 500;
```

*[quit ()]* quits Moscow ML immediately

*[installPP pp]* installs the prettyprinter *pp* at type *ty*, provided *pp* has type *ppstream -> ty -> unit*. The type *ty* must be a nullary (parameter-less) type constructor representing a datatype, either built-in (such as *bool*) or user-defined. Whenever a value of type *ty* is about to be printed by the interactive system, or function *printVal* is invoked on an argument of type *ty*, the pretty printer *pp* will be invoked to print it. See library unit *PP* for more information.

*[use "f"]* causes ML declarations to be read from file *f* as if they were entered from the console. A file loaded by *use* may, in turn, evaluate calls to *use*. For best results, use ‘use’ only at top level, or at top level within a used file.

*[compile "U.sig"]* will compile and elaborate the unit signature in file *U.sig*, producing a compiled signature file *U.sig*. During compilation, the compiled signatures of other units will be accessed if they are mentioned in *U.sig*.

*[compile "U.sml"]* will elaborate and compile the unit body in file *U.sml*, producing a bytecode file *U.uo*. If there is an explicit signature *U.sig*, then file *U.sig* must exist, and the unit body must match the signature. If there is no *U.sig*, then an inferred signature file *U.sig* will be produced also. No evaluation takes place. During compilation, the compiled signatures of other units will be accessed if they are mentioned in *U.sml*.

The declared identifiers will be reported if verbose is true (see below); otherwise compilation will be silent. In any case, compilation warnings are reported, and compilation errors about the compilation and raise the exception *Fail* with a string argument. During compilation, the compiled signatures of other units will be accessed if they are mentioned in *U.sml*.

*[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 (.no files), and source files (.sml files). This variable affects the load, loadOne, and use functions. The current directory is always searched first, followed by the directories in loadPath, in order. By default, only the standard library directory is in the list, but if additional directories are specified using option -I, then these directories are prepended to loadPath.

*[quietdec]* when \tt true, turns off the interactive system’s prompt and responses, except warnings and error messages. Useful for writing scripts in SML. The default value is false; can be set to true with the -quietdec command line option.

*[verbose]* determines whether the signature inferred by a call to compile will be printed. The printed signature follows the syntax of Moscow ML signatures, so the output of compile "U.sml" can be edited to subsequently create file U.sig. The default value is ref false.

*[exnName exn]* returns a name for the exception constructor in exn. Never raises an exception itself. The name returned may be that of any exception constructor aliasing with exn. For instance,

```
let exception E1; exception E2 = E1 in exnName E2 end
```

may evaluate to "E1" or "E2".

*[errMessage exn]* formats and returns a message corresponding to exception exn. For the exceptions defined in the SML Basis Library, the message will include the argument carried by the exception.

*[quotation]* determines whether quotations and antiquotations are permitted in declarations entered at top-level and in files compiled with compile. A quotation is a piece of text surrounded by backquote characters ‘`’ and is used to embed object language phrases in ML programs; see the Moscow ML Owner’s Manual for a brief explanation of quotations. When quotation is false, the backquote character is an ordinary symbol which can be used in ML symbolic identifiers. When quotation is \tt true, the backquote character is illegal in symbolic identifiers, and a quotation ‘`a b c’ will be recognized by the parser and evaluated to an object of type ‘`General.frag list’. The default value is ref false.

*[valuepoly]* determines whether the type checker should use ‘value

polymorphism’, making no distinction between imperative (-a) and applicative (-a) type variables, and generalizing type variables only in non-expansive expressions. An expression is non-expansive if it is a variable, a special constant, a function, a tuple or record of non-expansive expressions, a parenthesized or typed non-expansive expression, or the application of an exception or value constructor (other than ref) to a non-expansive expression.

If valuepoly is false, then the type checker will distinguish imperative and applicative type variables, generalizing all applicable type variables, and generalize imperative type variables only in non-expansive expressions. This is the default, required by the 1990 Definition of Standard ML, Section 4.8.

*[system "com"]* causes the command com to be executed by the operating system. If a non-zero integer is returned, this must indicate that the operating system has failed to execute the command. Under MS DOS, the integer returned tends to always equal zero, even when the command fails.

\*)

## A.11. Structure Option

exception *Option*

```
val valOf : 'a option -> 'a
val isSome : 'a option -> bool
val valOf : 'a option -> 'a
val filter : ('a -> bool) -> 'a option
val map : ('a -> 'b) -> 'a option ->'b option
val app : ('a -> unit) -> 'a option -> unit
val join : 'a option option -> 'a option
val compose : ('a -> 'b) * ('c -> 'a option) -> ('c -> 'b option)
val mapPartial: ('a -> 'b option) -> ('a option -> 'b option)
val composePartial: ('a -> 'b option) * ('c -> 'a option) -> ('c -> 'b option)
```

(\* *[getOpt (xopt, d)]* returns x if xopt is SOME x; returns d otherwise.

*[isSome xopt]* returns true if xopt is SOME x; returns false otherwise.

*[valOf xopt]* returns x if xopt is SOME x; raises Option otherwise.

*[filter p x]* returns SOME x if p x is true; returns NONE otherwise.

*[map f xopt]* returns SOME (f x) if xopt is SOME x; returns NONE otherwise.

*[app f xopt]* applies f to x if xopt is SOME x; does nothing otherwise.

*[join xopt]* returns x if xopt is SOME x; returns NONE otherwise.

*[compose (f, g) x]* returns SOME (f y) if g x is SOME y; returns NONE otherwise. It holds that compose (f, g) = map f o g.

*[mapPartial f xopt]* returns f x if xopt is SOME x; returns NONE otherwise. It holds that mapPartial f = join o map f.

*[composePartial (f, g) x]* returns f y if g x is SOME y; returns NONE otherwise. It holds that composePartial (f, g) = mapPartial f o g.

The operators (map, join, SOME) form a monad.

\*)

## A.12. Structure Random

(\* Random number generator 1995-04-23 \*)

```
type generator
val newseed : real -> generator
val neugen : 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

(* Type generator is the abstract type of random number generators,
producing uniformly distributed pseudo-random numbers.

[neugen seed] returns a random number generator with the given seed.
[neugen ()] returns a random number generator, taking the seed from
the system clock.

[random gen] returns a random number in the interval [0..1].
[randomList n, gen] returns a list of n random numbers in the
interval [0..1].

[range (min, max) gen] returns an integral random number in the
range [min, max]. Raises Fail if min > max.

[rangeList (min, max) (n, gen)] returns a list of n integral random
numbers in the range [min, max]. Raises Fail if min > max.
*)
```

### A.13. Structure Real

```

type real = real
exception Div by
and Overflow

[SameSign(x, y)] is true iff sign x = sign y.

[toFloat x] is x.

[fromDefault x] is x.

[fromInt i] is the floating-point number representing integer i.

[floor r] is the largest integer <= r (rounds towards minus infinity).
May raise Overflow.

[ceil r] is the smallest integer >= r (rounds towards plus infinity).
May raise Overflow.

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

[round r] is the integer nearest to r, using the default rounding
mode. NOTE: This isn't the required behaviour: it should round to
nearest even integer in case of a tie. May raise Overflow.

[=(x, y)] is equivalent to x=y in Moscow ML (because of the
absence of NaNs and Infs).

[!= (x, y)] is equivalent to x>y in Moscow ML (because of the
absence of NaNs and Infs).

[?=(x, y)] is false in Moscow ML (because of the absence of NaNs
and Infs).

[fmt spec r] returns a string representing r, in the format
specified by spec (see below). The requested number of digits must
be >= 0 in the SCI and FIX formats and > 0 in the GEN format;
otherwise Size is raised, even in a partial application fmt(spec).

spec description
----- -----
SCI NONE scientific, 6 digits after point %e
SCI (SOME n) scientific, n digits after point %e.n
FIX NONE fixed-point, 6 digits after point %f
FIX (SOME n) fixed-point, n digits after point %f.n
GEN NONE auto choice, 12 significant digits %1.2g
GEN (SOME n) auto choice, n significant digits %n.ng

[toString r] returns a string representing r, with automatic choice
of format according to the magnitude of r.
Equivalent to (fmt (GEN NONE) r).

[toFixedString 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]+)?
```

*fscan 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]+)?
```

\*)

## A.14. Structure Regex

```
(* Moscow ML interface to POSIX 1003.2 regular expressions *)
(* Scan getc charsrc 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]+)?
```

---

```
exception Regex of string
(* A compiled regular expression *)

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 *)
 val regcomp : string -> cflag list -> regex
 val regexec : regex -> eflag list -> string -> substring vector option
 val regexecBool : regex -> eflag list -> string -> bool
 val regexecc : regex -> eflag list -> substring
 val regexeccBool : regex -> eflag list -> substring -> bool
 val regmatch : pat : string, tgt : string -> cflag list
 -> eflag list -> substring vector option
 val regmatchBool : pat : string, tgt : string -> cflag list
 -> eflag list -> bool

 datatype replacer =
 Str of string
 | Sns of int
 | Tr of (string -> string) * int (* The i'th parenthesized group *)
 | Trs of substring vector -> string (* Transformation of all groups *)
 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 POSTIX 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                                                                                                                                                |
|-------------------|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>c</code>    | <code>c</code>                                   | Match the character <code>c</code>                                                                                                                     |
| <code>-</code>    | <code>-</code>                                   | Match any character                                                                                                                                    |
| <code>R*</code>   | <code>R*<br/>R\+<br/>R\S<br/>R\S S<br/>R?</code> | Match R zero or more times<br>Match R one or more times<br>Match R or S<br>Match R or the empty string                                                 |
| <code>Rm</code>   | <code>R\m<br/>R\m,\br\m,n\</code>                | Match R exactly m times<br>Match R at least m times<br>Match R at least m and at most n times                                                          |
| <code>Rm,n</code> | <code>[L]<br/>[^L]</code>                        | Match any character in L<br>Match any character not in L                                                                                               |
| <code>-</code>    | <code>-</code>                                   | Match at string's beginning<br>Match at string's end                                                                                                   |
| <code>\$</code>   | <code>\$<br/>(R)<br/>\d<br/>\ \+<br/>\ +</code>  | Match R as a group; save the group d<br>Match the same as previous group d<br>Match \ --- similarly for [ ] ~ \$<br>Match + --- similarly for [ ? ] () |

卷之三

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

`[recomp pat cflags]` returns a compiled representation of the regular expression pat. Raises Regex in case of failure. The cflags have the following meaning:

- [*Extended*] : compile as POSIX extended regular expression.
- [*Case*] : compile case-insensitive match.
- [*Newline*] : make the newline character `\n` significant, so `\n` 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]`` [Extended]`

Example: Match SML floating-point constant:  
`regcomp ``[+]?[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 match, such as *(ab)* in *"(ab)(cd)"* when matched against the string *"xcdy"*, the corresponding substring is the empty substring at the beginning of the underlying string. For a group that takes part in the match repeatedly, such as the group *(b<sup>+</sup>)* in *"(a(b<sup>+</sup>))\*"* when matched against *"bababb"*, the corresponding substring is the last (rightmost) one matched. The *eflags* have the following meaning:

|                 |                                                       |
|-----------------|-------------------------------------------------------|
| <i>[Notbol]</i> | : do not match <code>^</code> at beginning of string. |
| <i>[Noteol]</i> | : do not match <code>\$</code> at end of string.      |

*[regexecBool regex eflags s]* returns true if some substring of *s* matches *regex*, `false` otherwise. Equivalent to, but faster than, `Option.isSome(regexexec eflags s)`.

*[regexec 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.

*[regexecBool regex eflags sus]* returns true if some substring of *sus* matches *regex*, `false` otherwise. Equivalent to, but faster than, `Option.isSome(regexexec regexexec eflags sus)`.

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

*[regmatchBool pat, tgt cflags eflags]* is equivalent to  
`RegexexecBool (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 *rexexec* 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(repl)*

A replacer list *repl* evaluates to the concatenation of the results of the replacers. The replacers are applied from left to right.

*[substitute regex f s]* finds the (disjoint) substrings *b1, ..., bn* of *s* matching regex from left to right, and returns the string obtained from *s* by replacing every *bi* by *f(bi)*. Function *f* is applied to the matching substrings from left to right. Raises Regex if it fails to make progress in decomposing *s*. Equivalent to *replace regex [Tr (f, 0)] s*

*[map regex f s]* finds the leftmost substring *b* of *s* matching regex, and replaces 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, 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(a0, fb(vec1, ..., fa(ai, fb(vec1, fa(a0, e)))) ...)* where *veci* is the match vector corresponding to *bi*. Raises Regex if it fails to make progress in decomposing *s*.

If we define the auxiliary functions  
`fun fapp f (x, r) = f x :: r`  
`fun get 1 vec = Substring.string(Vector.sub(vec, i))`

then

```
map regex f s = List.rev (fold regex (#2, fapp f) [] s)
app regex f s = fold regex (ignore, f o #1) () s
fields regex s = List.rev (fold regex (op ::, #2) [] s)
substitute regex f s =
 Substring.concat(List.rev
 (fold regex (op ::, fapp (Substring.all o f o get 0)) [] s))
 *)
```

## A.15. Structure String

```

local

type char = Char.char

in

type string = string
val maxsize : int
val size : string -> int
val sub : string * int * int -> string
val extract : string * int * int option -> string
val concat : string list -> string
val str : string * string -> 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 option (* ML escape sequences *)
val fromCString : string -> string option (* C escape sequences *)
val toCString : string -> string option (* C escape sequences *)

val < : string * string -> bool
val <= : string * string -> bool
val > : string * string -> bool
val >= : string * string -> bool
end

(*
 [string] is the type of strings of characters.
*)

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

[size s] is the number of characters in string s.

[sub(s, i)] is the i'th character of s, counting from zero.
Raises Subscript if i<0 or i>size s.

[substring(s, i, n)] is the string s[i..i+n-1]. Raises Subscript

```

if i<0 or n<0 or i+n>size s. Equivalent to extract(s, i, SOME n).

[extract (s, i, NONE)] is the string s[i..size s-1].  
Raises Subscript if i<0 or i>size s.

[concat ss] is the concatenation of all the strings in ss.  
Raises Size if the sum of their sizes is greater than maxSize.

[s1 ~ s2] is the concatenation of strings s1 and s2.  
Equivalent to CharVector.map f s and to implode (List.map str cs).

[str c] is the string of size one which contains the character c.  
Equivalent to concat (List.map str cs).

[implode s] 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.  
Equivalent to concat (List.map f (explode s)).

[fields p s] returns the list of fields in s, from left to right,  
where a field is a (possibly empty) maximal substring of s not  
containing any delimiter, and a delimiter is a character satisfying p.  
Two tokens may be separated by more than one delimiter, whereas two  
fields are separated by exactly one delimiter. If the only delimiter  
is the character "#!" , then  
"abc || def" contains two tokens: "abc" and "def"  
"abc || def" contains three fields: "abc" and "" and "def"

[isPrefix s1 s2] is true if s1 is a prefix of s2.  
That is, if there exists a string t such that s1 ~ t = s2.

[compare (s1, s2)] does lexicographic comparison, using the  
standard ordering Char.compare on the characters. Returns LESS,  
EQUAL, or GREATER, according as s1 is less than, equal to, or  
greater than s2.

*[collate cmp (s1, s2)]* performs lexicographic comparison, using the given ordering *cmp* on characters.

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

*[toString s]* returns a string corresponding to *s*, with non-printable characters replaced by C escape sequences. Equivalent to String.translate Char.toCString.

*[<], [<=], [=], [>]*, and *[>=]* compare strings lexicographically. Not skip leading whitespace.

*[toCString s]* returns a string corresponding to *s*, with non-printable characters replaced by C escape sequences.

Equivalent to String.translate Char.toCString.

*[<], [<=], [=], [=], [>]*, and *[>=]* compare strings lexicographically.

\*)

## A.16. Structure StringCvt

```
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
```

(\* This structure presents tools for scanning strings and values from functional character streams, and for formatting ML characters and strings containing escape sequences.

A character source reader

```
getc : (char, cs) reader
is used for obtaining characters from a functional character source src of type cs, one at a time. It should hold that
getc src = SOME(c, src')
 if the next character in src
 is c, and src' is the rest of src;
 = NONE
 if src contains no characters
```

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

## A.17. Structure Substring

*[drop 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

$$\text{drop } p \text{ getc src} = \#2 (\text{splitl } p \text{ getc src})$$

*[skipWS getc src]* drops any leading whitespace from src. Equivalent to `drop Char.isSpace`.

*[padLeft c n s]* returns the string s if size s  $\geq$  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  $\geq$  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.

\*)

```
type substring

val substring : string * int * int -> substring
val extract : string * int * int option -> substring
val all : string -> substring
val string : substring -> string
val base : substring -> (string * int * int)

[skipWS getc src] drops any leading whitespace from src.

Equivalent to drop Char.isSpace.

(*
 A substring is an abstract representation of a piece of a string.

 [substring] is the abstract type of substrings of a basestring.

 A substring (s,i,n) is valid if 0 <= i <= i+n <= size s, or
```

---

```
val isEmpty : substring -> bool
val getc : substring -> (char * substring) option
val first : substring -> char option
val trimL : int -> substring -> substring
val trimR : int -> substring -> substring
val sub : substring * int -> char
val size : substring -> int
val slice : substring * int * int option -> substring
val concat : substring list -> string
val explode : substring -> char list
val isPrefix : string -> substring -> bool
val compare : substring * substring -> order
val collate : (char * char -> order) -> substring * substring -> order

val dropL : (char -> bool) -> substring -> substring
val dropR : (char -> bool) -> substring -> substring
val takeL : (char -> bool) -> substring -> substring
val takeR : (char -> bool) -> substring -> substring
val splitL : (char -> bool) -> substring -> substring * substring
val splitR : (char -> bool) -> substring -> substring * substring
val splitAt : substring * int -> substring * substring

val position : string -> substring -> substring * substring

exception Span
val span : substring * substring -> substring

val translate : (char -> string) -> substring -> string

val tokens : (char -> bool) -> substring -> substring list
val fields : (char -> bool) -> substring -> substring list

val foldl : (char * 'a -> 'a) -> 'a -> substring -> 'a
val foldr : (char * 'a -> 'a) -> 'a -> substring -> 'a
val app : (char -> unit) -> substring -> unit

(*
```

A substring is an abstract representation of a piece of a string.

[substring] is the abstract type of substrings of a basestring.

equivalently,  $0 \leq i$  and  $0 \leq n$  and  $i+n \leq \text{size } s$ .

A valid substring  $(s, i, n)$  represents the string  $s[i \dots i+n-1]$ .  
Invariant in the implementation: Any value of type substring is valid.

*[substring(s, i, n)]* creates the substring  $(s, i, n)$ , consisting of the substring of  $s$  with length  $n$  starting at  $i$ . Raises Subscript if  $i < 0$  or  $n < 0$  or  $i+n > \text{size } s$ . Equivalent to *extract(s, i, SOME n)*.

*[extract(s, i, NONE)]* creates the substring  $(s, i, \text{size } s-i)$  consisting of the tail of  $s$  starting at  $i$ .  
Raises Subscript if  $i < 0$  or  $i > \text{size } s$ .

*[extract(s, i, SOME n)]* creates the substring  $(s, i, n)$ , consisting of the substring of  $s$  with length  $n$  starting at  $i$ .  
Raises Subscript if  $i < 0$  or  $n < 0$  or  $i+n > \text{size } s$ .

*[all s]* is the substring  $(s, 0, \text{size } s)$ .

*[string sus]* is the string  $s[i..i+n-1]$  represented by  $\text{sus} = (s, i, n)$ .

*[base sus]* is the concrete triple  $(s, i, n)$ , where  $\text{sus} = (s, i, n)$ .

*[isEmpty (s, i, n)]* true if the substring is empty (that is,  $n = 0$ ).

*[getc sus]* returns *SOME(c, rst)* where  $c$  is the first character and  $\text{rst}$  the remainder of  $\text{sus}$ , if  $\text{sus}$  is non-empty; otherwise returns *NONE*. Note that

$\#1 \circ \text{valOf } o \circ \text{scanFn } \text{Substring.getc}$   
 $\text{valOf } o \circ \text{StringCvt.scanString } \text{scanFn } o \circ \text{Substring.string}$

*[first sus]* returns *SOME c* where  $c$  is the first character in  $\text{sus}$ , if  $\text{sus}$  is non-empty; otherwise returns *NONE*.

*[triml k sus]* returns  $\text{sus}$  less its leftmost  $k$  characters; or the empty string at the end of  $\text{sus}$  if it has less than  $k$  characters.  
Raises Subscript if  $k < 0$ , even in the partial application *triml(k)*.

*[trimr k sus]* returns  $\text{sus}$  less its rightmost  $k$  characters; or the empty string at the beginning of  $\text{sus}$  if it has less than  $k$  characters.  
Raises Subscript if  $k < 0$ , even in the partial application *trimr(k)*.

*[sub (sus, k)]* returns the  $k$ 'th character of the substring; that is,  $s(i+k)$  where  $\text{sus} = (s, i, n)$ . Raises Subscript if  $k < 0$  or  $k > n$ .

*[size (s, i, n)]* returns the size of the substring, that is,  $n$ .

*[slice (sus, i', NONE)]* returns the substring  $(s, i'+i, n-i')$ , where  $\text{sus} = (s, i, n)$ . Raises Subscript if  $i' < 0$  or  $i' > n$ .

*[slice (sus, i', SOME n')]* returns the substring  $(s, i+i', n')$ , where

$\text{sus} = (s, i, n)$ . Raises Subscript if  $i' < 0$  or  $i'+n' > n$ .

*[concat sus]* returns a string consisting of the concatenation of the substrings. Equivalent to *String.concat (List.map string sus)*.

*[explode sus]* returns the list of characters of  $\text{sus}$ , that is,  
 $[s(i), s(i+1), \dots, s(i+n-1)]$   
where  $\text{sus} = (s, i, n)$ . Equivalent to *String.explode(string sus)*.

*[isPrefix s1 s2]* is true if  $s1$  is a prefix of  $s2$ . That is, if there exists a string  $t$  such that  $\text{string } s1 \sim t = \text{string } s2$ .

*[compare (sus1, sus2)]* performs lexicographic comparison, using the standard ordering *Char.compare* on the characters. Returns *LESS*, *EQUAL*, or *GREATER*, according as  $\text{sus1}$  is less than, equal to, or greater than  $\text{sus2}$ . Equivalent to, but more efficient than,  
*String.compare(string sus1, string sus2)*.

*[collate cmp (sus1, sus2)]* performs lexicographic comparison, using the given ordering *cmp* on characters. Equivalent to, but more efficient than, *String.collate cmp (string sus1, string sus2)*.

*[dropl p sus]* drops the longest prefix (left substring) of  $\text{sus}$  all of whose characters satisfy predicate  $p$ . If all characters do, it returns the empty substring  $(s, i, 0)$  where  $\text{sus} = (s, i, n)$ .

*[dropr p sus]* drops the longest suffix (right substring) of  $\text{sus}$  all of whose characters satisfy predicate  $p$ . If all characters do, it returns the empty substring  $(s, i, 0)$  where  $\text{sus} = (s, i, n)$ .

*[takei p sus]* returns the longest prefix (left substring) of  $\text{sus}$  all of whose characters satisfy predicate  $p$ . That is, if the left-most character does not satisfy  $p$ , returns the empty  $(s, i, 0)$  where  $\text{sus} = (s, i, n)$ .

*[takei' p sus]* returns the longest suffix (right substring) of  $\text{sus}$  all of whose characters satisfy predicate  $p$ . That is, if the right-most character satisfies  $p$ , returns the empty  $(s, i, 0)$  where  $\text{sus} = (s, i, n)$ .

Let  $p$  be a predicate and *xxxxfyyyyzzzz* a string where all characters in *xxxx* and *zzzz* satisfy  $p$ , and *f* a is character not satisfying  $p$ . Then

|                      |                  |
|----------------------|------------------|
| <i>dropl p sus</i> = | <i>yyyyfzzzz</i> |
| <i>dropr p sus</i> = | <i>xxxxfyyjf</i> |
| <i>takel p sus</i> = | <i>xxxx</i>      |
| <i>taker p sus</i> = | <i>zzzz</i>      |

It also holds that

```
concat[takep sus, dropp sus] = string sus
concat[dropp sus, takep sus] = string sus

[splitp sus] splits sus into a pair (sus1, sus2) of substrings
where sus1 is the longest prefix (left substring) all of whose
characters satisfy p, and sus2 is the rest. That is, sus2 begins
with the leftmost character not satisfying p. Disregarding
sideeffects, we have:
splitp sus = (takep sus, dropp sus).
```

[split<sub>p</sub> sus] splits sus into a pair (sus1, sus2) of substrings
where sus2 is the longest suffix (right substring) all of whose
characters satisfy p, and sus1 is the rest. That is, sus1 ends
with the rightmost character not satisfying p. Disregarding
sideeffects, we have:

```
splitp sus = (dropp sus, takep sus)
```

[splitAt<sub>(sus, k)</sub>] returns the pair (sus1, sus2) of substrings,
where sus1 contains the first k characters of sus, and sus2
contains the rest. Raises Subscript if k < 0 or k > size sus.

[positions<sub>(s', i, n)</sub>] splits the substring into a pair (pref, suff)
of substrings, where suff is the longest suffix of (s', i, n) which
has s as a prefix. More precisely, let s' = size s. If there is a
least index k in i..i+n for which s = s'[k..k+m-1],
then the result is pref = (s', i, k-i) and suff = (s', k, n-(k-i));
otherwise the result is pref = (s', i, n) and suff = (s', i+n, 0).

[span<sub>(sus1, sus2)</sub>] returns a substring spanning from the start of
sus1 to the end of sus2, provided this is well-defined; sus1 and
sus2 must have the same underlying string, and the start of sus1
must not be to the right of the end of sus2; otherwise raises Span.

More precisely, if base(sus1) = (s', i, n) and base(sus2) = (s'', i', n')
and s' = s'', and i' <= i + n', then base(join(sus1, sus2)) = (s', i, i' + n' - i'). This may be used to compute 'span', 'union', and 'intersection'.

[translate<sub>f</sub> sus] applies f to every character of sus, from left to
right, and returns the concatenation of the results. Raises Size
if the sum of their sizes is greater than String.maxSize.
Equivalent to String.concat(List.map f (explode sus)).

[tokens<sub>p</sub> sus] returns the list of tokens in sus, from left to right,
where a token is a non-empty maximal substring of sus not containing
any delimiter, and a delimiter is a character satisfying p.

[fields<sub>p</sub> sus] returns the list of fields in sus, from left to right,
where a field is a (possibly empty) maximal substring of sus 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".

[foldl<sub>f</sub> e sus] folds f over sus from left to right. That is,  
 evaluates f(s[i], f(... f(s[i+1] f(s[i+n-1] % e) ...))  
 tail-recursively, where sus = (s, i, n).  
 Equivalent to List.foldl f e (explode sus).

[foldr<sub>f</sub> e sus] folds f over sus from right to left. That is,  
 evaluates f(s[i], f(s[i+1], f(... f(s[i+n-1] % e) ...)))  
 tail-recursively, where sus = (s, i, n).  
 Equivalent to List.foldr f e (explode sus).

[app<sub>f</sub> sus] applies f to all characters of sus, from left to right.  
 Equivalent to List.app f (explode sus).

\*)

## A.18. Structure Susp

(\* Support for lazy evaluation 1995-05-21 \*)

```
type 'a susp
val delay : (unit -> 'a) -> 'a susp
val force : 'a susp -> 'a
```

(\*  
*[`a susp]* is the type of lazily evaluated expressions with  
 result of type *'a*.

*[delay (fn () => e)]* creates a suspension for the expression *e*.  
 The first time the suspension is forced, the expression *e* will be  
 evaluated, and the result stored in the suspension. All subsequent  
 forcing of the suspension will just return this result, so *e* is  
 evaluated at most once. If the suspension is never forced, then *e*  
 is never evaluated.

*[force su]* forces the suspension *su* and returns the result of the  
 expression *e* stored in the suspension.  
 \*)

## A.19. Structure TextIO

```
type elem = Char.char
type vector = string

(* Text input : *)

type inStream

val openIn : string -> inStream
val closeIn : inStream -> unit
val input : inStream -> vector
val inputAll : inStream -> vector
val inputNofBlock : inStream -> vector option
val inputI : inStream -> elem option
val inputN : inStream * int -> vector
val inputLine : inStream -> string
val endInputStream : 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 outputI : outStream * elem -> unit
val outputSubstr : outStream * substring -> unit
val flushOut : outStream -> unit

val stdOut : outStream
val stdErr : outStream

val print : string -> unit
```

(\* This structure provides input/output functions on text streams.  
 The functions are state-based: reading from or writing to a stream  
 changes the state of the stream. The streams are buffered: output  
 to a stream may not immediately affect the underlying file or  
 device.

Note that under DOS, Windows, OS/2, and MacOS, text streams will be  
 ‘translated’ by converting (e.g.) the double newline CRLF to a

single newline character `\n`.

`[instream]` is the type of state-based characters input streams, and type `outstream` is the type of state-based character output streams.

`[elem]` is the type char of characters, and type `vector` is the type of character vectors (strings).

TEXT INPUT:

`[closeIn istr]` closes stream `istr`. Has no effect if `istr` is closed already. Further operations on `istr` will behave as if `istr` is at end of stream (that is, will return `" "` or `NONE` or `true`).

`[input istr]` reads some elements from `istr`, returning a vector `v` of those elements. The vector will be empty (`size v = 0`) if and only if `istr` is at end of stream or is closed. May block (not return until data are available in the external world).

`[inputAll istr]` reads and returns the string `v` of all characters remaining in `istr` up to end of stream.

`[inputNofBlock istr]` returns `SOME(v)` if some elements `v` can be read without blocking; returns `SOME("")` if it can be determined without blocking that `istr` is at end of stream; returns `NONE` otherwise. If `istr` does not support non-blocking input, raises `Io.NonblockingNotSupported`.

`[input1 istr]` returns `SOME(e)` if at least one element `e` of `istr` is available; returns `NONE` if `istr` is at end of stream or is closed; blocks if necessary until one of these conditions holds.

`[inputN(istr, n)]` returns the next `n` characters from `istr` as a string, if that many are available; returns all remaining characters if end of stream is reached before `n` characters are available; blocks if necessary until one of these conditions holds. (This is the behaviour of the ‘`input`’ function prescribed in the 1990 Definition of Standard ML).

`[inputLine istr]` returns one line of text, including the terminating newline character. If end of stream is reached before a newline character, then the remaining part of the stream is returned, with a newline character added. If `istr` is at end of stream or is closed, then the empty string `" "` is returned.

`[endOfStream istr]` returns `false` if any elements are available in `istr`; returns `true` if `istr` is at end of stream or closed; blocks if necessary until one of these conditions holds.

`[lookahead istr]` returns `SOME(e)` where `e` is the next element in the stream; returns `NONE` if `istr` is at end of stream or is closed; blocks if necessary until one of these conditions holds. Does not advance the stream.

`[stdin]` is the buffered state-based standard input stream.

`[scanStream scan istr]` turns the instream `istr` into a character source and applies the scanner ‘`scan`’ to that source. See `StringCvt` for more on character sources and scanners. The Moscow ML implementation currently can backtrack only 512 characters, and raises `Fail` if the scanner backtracks further than that.

TEXT OUTPUT:

`[openOut s]` creates a new outstream associated with the file named `s`. If file `s` does not exist, and the directory exists and is writable, then a new file is created. If file `s` exists, it is truncated (any existing contents are lost).

`[openAppend s]` creates a new outstream associated with the file named `s`. If file `s` does not exist, and the directory exists and is writable, then a new file is created. If file `s` exists, any existing contents are retained, and output goes at the end of the file.

`[closedOut 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`.

`[flushOut ostr]` flushes the outstream `ostr`, so that all data written to `ostr` becomes available to the underlying file or device.

`[stdOut]` is the buffered state-based standard output stream.

`[stdErr]` is the unbuffered state-based standard error stream. That is, it is always kept flushed, so `flushOut(stdErr)` is redundant.

The functions below are not yet implemented:

`[setPosIn(istr, i)]` sets `istr` to the (untranslated) position `i`. Raises `Io.Io` if not supported on `istr`.

`[getPosIn istr]` returns the (untranslated) current position of `istr`. Raises `Io.Io` if not supported on `istr`.

`[endPosIn istr]` returns the (untranslated) last position of `istr`. Because of translation, one cannot expect to read

`endPosIn` `istr` - `getPosIn` `istr`  
from the current position.

`[getPosOut` `ostr`]`] returns the current position in stream ostr.  
Raises Io.Io if not supported on ostr.`

`[setPosOut` `(ostr, i)]` sets the current position in stream to `ostr` to  
i. Raises `Io`.`Io` if not supported on `ostr`.

`[mkInstream` `sistr]` creates a state-based instream from the  
functional instream `sistr`.

`[getInstream` `istr]` returns the functional instream underlying the  
state-based instream `istr`.

`[setInstream` `(istr, sistr)]` redirects `istr`, so that subsequent input  
is taken from the functional instream `sistr`.

`[mkOutstream` `sostr]` creates a state-based outstream from the  
outstream `sostr`.

`[getOutstream` `ostr]` returns the outstream underlying the  
state-based outstream `ostr`.

`[setOutstream` `(ostr, sostr)]` redirects the outstream `ostr` so that  
subsequent output goes to `sostr`.

\*)

## A.20. Structure Time

`eqtype` `time`  
exception `Time`

```
val zeroTime : time
val now : unit -> time
val toSeconds : time -> int
val toMilliseconds : time -> int
val toMicroseconds : time -> int
val fromSeconds : int -> time
val fromMilliseconds : int -> time
val fromMicroseconds : int -> time
val fromReal : real -> time
val toReal : time -> real
val toString : time -> string (* rounded to millisecond precision *)
val fmt : int -> time -> string
val fromString : string -> time option
val scan : (char, 'a) StringCvt.reader -> (time, 'a) StringCvt.reader

val + : time * time -> time
val - : time * time -> time
val < : time * time -> bool
val <= : time * time -> bool
val > : time * time -> bool
val >= : time * time -> bool
val compare : time * time -> order
```

(\*  
`[Time]` is the type of values representing durations as well as absolute  
points in time (which can be thought of as durations since some time zero).

`[zeroTime]` represents the 0-second duration, and the origin of time,  
so `zeroTime + t = t + zeroTime = t` for all `t`.

`[now ()]` returns the point in time at which the application occurs.  
Raises `Time` if `s < 0`.

`[fromMilliseconds ms]` returns the time value corresponding to `ms`  
milliseconds. Raises `Time` if `ms < 0`.

`[fromMicroseconds us]` returns the time value corresponding to `us`  
microseconds. Raises `Time` if `us < 0`.

`[toSeconds t]` returns the number of seconds represented by `t`,

truncated. Raises Overflow if that number is not representable as an int.

*[toMilliSeconds t]* returns the number of milliseconds represented by t, truncated. Raises Overflow if that number is not representable as an int.

*[toMicroSeconds t]* returns the number of microseconds represented by t, truncated. Raises Overflow if t that number is not representable as an int.

*[realToTime r]* converts a real to a time value representing that many seconds. Raises Time if r < 0 or if r is not representable as a time value. It holds that realToTime 0.0 = zeroTime.

*[timeToReal t]* converts a time the number of seconds it represents; hence realToTime and timeToReal are inverses of each other when defined. Raises Overflow if t is not representable as a real.

*[fmt n t]* returns as a string the number of seconds represented by t, rounded to n decimal digits. If n <= 0, then no decimal digits are reported.

*[toString t]* returns as a string the number of seconds represented by t, rounded to 3 decimal digits. Equivalent to (fmt 3 t).

*[fromString s]* returns SOME t where t is the time value represented by the string s of form [*\n*|t]\*[0-9]+(|[0-9]+)?|(|[0-9]+); or returns NONE if s cannot be parsed as a time value.

*[scan getc src]*, where getc is a character accessor, returns SOME (t, rest) where t is a time and rest is rest of the input, or NONE if s cannot be parsed as a time value.

*[t1 + t2]* is the sum of the times t1 and t2. For reals r1, r2 >= 0.0, realToTime r1 + realToTime r2 = realToTime(Real.+ (r1,r2)). Raises Overflow if the result is not representable as a time value.

*[t1 - t2]* is the t1 minus t2, that is, the duration from t2 to t1. Raises Time if t1 < t2 or if the result is not representable as a time value. It holds that t - zeroTime = t.

*[t1 < t2]* asserts that t1 is strictly before t2. Similarly for <=, >, >=. It holds for reals r1, r2 >= 0.0 that realToTime r1 < realToTime r2 iff Real.<(r1, r2)

*[compare(t1, t2)]* returns LESS, EQUAL, or GREATER, according as t1 precedes, equals, or follows t2 in time.

\*)

## A.21. Structure Timer

```

local
 type time = Time.time
 type cpu_timer
 type real_timer

in

 val startCPUtimer : unit -> cpu_timer
 val totalCPUtimer : unit -> cpu_timer
 val checkCPUtimer : cpu_timer -> user : time, sys : time, gc : time

 val startRealTimer : unit -> real_timer
 val totalRealTimer : unit -> real_timer
 val checkRealTimer : real_timer -> time

end

(*
 [cpu_timer] is the type of values measuring the CPU time consumed.
 [real_timer] is the type of values measuring the real time that has passed.

 [startCPUtimer ()] returns a cpu_timer started at the moment of
 the call.

 [totalCPUtimer ()] returns a cpu_timer started at the moment the
 library was loaded.

 [checkCPUtimer tmr] returns user, sys, gc where user is the amount
 of user CPU time consumed since tmr was started, gc is the amount
 of user CPU time spent on garbage collection, and sys is the
 amount of system CPU time consumed since tmr was started. Note
 that gc time is included in the user time. Under MS DOS, user time
 and gc time are measured in real time.

 [startRealTimer ()] returns a real_timer started at the moment of
 the call.

 [totalRealTimer ()] returns a real_timer started at the moment the
 library was loaded.

 [checkRealTimer tmr] returns the amount of real time that has passed
 since tmr was started.

*)

```

## A.22. Structure Vector

```

type 'a vector = 'a vector
val maxLen : int

val fromList : 'a list -> 'a vector
val tabulate : int * (int -> 'a) -> 'a vector

val valLength : 'a vector -> int
val valSub : 'a vector * int -> 'a
val valExtract : 'a vector * int * int option -> 'a vector
val concat : 'a vector list -> 'a vector

val valApp : ('a -> unit) -> 'a vector -> unit
val valMap : ('a -> 'b) -> 'a vector -> 'b vector
val valFoldl : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b
val valFoldr : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b

val valAppi : (int * 'a -> unit) -> 'a vector * int * int option -> unit
val valMapi : (int * 'a -> 'b) -> 'a vector * int * int option -> 'b vector
val valFoldli : (int * 'a * 'b -> 'b) -> 'b -> 'a vector * int * int option -> 'b
val valFoldri : (int * 'a * 'b -> 'b) -> 'b -> 'a vector * int * int option -> 'b

(*
 [ty vector] is the type of one-dimensional, immutable, zero-based
 constant-time-access vectors with elements of type ty.
 Type ty vector admits equality if ty does. Vectors v1 and v2
 are equal if they have the same length and their elements are equal.

 [maxLen] is the maximal number of elements in a vector.

 [fromList xs] returns a vector whose elements are those of xs.
 Raises Size if length xs > maxLen.

 [tabulate(n, f)] returns a vector of length n whose elements
 are f 0, f 1, ..., f (n-1), created from left to right. Raises
 Size if n<0 or n>maxLen.

 [length v] returns the number of elements in v.

 [sub(v, i)] returns the i'th element of v, counting from 0.
 Raises Subscript if i<0 or i>length v.

 [extract(v, i, None)] returns a vector of the elements v[i..length v-1]
 of v. Raises Subscript if i<0 or i>length v.

 [extract(v, i, Some n)] returns a vector which is the concatenation from left
 to right of the vectors in vs. Raises Size if the sum of the
 sizes of the vectors in vs is larger than maxLen
*)

```

194

A FEJEZET VÁLOGATÁS AZ SMI 97 KÖNYVTÁRAIBÓL

computes  $f(i, v[i], f(i+1, v[i+1], \dots, f(len-1, v[len-1], e) \dots))$ .  
Raises Subscript if  $i < 0$  or  $i > \text{length } v$ .

*[app<sub>i</sub> f (v, i, SOME n)]* applies  $f$  to successive pairs  $(j, v[j])$  for  $j=i, i+1, \dots, i+n-1$ . Raises Subscript if  $i < 0$  or  $n < 0$  or  $i+n > \text{length } v$ , or  $i > \text{length } v$ .

*[map<sub>i</sub> f (v, i, NONE n)]* applies  $f$  to successive pairs  $(j, v[j])$  for  $j=i, i+1, \dots, i+n-1$  and returns a new vector (of length  $n$ ) containing the results. Raises Subscript if  $i < 0$  or  $n < 0$  or  $i+n > \text{length } v$ .

*[map<sub>i</sub> f (v, i, NONE)]* applies  $f$  to successive pairs  $(j, v[j])$  for  $j=i, i+1, \dots, \text{len}-1$ , where  $\text{len} = \text{length } v$ , and returns a new vector (of length  $\text{len}-i$ ) containing the results. Raises Subscript if  $i < 0$  or  $i > \text{length } v$ .  
\*)

---

**A.23. Structure Word**

```

type word = word
val wordSize : int

val orb : word * word -> word
val andb : word * word -> word
val xorb : word * word -> word
val notb : word -> word

val << : word * word -> word
val >> : word * word -> word
val ^>> : word * word -> word

val + : word * word -> word
val - : word * word -> word
val * : word * word -> word
val div : word * word -> word
val mod : word * word -> word

val > : word * word -> bool
val < : word * word -> bool
val >= : word * word -> bool
val <= : word * word -> bool
val compare : word * word -> order

val min : word * word -> word
val max : word * word -> word

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

val toLargeWord : word -> word
val toLargeWordX : word -> word (* with sign extension *)
val fromLargeWord : word -> word

val toLargeInt : word -> int
val toLargeIntX : word -> int (* with sign extension *)
val fromLargeInt : int -> word

val toInt : word -> int
val toIntX : word -> int (* with sign extension *)
val fromInt : int -> word

(*
[word] is the type of n-bit words, or n-bit unsigned integers.
[wordSize] is the value of n above. In Moscow ML, n=31 on 32-bit
machines and n=63 on 64-bit machines.
)
```

---

*[orb(w1, w2)]* returns the bitwise ‘or’ of w1 and w2.

*[andb(w1, w2)]* returns the bitwise ‘and’ of w1 and w2.

*[xorb(w1, w2)]* returns the bitwise ‘exclusive or’, or w1 and w2.

*[notb w]* returns the bitwise negation of w.

*[<<(w, k)]* returns the word resulting from shifting w left by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

*[>>(w, k)]* returns the word resulting from shifting w right by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

*[^>>(w, k)]* returns the word resulting from shifting w right by k bits. The bits shifted in are replications of the left-most bit: the ‘sign bit’, so this is an arithmetical shift. Consequently, for k >= wordsize and wordToInt w >= 0 the result is all 0-bits, and for k >= wordSize and wordToInt w < 0 the result is all 1-bits.

To make <, >, and ^>> infix, use the declaration  
infix 5 < > ^>>

*[+, -, \*, div, mod]* represent unsigned integer addition, subtraction, multiplication, division, and remainder, modulus two to wordsize. The operations (i div j) and (i mod j) raise Div when j=0. Otherwise no exceptions are raised.

*[w1 > w2]* returns true if the unsigned integer represented by w1 is larger than that of w2, and similarly for <, >=, <=.

*[compare(w1, w2)]* returns LESS, EQUAL, or GREATER, according as w1 is less than, equal to, or greater than w2 (as unsigned integers).

*[min(w1, w2)]* returns the smaller of w1 and w2 (as unsigned integers).

*[max(w1, w2)]* returns the larger of w1 and w2 (as unsigned integers) specified by radix.

*[fmt radix w]* returns a string representing w, in the radix (base)

| 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-fF]+ |

*[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 chars]* attempts to scan an unsigned numeral from the character source chars, 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 [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.

*[toLargeIndex w]* returns the (signed) integer represented by bit-pattern w.

*[fromLargeInt ij]* returns the word representing integer i.

*[toLocaleWord w]* returns w.

*[toLocaleWordX w]* returns w.

*[fromLocaleWord w]* returns w.

\*)

## A.24. Structure Word8

```

type word = word8
val wordSize : int

val orb : word * word -> word
val andb : word * word -> word
val xorb : word * word -> word
val notb : word -> word

val << : word * Word.word -> word
val >> : word * Word.word -> word
val ^>> : word * Word.word -> word

val + : word * word -> word
val - : word * word -> word
val * : word * word -> word
val div : word * word -> word
val mod : word * word -> word

val > : word * word -> bool
val < : word * word -> bool
val >= : word * word -> bool
val <= : word * word -> bool
val compare : word * word -> order

val min : word * word -> word
val max : word * word -> word

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

val fmt : StringCvt.radix -> word -> string

val.toInt : word -> int
val.toIntX : word -> int (* with sign extension *)
val.fromInt : int -> word

val.tolargeInt : word -> int
val.tolargeIntX : word -> int (* with sign extension *)
val.fromLargeInt : int -> word

val.tolargeWord : word -> Word.word (* with sign extension *)
val.fromLargeWord : Word.word -> word
(* [word] is the type of 8-bit words, or 8-bit unsigned integers in
 the range 0..255.
[wordSize] is 8.
```

*[orb(w1, w2)]* returns the bitwise ‘or’ of w1 and w2.

*[andb(w1, w2)]* returns the bitwise ‘and’ of w1 and w2.

*[notb w]* returns the bitwise negation of w.

*[<<(w, k)]* returns the word resulting from shifting w left by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

*[>>(w, k)]* returns the word resulting from shifting w right by k bits. The bits shifted in are zero, so this is a logical shift. Consequently, the result is 0-bits when k >= wordSize.

*[^>>(w, k)]* returns the word resulting from shifting w right by k bits. The bits shifted in are replications of the left-most bit: the ‘sign bit’, so this is an arithmetical shift. Consequently, for k >= wordSize and wordToInt w >= 0 the result is all 0-bits, and for k >= wordSize and wordToInt w < 0 the result is all 1-bits.

To make <, >, and >> infix, use the declaration:

infix 5 << >>

*[+, -, \*, div, mod]* represent unsigned integer addition, subtraction, multiplication, division, and remainder, modulus 256. The operations (i div j) and (i mod j) raise Div when j = 0. Otherwise no exceptions are raised.

*[w1 > w2]* returns true if the unsigned integer represented by w1 is larger than that of w2, and similarly for <, >=, <.

*[compare(w1, w2)]* returns LESS, EQUAL, or GREATER, according as w1 is less than, equal to, or greater than w2 (as unsigned integers). The operations (i div j) and (i mod j) raise Div when j = 0.

*[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) [0 1]+  |
| OCT   | unsigned octal                 | (base 8) [0-7]+  |
| DEC   | unsigned decimal               | (base 10) [0-9]+ |
| HEX   | unsigned hexadecimal (base 16) | [0-9A-F]+        |

*[toString w]* returns a string representing w in unsigned (\* [word] is the type of 8-bit words, or 8-bit unsigned integers in the range 0..255.

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 | <code>(0w)?[0-1]+</code>              |
| OCT | <code>(0w)?[0-7]+</code>              |
| DEC | <code>(0w)?[0-9]+</code>              |
| HEX | <code>(0wx 0wX 0X [0-9a-fA-F]+</code> |

---

`[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.

\*)