Standard ML

ניר אדר

1. תוכן עניינים

1. תוכן עניינים 2

2. מבוא 4

2.1 שמירה וטעינה של קבצי ML 4
2.2 מאפייני שפת ML 4
2.3 FUNCTIONAL VS. IMPERATIVE 5
2.4 IT וביטויים פשוטים 5

3. תחביר בסיסי 6

3.1 הגדרת קבועים 6
3.2 הערות 6
3.3 פונקציות 7
3.3.1 הגדרת פונקציות 7
3.3.2 קריאה לפונקציה 7
3.3.3 פונקציות כערכים 8
3.3.4 פונקציות כפרמטרים 9
3.3.5 פונקציות כערך מוחזר 9
3.4 מזהים (IDENTIFIERS) בשפת ML 9
3.4.1 שמות חוקיים בשפת ML 10
3.5 סוגים (TYPES) ב-ML 11
3.5.1 הסוג INT 11
3.5.2 הסוג REAL 11
3.5.3 המרות 11
3.5.4 פענוח הסוג 12
3.5.5 בדיקת סוג פולימורפית 12
3.5.6 הסוג BOOL 13
3.5.7 מחרוזות 13
3.6 מילות המפתח של שפת ML 15
3.7 TUPLES 15
3.8. UNIT 16
3.9 הצהרות לוקליות 17
3.9.1 הצהרות לוקליות בתוך ביטויים 17
3.9.2 הצהרות מקומיות בתוך הצהרות 18
3.9.3 הצהרות הדדיות 18
3.10 אופרטורים בוליאניים 18
3.11 אופרטורים INFIX 19

.3.12	רקורסית זנב	20
.4	רשימות	21
.4.1	מבוא	21
.4.2	יצירת רשימות	21
.4.3	פונקציות בסיסיות	22
.4.4	פעולות על רשימות	22
.4.5	פונקציות ספריה	23
.4.6	רשימות כקבוצות	25
.4.7	מחרוזות ורשימות	26
.5	רשומות	27
.5.1	רשומות	27
.5.2	תבניות רשומות	28
.6	DATA TYPES	29
.6.1	הגדרת סוג חדש	29
.6.2	טיפוסים מורכבים	30
.6.3	POLYMORPHIC DATATYPE	32
.7	EXCEPTIONS – חריגות	35
.7.1	העלאת חריגות	35
.7.2	טיפול בחריגות	36
.7.3	חריגות סטנדרטיות	37
8.	CURRIED FUNCTIONS	38
8.1.	GENERAL-PURPOSE FUNCTIONS	40
.8.1.1	SECTIONS	40
.8.1.2	הרכבה	41
.8.1.3	מיון סוג סדור – דוגמא לקבלת פונקציות כארגומנטים	41
.8.2	דוגמאות לשאלות בנושא	42
.9	SEQUENCES, INFINITE LISTS	43
.10	REFERENCE TYPES – משתני התייחסות	45

2. מבוא

2.1. שמירה וטעינה של קבצי ML

קבצי ML הם בעלי הסיומת sml, שם לדוגמא: myfile.sml.
עריכת קובץ תתבצע על ידי עורך טקסט, ושמירת הקובץ עם הסיומת הנדרשת.

טעינת קובץ מתבצעת:

1. על י די הפונקציה use:

```
use "myfile.sml";
```

2. על ידי הכוונת קלט פלט:

```
sml < myfile.sml > myoutput
```

2.2. מאפייני שפת ML

- שפה פונקציונאלית
- הזיכרון בשפה מנוהל על ידי Garbage Collector
- Strongly typed – איננו יכולים להסתכל כיצד טיפוסים מיוצגים בפועל (בניגוד לשפות כגון C בהן ידוע הייצוג).
- Polymorphism – ניתן לכתוב פונקציות המקבלות פרמטרים כלליים ולאו דווקא סוג אחד קבוע.
- Type inference – כאשר איננו מגדירים סוג במפורש, השפה מנחשת את הסוג.
- Static binding – יודגם בהמשך.

2.3. Functional vs. Imperative

שפה אימפרטיבית משתמשת בפקודות כדי לשנות את מצב התוכנית.

שפה פונקציונלית היא ללא מצבים. אנו משתמשים בביטויים רקורסיביים כדי לחשב את התוצאה.

נציג את ההבדלים על ידי הצגת מימוש הפונקציה GCD בשפת פסקל מול שפת ML.

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

ובשפת ML:

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

2.4. it וביטויים פשוטים

המשתנה it שומר כל הזמן את תוצאות החישוב האחרון שביצענו. לאחר כל פקודה הגורמת לתגובה מצד השפה, ערכו של it ישתנה. לכן, אם נרצה להשתמש ב-it יותר מפעם אחת נשמור את ערכו לתוך משתנה. באופן כללי - ביטויים כאשר לאחריהם נקודה פסיק גורמים לתגובה מצד השפה. כל עוד לא נכתוב נקודה פסיק, האינטרפרטר יחכה להמשך הפקודה. ניתן לבצע חישובים פשוטים, למשל:

2+2;

ML יפלוט את הפלט:

val it = 4 : int

האומר כי it שומר את תוצאת החישוב שזה עתה בוצע.

3. תחביר בסיסי

3.1. הגדרת קבועים

הגדרת קבועים נעשית על ידי הפקודה val, לדוגמא:

```
val seconds = 60;
```

נוכל בהמשך לתת לקבוע ערך חדש, למשל:

```
val seconds = 120;
```

מדוע ערך זה נקרא קבוע ולא משתנה? הרי אנחנו מציבים לו ערך, כמו למשתנה בשפות אחרות.
הסיבה היא שאנחנו מסוגלים גם, למשל, לכתוב את הפקודה הבאה:

```
val seconds = "some string";
```

ולהציב לתוך seconds מחרוזת. השם second אינו מתאים לכתובת בזיכרון, כמו שמשתנים בשפות אחרות מתאימים לכתובת בזיכרון. השם second הוא סימבול שערכו הוא 60.

בהמשך, לצורך נוחות, נכנה את הקבועים בשם משתנים.

3.2. הערות

הערה בשפת ML מתחילה בסימון *) ומסתיימת בסימון (*.

לדוגמא:

```
(* this is a comment *)
```

3.3. פונקציות

3.3.1. הגדרת פונקציות

תחביר:

```
fun <function name> (<formal parameters>) = <body>;
```

לדוגמא:

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

כפי שניתן לראות מהדוגמא, ML מציגה על המסך את סוג הפונקציה (פונקציה הלוקחת מספר שלם ומחזירה מספר שלם).

מילת המפתח **fun** מתחילה את הגדרת הפונקציה. sq זהו **שם הפונקציה**, x הוא **הפרמטר הפורמלי**, **והביטוי** x*x זהו **גוף הפונקציה**.

סוג הפונקציה הינו fn.

התוצאה של הפונקציה היא תוצאת חישוב גוף הפונקציה עם הפרמטר האקטואלי.

ניתן לוותר על הסוגריים בהגדרת הפונקציה, למשל:

```
fun sq x = x*x;
```

ניתן גם לציין במפורש מה הסוג שאנחנו רוצים שהפונקציה תקבל על ידי נקודתיים וציון הסוג.

```
fun sq (x:int) = (* sqaure *) x*x;
```

3.3.2. קריאה לפונקציה

ניתן לקרוא לפונקציה על ידי שמה ולאחר מכן בסוגריים כתיבת הפרמטרים עבורה:

```
sq (3);
> val it = 9 : int
```

כאשר קוראים לפונקציה, ראשית מחושב הפרמטר שלה, ולאחר מכן הבקרה עוברת אל הפונקציה:

```
sq (sq(3));
> val it = 81 : int
```

אם נרצה, נוכל לוותר גם על הסוגריים מסביב לפרמטרים של הפונקציה, לדוגמא:

```
sq 3;
> val it = 9 : int
```

3.3.3. פונקציות כערכים

ML מכילה מושג שלא קיים בהרבה שפות אחרות, שהוא ההתייחסות לפונקציות כערכים.

פונקציה היא ערך לכל דבר. נוכל להגדיר קבועים שיכילו פונקציות. כמו כן נוכל לייצר פונקציות חדשות בהתאם לצורך שלנו.

מנגנון יצירת הפונקציות כולל שימוש ב**פונקציות אנונימיות** – פונקצית שאנחנו מגדירים ללא שם עבור הפונקציה, בעזרת המילה השמורה **fn**.

```
fn x => x*x;
> val it = fn : int -> int
it(3);
> val it = 9 : int
```

ההגדרות הבאות יוצרות אפקט זהה:

```
fun sq x = x*x;
val sq = fn x => x*x
```

הגדרת פונקציות רקורסיביות:

```
fun f(n) = if n=0 then 1 else n * f(n-1);
```

תתעורר בעיה אם נרצה להגדיר פונקציה אנונימית בצורה רקורסיבית מכיוון שאם לפונקציה אין שם, נהיה בבעיה כאשר נבוא להגדירה בצורה רקורסיבית:

```
val f = fn (n) => if n=0 then 1 else n * ??;
```

לצורך פתרון הבעיה, באה המילה השמורה **rec**:

```
val rec f = fn (n) => if n=0 then 1 else n * f(n-1);
```

3.3.4. פונקציות כפרמטרים

ניתן לתת פונקציה כפרמטר לפונקציות אחרות. לדוגמא:

```
fun Sigma(f,x,y) = if x>y then 0 else f(x) + Sigma(f,x+1,y);
> val Sigma = fn : (int -> int) * int * int -> int
Sigma(sq,1,3);
> val it = 14 : int
Sigma(fn x => x*x,1,3);
> val it = 14 : int
```

3.3.5. פונקציות כערך מוחזר

על ידי שימוש בפונקציות אנונימיות ניתן **להחזיר** פונקציות מתוך פונקציות אחרות.

```
fun twice(f:(int->int)) = fn x => f(f(x));
> val twice = fn : (int -> int) -> int -> int
```

משמעות השורה האחרונה: הוחזרה פונקציה המקבלת פונקציה מ-int אל int, ומחזירה פונקציה מ-int אל int.

3.4. מזהים (Identifiers) בשפת ML

שפת ML משתמשת ב-static binding עבור שמות. כפי שאמרנו, שמות הערכים יכונו משתנים. לא ניתן לעדכן ערך של משתנה, אולם שם המשתנה יכול לשמש מאוחר יותר למטרה אחרת. לדוגמא:

```
val x = 5;
fun foo y = x + y;
val x = 10;
foo 5;
```

הפל של תוכנית זו יהיה 10. הערך של x המחושב בפונקציה foo הוא הערך שהיה ל-x ברגע הגדרת הפונקציה.

הערה: מכיוון ש-ML משתמש ב-static binding, נרצה לרוב להדר מחדש את כל התוכנית במידה ואנחנו משנים פונקציה או ערך, מכיוון שפונקציות וערכים שהוגדרו קודם המשתמש בו – יתכן שלא יושפעו מהעדכון שנבצע.

3.4.1. שמות חוקיים בשפת ML

שמות חוקיים בשפת ML:

שמות אלפביתיים

- שמות המתחילים באות, ולאחריה אותיות, מספרים, קו תחתון או מרכאות בודדות.
- דוגמאות:
  - x
  - Ue40
  - Hamlet_Prince_of_Denmark
  - h''3_uhchv
  - or_any_other_name_that_is_as_long_as_you_like

שפת ML הינה case-sensitive, כלומר משתנה הכתוב באותיות גדולות שונה ממשתנה הכתוב באותיות קטנות.

שמות סימבוליים

- שפת ML תומכת גם בשמות סימבוליים. שמות כאלה מכילים רצף כלשהו של הסימנים הבאים:
  ! % & $ # + - * / : < = > ? @ \ ~ ' ^ |
- בשמות אלו ניתן להשתמש בכל מקום בו היינו משתמשים בשם אלפא-ביתי.
- דוגמאות:
  - $^%#%^&
  - #!*<<>>>!~

3.5. סוגים (types) ב-ML

3.5.1. הסוג int

int זהו מספר שלם. טווח הייצוג שלו תלוי בחומרה עליה רץ ML.
מספרים שלמים מיוצגים ב-ML על ידי רצף של ספרות, למשל 3848575. כדי לציין מספר שלילי, נרשום את הסימן ~ לפני המספר, למשל המספר "מינוס שבע" יכתב 7~.

פעולות המותרות על שלמים (כפי שראינו, הפעולות הן infix): +, -, *, div, mod.
הפעולות מקיימות את סדר פעולות החשבון. ניתן להוסיף סוגריים אם רוצים ליצור סדר פעולות אחר.

3.5.2. הסוג real

real הוא סוג המציין מספר עשרוני. גם הגודל שלו תלוי בחומרה.
מספרים עשרוניים מזוהים על ידי נקודה עשרונית, לדוגמא: 0.01, 2.7182812.
ניתן לכתוב גם מספרים עשרוניים על ידי הסימון E. לדוגמא: 123.4E~2 זהה ל-1.234~.
כמו במספרים שלמים, הסימון ~ מסמן מספר שלילי.
האופרטורים הפועלים על ממשיים: +, -, *, /.

פונקציות הפועלות על שלמים כוללות בין השאר את הפונקציות: sqrt, sin, cos, tan, exp, ln. כל פונקציות אלו הינן real -> real. כדי להשתמש בהן נשתמש בקידומת Math ושם הפונקציה, לדוגמא נכתוב Math.sqrt, Math.sin.

3.5.3. המרות

real(i)

ממירה מספר שלם i לערך ממשי מקביל.

floor(r)

ממירה מספר ממשי r לשלם הגדול ביותר הקטן מ-r.

abs(x)

הערך המוחלט של x. יכול להיות גם שלם וגם ממשי.

3.5.4. פענוח הסוג

כאשר אנחנו כותבים ביטוי או פונקציה שפת ML מחליטה בשבילנו באופן אוטומטי מהו הסוג בו אנחנו משתמשים, למשל אם נכתוב את הפונקציה הבאה:

```
fun min(x,y) = if x < y then x else y;
```

שפת ML תחליט כי x,y הינם משתנים מסוג int וכי הערך המוחזר אף הוא מסוג int.

אנחנו יכולים לציין במפורש את הסוג איתו נתעסק, למשל:

```
fun min(x:real, y) = if x < y then x else y;
```

במקרה זה ML תחליט שהפונקציה מקבלת שני real ומחזירה real (זאת מכיוון שאופרטורים למיניהם מחייבים ששני הצדדים יהיו מאותו סוג, ולכן נובע מהפונקציה ש-y בהכרח הוא real, ולכן גם הערך המוחזר.

ניתן לציין כי הפונקציה מחזירה ערך מסוג נקבע (למשל real), על ידי נקודתיים וציון הסוג אחרי שם הפונקציה, למשל:

```
fun min(x,y):real = if x < y then x else y;
```

במידה והערך המוחזר הוא זוג סדור, נצהיר עליו בצורה כזו:

```
fun neg_vec(x,y): real*real = (~x,~y);
> val neg_vec = fn : real * real -> real * real
```

3.5.5. בדיקת סוג פולימורפית

שפות מתחלקות למספר סוגים:

- **Weakly typed languages** - למשל, שפת LISP, אינה בודקת את הסוגים ונותנת למשתמש להכניס למעשה כל ערך לתוך המשתנים.
- **Strongly typed languages** - מגבילות את החופש של המשתמש. לכל משתנה יש סוג. על ידי הגבלת החופש של השמת הערכים בתוך משתנה השפה מבטיחה פחות טעויות.

- **Polymorphic type checking** - בשפת ML ישנה גישה המשלבת את היתרונות של שתי השיטות הקודמות. השפה בודקת את הסוג של כל משתנה, ומספקת את אותה אבטחה ש- strong type checking מספקת. פעולה זו נעשית על ידי שהשפה מנחשת מהו הסוג המתאים לכל משתנה, ובודקת שאין סתירות בשימוש במשתנה זה, לגבי אותו סוג. המשתמש לרוב פטור מציון הסוג בעצמו - השפה מנחשת את הסוג באופן אוטומטי. במידה ובמקום מסוים כל סוג של משתנה יכול להתאים, האובייקט מכונה פולימורפי, והשפה לא מגבילה את הסוג היכול להיות בו.

בשפת ML נסמן משתנים פולימורפיים על ידי 'a, 'b וכדו'. משתנים אלו נקראים משתני סוג (**type variables**). 'a מייצג טיפוס כללי כלשהו. סימון נוסף הקיים בשפה הינו "a, שהוא טיפוס כללי כלשהו שהינו טיפוס בר השוואה.

דוגמא לפונקציה פולימורפית:

```
fun ident x = x;
> val ident = fn : 'a -> 'a
```

3.5.6. הסוג bool

סוג נוסף בשפת ML הוא bool. טיפוס זה מקבל אחד משני ערכים: אמת או שקר (true או false).

דוגמא לפונקציה המקבלת מספר שלם ומחזירה אמת אם הוא זוגי, ושקר אחרת:

```
fun even(x)   = if (x mod 2 = 0) then true else false;
```

דוגמא למימוש של הפונקציה xor בין משתנים בוליאניים:

```
fun xor(x : bool,y : bool)  = not (x = y);
```

3.5.7. מחרוזות

קבועי מחרוזות מוגדרים על ידי מרכאות כפולות, למשל:

```
"ML is the best";
> val it = "ML is the best" : string
```

תווים מיוחדים, עם משמעות זהה לזו של תווים אלו בשפת C, הינם \\ ,\", \n, \t.

שרשור מחרוזות:

```
"Standard" ^ " ML";
> val it = "Standard ML" : string
```

קבלת אורך מחרוזת נעשית על ידי הפקודה size, לדוגמא:

```
size (it);
> val it = 11 : int
```

הגודל של מחרוזת ריקה - "" מוגדר להיות 0.

כדי להבדיל בין מחרוזות של תו בודד לבין תווים בודדים, נשים # לפני מרכאות המייצגות תו בודד, למשל "a"#.

```
"0";
> val it = "0" : string
#"0";
> val it = #"0" : char
```

ממירים בין מחרוזות לתווים על ידי הפונקציות sub ו-str:

```
str(#"0");
> val it = "0" : string
String.sub("hello",0);
> val it = #"h" : char
```

הפונקציה chr מקבלת ערך ASCII והופכת אותו לתו, ואילו הפקודה ord מקבלת תו ומחזירה את ערך ה-ASCII שלו.

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

דוגמא: הפונקציה הבאה מקבלת מחרוזת המייצגת מספר, וממירה את המחרוזת למספר המתאים לה.

```
(* Convert the current char to int, and then call recusively on the
rest of the string *)
fun string_to_int(s) =
      if (size(s) = 0) then 0 else
      ord (String.sub(s, size(s) - 1)) - ord #"0" + 10 *
      string_to_int(String.substring(s, 0, size(s) - 1));
```

דוגמא: הפונקציה הבאה מקבלת תו בודד. במידה והתו הוא אות קטנה, היא ממירה אותו לאות גדולה.
אחרת, התו נשאר ללא שינוי.

```
(* If the ascii value of the character is between 'a' and 'z' make it
uppercase *)
fun char_to_upper(c : char) =
      if ( (ord(c) >= ord #"a") andalso (ord(c) <= ord #"z")) then
            chr(ord(c) - ord #"a" + ord #"A")
      else c;
```

השוואת מחרוזות, על ידי String.compare. הפונקציה מחזירה LESS אם המחרוזת הראשונה מופיעה ראשונה בסדר לקסיקוגרפי, GREATER אם המחרוזת השניה מופיעה ראשונה ו-EQUAL אם המחרוזות שוות. לדוגמא:

```
String.compare("aaa", "bbb");
> val it = LESS : order
```

3.6. מילות המפתח של שפת ML

```
abstype and andalso as case datatype do else end eqtype exception fn
fun functor handle if in include infix infixr let local nonfix of op
open orelse raise rec sharing sig signature struct structure then
type val while with withtype
```

3.7. Tuples

Tuples הינן n-יות סדורות של איברים.
תחביר: (x1, x2, ... ,xn).
רכיבי ה-n-יה יכולים להיות מכל סוג, כולל n-יות אחרות. ל-n-יות כל הזכויות שיש לסוגים הבסיסיים של השפה.
Tuple יכול להיות הן ארגומנט והן ערך מוחזר של פונקציה. אנחנו משתמשים ב-tuples כדי להעביר/להחזיר ערכים רבים מ/אל הפונקציה.

דוגמא לשימוש ב-tuple:

```
val zero_vector = (0.0, 0.0);
```

מדוע בעצם אנחנו צריכים n-יות של איברים? כפי שנראה, ל-n-יות יש שימושים רבים.

ראשית, ווקטור הוא n-יה מסוג real*real. נקודה במרחב היא n-יה מסוג real*real*real.

פונקציה המקבלת יותר מפרמטר אחד, למעשה מקבלת n-יה של איברים, לדוגמא:

```
fun vec_length (x, y) = Math.sqrt(x*x + y*y);
```

שפת ML, בעזרת המילה **type**, מאפשרת לנו להגדיר n-יות מסוימות כסוגים חדשים.

דוגמא לתחביר:

```
type vector = real * real;
type point = real * real * real;
```

לאחר הגדרה כזו נוכל, למשל, לכתוב פונקציה שהסוג שהיא תקבל יהיה מסוג vector או point.

```
fun vec_length ((x, y) : vector) = Math.sqrt(x*x + y*y);
```

Unit .3.8

n-יה באורך אפס מכונה **unity**.ל-unity אין כל רכיבים: ().

הסוג **unit** מסוגל לקבל רק ערך יחיד והוא (). הסימון המתמטי עבור סוג זה הינו UNIT.

נאמר שפונקציה כלשהי הינה **פרוצדורה** אמ"מ התוצאה שלה היא unit.

פונקציה המקבלת unit בתור פרמטרים אינה מעבירה אף מידע לגוף הפונקציה בעת הקריאה.

3.9. הצהרות לוקליות

3.9.1. הצהרות לוקליות בתוך ביטויים

נניח שנרצה לכתוב פונקציה המקבלת שני מספרים ומציגה את השבר המצומצם שהן מייצגות.

הפונקציה תוכל להיכתב כך:

```
fun fraction (n, d) = (n div gcd(n, d), d div gcd(n,d));
```

קיים כאן בזבוז: אנו קוראים פעמים לפונקציה gcd. ניתן למנוע בזבוז זה בצורה הבאה:

```
fun fraction (n, d) =
      let val com = gcd(n, d)
      in (n div com, d div com) end;
```

אנחנו מחשבים את gcd פעם אחת ושומרים את התוצאה ב-com, ולאחר מכן משתמשים רק בתוצאה שחישבנו כבר.

המבנה הכללי של let הינו:

```
let D in E end
```

ראשית D מחושב. הסביבה מחשבת את הביטויים ב-D ושומרת אותם בשמות שנתבקשו. ערכים אלו נראים רק בתוך התחום של ביטוי ה-let. D יכול להיות מורכב מ-1 או יותר ביטויים שיכולים להיות מופרדים ביניהם בפסיקים ויכולים שלא. לאחר החישוב, E מבוצע ומוחזר ערכו.

שפת ML מאפשרת לנו להגדיר פונקציות בתוך פונקציות. הפונקציה הבאה לדוגמא מעלה מספר בריבוע ואז מכפילה אותו ב-2:

```
fun power_and_sum x =
      let
            val pow = x * x;
            fun sum s = s + s
      in
            sum x*x
      end;
```

3.9.2. הצהרות מקומיות בתוך הצהרות

בעזרת מילת המפתח local נוכל להצהיר על פונקציה ומשתנים שיוכרו רק בתחום מסוים עבור הצהרה, למשל:

בניגוד ל-let ששמה התוצאה הייתה חישוב, כאן התוצאה היא הצהרה על פונקציה.

```
local
      fun itfib(n, prev, curr) : int =
            if n = 1 then  curr
            else itfib(n-1, curr, prev+curr)
in
      fun fib(n)  = itfib(n, 0, 1)
end;
```

הפונקציה itfib מוכרת רק ל-fib. זהו למעשה דרך להסתיר את המימוש הפנימי.

3.9.3. הצהרות הדדיות

תחביר:

```
val Id1 = E1 and … and Idn = En
```

הפעולה: הביטויים E1, …, En יחושבו, ורק לאחר מכן יוצבו בתוך Id1 עד Idn. הסדר בו יוצבו אינו ידוע. דוגמא לשימוש, הפיכת ערכים של שמות:

```
val x = y and y = x;
```

3.10. אופרטורים בוליאניים

אופרטורים בוליאנים בהם השפה תומכת:

- אופרטור השלילה: not
- אופרטור "או" – orelse
- אופרטור "וגם" - andalso

not הוא אופרטור אונארי ו-orelse, andalso הם אופרטורים בינאריים. עבור orelse, andalso האופרנד השני מחושב רק אם יש צורך בכך.

3.11. אופרטורים Infix

אופרטור infix הוא פונקציה שנכתבת בין שני הארגומנטים שלה.

כדי לכתוב פונקצית infix נציין ראשית את ההנחיה המורה ל-ML שזוהי פונקצית infix ואז נממש את הפונקציה.

לדוגמא, מימוש XOR:

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

שימוש לדוגמא:

```
true xor false;
```

קדימויות: ML כוללת מנגנון קדימיות, כדי שנוכל לממש סדר פעולות בין האופרטורים השונים בדומה לסדר פעולות חשבון. אופרטור infix יכול להגדיר עבורו עדיפות בין 0 ל-9. ברירת המחדל היא 0, שזוהי הקדימות הנמוכה ביותר. גם לאופרטורים של השפה עצמה מוגדרת עדיפות, למשל, לאופרטור + מוגדרת העדיפות 6.

האופרטור infix גורר אסוציאטיביות לשמאל. במידה ונרצה אסוציאטיביות לימין נשתמש באופרטור infixr.

נדגים שרשור מחרוזות.

```
infix 6 plus;
fun (a plus b) = "(" ^ a ^ "+" ^ b ^ ")";
"1" plus "2" plus "3";
> val it = "((1+2)+3)" : string
```

```
infixr 6 plus2;
fun (a plus2 b) = "(" ^ a ^ "+" ^ b ^ ")";
"1" plus2 "2" plus2 "3";
> val it = "(1+(2+3))" : string
```

הפיכת infix לפונקציות:

שפת ML מאפשרת להתייחס לכל אופרטור infix כפונקציה. אם X הוא אופרטור infix, אזי הפונקציה opX (המוגדרת אוטומטית בשפה) היא הפונקציה המבצעת את אותה פעולה.

3.12. רקורסית זנב

שפת ML היא שפה בה אנו מסתמכים במידה רבה על שימוש ברקורסיות. כידוע משפות אחרות שימוש ברקורסיה לרוב פוגע בביצועי התוכנית. נרצה לגרום לכך שלמרות שאנו משתמשים ברקורסיה התוכנית שלנו תהיה יעילה.
המהדר, כאשר הוא יכול, הופך את הרקורסיות שאנחנו כותבים ללולאות כאשר הוא ממש אותן, ובכך אנו משיגים את השיפור שאנו מעוניינים בו.

כדי שלמהדר יהיה קל לעשות זאת, אנו נעדיף כשאפשר להשתמש ברקורסיה הנקראת רקורסית זנב.
הרעיון של רקורסיה זו: כאשר אנחנו מגיעים לתנאי העצירה של הרקורסיה, יש בידינו את תוצאת החישוב, וכל מה שעלינו לעשות זה להחזיר את התוצאה.

לדוגמא, כך תראה פונקציה לחישוב עצרת ללא רקורסית זנב:

```
fun
      azeret(0) = 1
      | azeret(n) = n * azeret(n-1);
```

כאשר אנחנו מגיעים לתנאי העצירה, אנחנו מתחילים לעלות כדי להחזיר את התוצאה הסופית.

עם רקורסית זנב, הפונקציה תראה כך:

```
local
      fun
            itazeret(0, result) = result
            | itazeret(n, result) = itazeret(n-1, result * n);

in
      fun azeret(n) = itazeret(n, 1)
end;
```

הגדרנו פונקצית עזר, המקבלת כפרמטר נוסף את התוצאה עד כה. כאשר אנו מגיעים לסוף הרקורסיה, אנו פשוט מחזירים אותו.

4. רשימות

4.1. מבוא

רשימה היא רצף סופי של איברים, רשימות לדוגמא הן [3, 5, 9] או ["a", "list"]. הרשימה הריקה מסומנת בשפה על ידי [] ואין לה איברים. על האלמנטים ברשימה מוגדר סדר. כל איבר יכול להופיע פעם אחת או יותר, ללא הגבלה. איברי הרשימה יכולים להיות מכל סוג, כולל רשימת אחרות או tuples.
נשים לב עם זאת כי כל איברי הרשימה חייבים להיות מאותו סוג.
אברי הרשימה מגדירים את סוג הרשימה. נגיד אברי הרשימה הם מסוג τ, אזי הרשימה היא מסוג $\tau - list$. הרשימה הריקה מוגדרת להיות מסוג $\alpha - list$, שזו רשימה היכולה להחזיק כל סוג אברים.

4.2. יצירת רשימות

המילה השמורה nil מסמלת את הרשימה הריקה. כל רשימה בשפת ML היא או הרשימה הריקה, או שהיא מהצורה x :: l כאשר x זהו **ראש** הרשימה ו-l זהו **זנב** הרשימה. זנב הרשימה הוא רשימה בעצמו. הפעולות על רשימה אינן סימטריות מכיוון שקל הרבה יותר לגשת אל ראש הרשימה מאשר אל זנב הרשימה.
הרשימה [3, 5, 9] נבנית בצורה הבאה:

```
(3::(5::(9::nil)))
```

דוגמא: בניית הרשימה: $[m,...,n]$.

```
fun upto(m, n) =
      if (m > n) then [] else m :: upto(m+1, n);
```

נשים לב שלא ניתן לשרשר h :: l כאשר l זוהי רשימה ו-h הוא איבר. ניתן רק לשרשר איבר לראש רשימה, או לחילופין, להשתמש באופרטור שרשור רשימות list @ [element] (נציג אופרטור זה בהמשך).

4.3. פונקציות בסיסיות

הפונקציה null

הפונקציה מקבלת רשימה כפרמטר ומחזירה true אם היא ריקה, ו-false אם לא.

הפונקציה hd

הפונקציה hd מקבלת רשימה לא ריקה ומחזירה את האיבר שהוא ראש הרשימה.

הפונקציה tl

הפונקציה מקבלת רשימה לא ריקה ומחזירה את זנב הרשימה. זנב הרשימה הוא תמיד רשימה.

הפונקציה length

הפונקציה מקבלת רשימה ומחזירה את מספר האיברים שבה.

הפונקציה rev

הפונקציה מקבלת רשימה והופכת את סדר איבריה.

4.4. פעולות על רשימות

באופן דומה ל-tuples ניתן לבצע פעולות גם על רשימות. רשימות יכולות להיות ערך מוחזר או מתקבל של פונקציות. נביט למשל בפונקציה הבאה:

```
fun mult3 [i, j, k] : int = i*j*k;
```

פונקציה זו מקבלת רשימה בה 3 איברים בדיוק ומחזירה את המכפלה שלהם.

לרוב פעולות על רשימות מוגדרות ברקורסיה, המופרדות ל-2 מקרים: מקרה בו הרשימה ריקה ומקרה בו הרשימה איננה ריקה, ואז מבצעים פעולה על הראש וקוראים רקורסיבית לפעולה על הזנב, לדוגמא:

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

דוגמא נוספת: מציאת מקסימום:

```
fun
      max_list [m] : int = m
      | max_list(m :: n :: ns) =
            if (m > n) then max_list(m :: ns)
            else max_list(n :: ns);
```

נשים לב שפונקציה זו איננה מוגדרת עבור הרשימה הריקה. ML תאפשר זאת אולם תוסיף הערה (warning) כאשר נצהיר על פונקציה זו.

דוגמא: הפונקציה take.

הפונקציה מקבלת רשימה ומספר i, ומחזירה את i האיברים הראשונים ברשימה.

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

שרשור רשימות מתבצע על ידי האופרטור @, למשל:

```
fun slalom (l) = l @ rev(l);
```

4.5. פונקציות ספריה

עבור הפונקציות הבאות, נניח כי מוגדר:

```
val l = [1, 2, 3, 4, 5, 6];
```

List.take

הפונקציה מקבלת רשימה ומספר שלם n ומחזירה את n האיברים הראשונים ברשימה, לדוגמא:

```
List.take(l, 3);
> val it = [1,2,3] : int list
```

List.drop

הפונקציה מקבלת רשימה ומספר שלם n ומחזירה את n האיברים האחרונים ברשימה, לדוגמא:

```
List.drop(l, 3);
> val it = [4,5,6] : int list
```

List.concat

הפונקציה מקבלת רשימה של רשימות ומחזירה רשימה של האיברים של רשימות אלה, לדוגמא:

```
List.concat [ [1, 2, 3], [4, 5, 6] ];
> val it = [1,2,3,4,5,6] : int list
```

ListPair.zip

הפונקציה לוקחת שתי רשימות והופכת אותן לרשימת זוגות, כלומר:

$$zip([x_1,...,x_n],[y_1,...,y_n]) = [(x_1,y_1),...,(x_n,y_n)]$$

```
ListPair.zip ([1, 2, 3] , [4, 5, 6]);
> val it = [(1,4),(2,5),(3,6)] : (int * int) list
```

ListPair.unzip

הפונקציה לוקחת רשימה המכילה זוגות איברים ומחזירה זוג רשימות, בצורה:

$$unzip[(x_1,y_1),...,(x_n,y_n)] = ([x_1,...,x_n],[y_1,...,y_n])$$

```
ListPair.unzip [(1,4),(2,5),(3,6)];
> val it = ([1,2,3],[4,5,6]) : int list * int list
```

4.6. רשימות כקבוצות

פונקציה הבודקת האם איבר שייך לקבוצה:

```
infix mem;
fun x mem [] = false
| x mem (y::l) = (x=y) orelse (x mem l);
```

נשים לב כי הפונקציה מוגדרת כאופרטור infix. דוגמא לשימוש בפונקציה:

```
val l = [1 ,2 ,3, 4, 5, 6];
12 mem l;
```

הוספת איבר חדש לקבוצה:

```
fun newmem(x,xs) = if x mem xs then xs else x::xs;
```

איחוד קבוצות:

```
fun
      union([],ys) = ys
      | union(x::xs, ys) = newmem(x, union(xs, ys));
```

חיתוך קבוצות:

```
fun
      inter([],ys) = []
      | inter(x::xs,ys) =
            if x mem ys then x::inter(xs,ys)
            else inter(xs,ys);
```

בדיקת הכלה:

הפונקציה הבאה מקבלת שתי קבוצות ובודקת אם ה-LHS מוכל ב-RHS.

```
infix subs;
fun
      [] subs ys = true
      | (x::xs) subs ys =
            (x mem ys) andalso (xs subs ys);
```

שוויון קבוצות:

```
infix seq;
fun xs seq ys =
      (xs subs ys) andalso (ys subs xs);
```

4.7. מחרוזות ורשימות

בשפת ML הסוג string הוא סוג בסיסי, ואינו מיוצג על ידי רשימה של אותיות.

הפונקציה explode מקבלת מחרוזת ומחזירה רשימה המורכבת מאותיות המחרוזת, למשל:

```
explode "UnderWarrior";
> val it =
[#"U",#"n",#"d",#"e",#"r",#"W",#"a",#"r",#"r",#"i",#"o",#"r"]
```

הפונקציה ההפוכה לה היא implode. פונקציה זו מקבלת רשימה של תווים ומחזירה את המחרוזת המתאימה.

```
implode
([#"U",#"n",#"d",#"e",#"r",#"W",#"a",#"r",#"r",#"i",#"o",#"r"]);
> val it = "UnderWarrior" : string
```

5. רשומות

5.1. רשומות

רשומה היא tuple אשר למרכיביה, הקרויים **שדות**, יש תוויות המייצגות אותם.
היתרון בשימוש ברשומות על tuples הוא שב-tuples כך לטעות בסדר השדות כשיוצרים אותם, ואילו ברשומות הבעיה נפתחת.
לדוגמא, עבור השלשה (Name, Age, Salary) יש הבדל גדול בין (Nir, 22, 20000) ל-(Nir, 20000, 22), אולם קל לטעות ולהחליף ביניהם. לעומת זאת, אם נכתוב:
{ Name = "Nir", Age = 22, Salary = 20000 } או לחילופין נכתוב בצורה הבאה:
{ Name = "Nir", Salary = 20000, Age = 22 }, נקבל את אותה תוצאה.

מבחינה תחבירית, רשומה מוקפת על ידי סוגריים מסולסלים { ... }. כל שדה ברשומה הוא מהצורה label = expression.

לדוגמא:

```
val Pub =
      { BeerPrice       = 10,
        Girls     = "Alot",
        Fun       = true };
```

#label בוחר את הערך של השדה label, לדוגמא:

```
#Girls Pub;
> val it = "Alot" : string
```

זהות בין tuples ל-records: ניתן להתייחס ל-tuple כאל record אשר לתא הראשון קוראים 1#, לתא השני 2# וכו', למשל:

```
#2 (3, 4, 2);
> val it = 4 : int
```

5.2. תבניות רשומות

תבנית רשומה עם שדות מהצורה label = variable מאפשר לנו להגדיר מספר משתנים במכה, כאשר כל משתנה מקבל את הערך של התווית המתאימה., לדוגמא:

```
val { BeerPrice = costs, ... } = Pub;
> val costs = 10 : int
```

הגדרנו שהמשתנה costs יקבל את השדה BeerPrice. שלושת הנקודות מציינות ששאר השדות לא מעניינים אותנו.

6. Data Types

6.1. הגדרת סוג חדש

שפת ML מאפשרת לנו להגדיר סוגים חדשים בשפה. סוגים אלו יוכלו להיות אחר כך משתנים, ערכים המועברים לפונקציות או ערכים המוחזרים מהפונקציות.

הגדרת הסוגים החדשים מתבצעת על ידי מילת המפתח datatype.

הדוגמא הכי פשוטה לשימוש היא זו:

```
datatype color = red | blue | black;
```

בהגדרה זו הגדרנו סוג חדש – בשם color, שהערכים שהוא יכול לקבל הם red, blue או black.

בפעולה זו יצרנו מבנה הדומה ל-enum בשפת C.

דוגמא לפונקציה העושה שימוש בסוג החדש:

```
fun
      f1 (blue, red) = false
      | f1 (i, j) = true;
```

הפונקציה תחזיר false אם הצבע הראשון הוא כחול והשני אדום, אחרת תחזיר true.

דוגמא נוספת היא ההגדרה לטיפוס הבנוי בשפה – bool. טיפוס זה מוגדר כך:

```
datatype bool = false | true;
```

הפונקציה not מוגדרת כך:

```
fun
      not true = false
      | not false = true;
```

6.2. טיפוסים מורכבים

שפת ML מאפשרת לנו להגדיר טיפוסים מורכבים, בדומה ל-union בשפת C, ו-variant record בשפת PASCAL. טיפוס מורכב מסוגל להכיל ערכים מסוגים שונים.

נגדיר למשל את הסוג "צורה". צורה יכולה להיות ריבוע, ואז יש לה אורך ורוחב. צורה יכולה להיות משולש, ואז שלושת אורכי הצלעות מגדירים אותה, וצורה יכולה להיות עיגול, לו יש רדיוס:

```
datatype shape =
      rectangle of int*int
      | triangle of int*int*int
      | circle of int;
```

ההצהרה הזו יוצרת 4 דברים: היא מגדירה את הסוג shape, וכמו כן היא מגדירה 3 **פונקציות בונות (constructors)**, בשם rectangle, triangle ו-circle.

הפונקציות הבונות משמשות כדי ליצור איברים מהסוג אותו יצרנו, למשל:

```
val ribua = rectangle(10, 10);
```

המשתנה ribua יהיה מסוג shape.

משתנים מהסוג החדש הם ערכים לכל דבר, וניתן להשתמש בהם בפונקציות כפרמטרים וערכים מוחזרים. נוכל גם ליצור רשימה של איברים, למשל:

```
val shapes = [ circle 10, circle 20, rectangle(2,3) ];
```

מכיוון שלכל צורה פונקציה בונה ייחודית, ניתן לפצל את הטיפול בצורות השונות. פונקציה המוגדרת על datatype תוגדר בעזרת **תבניות (patterns)** המערבות פונקציות בונות.

דוגמא:

נגדיר datatype לייצוג נמלה. נמלה יכולה להיות מלכה (יחידה) או פועל (בעל שם).

```
datatype ant =
      queen
      | servant of string;

fun
      ant_title (queen) = "The queen"
      | ant_title(servant(name)) = "Ant named " ^ name;

val x = queen;
> val x = queen : ant

val y = servant("moshe");
> val y = servant "moshe" : ant

ant_title(x);
> val it = "The queen" : string

ant_title(y);
> val it = "Ant named moshe" : string
```

דוגמא נוספת:

טיפוס המייצג בן אדם. אדם יכול להיות מלך (יחיד), אביר (בעל שם) או איכר (בעל שם).

```
datatype person =
      King
      | Knight of string
      | Peasant of string;
```

נגדים פונקציה המקבלת שני אנשים ומחזירה האם הראשון במעמד יותר גבוה מהשני. נשים לב לשימוש שנעשה ב-wildcards כאשר אנחנו מעבירים תבניות (patterns) של בנאים.

```
fun
      superior (King, Knight _) = true
      | superior (King, Peasant _) = true
      | superior (Knight _, Peasant _) = true
      | superior _ = false;
```

Polymorphic datatype .6.3

ראינו כי list איננו סוג בשפת ML, אולם list(int) למשל הוא כן סוג.

המילה datatype יכולה לשמש אותנו כדי כאופרטור לייצור סוגים – **type constructor**.

דוגמא היא אופרטור המוגדר כבר בשפת ML, והוא option. הוא מוגדר כך:

```
datatype 'a option = NONE | SOME of 'a;
```

הכוונה: יצרנו *אופרטור סוג* בשם a' option. אופרטור זה ייצר את הסוג הבא לפי דרישה:

סוג היכול להכיל ערך כלשהו ('a) או לחילופין את הערך NONE.

דוגמא: הפונקציה Real.fromString מחזירה ערך מסוג real option:

```
Real.fromString("3421");
> val it = SOME 3421.0 : real option
Real.fromString("abc");
> val it = NONE : real option
```

הערך NONE מוחזר במקרה זה אם הפרמטר איננו מייצג מספר.

דוגמא נוספת: נראה אופרטור המממש disjoint sum (union) המורכב משני סוגי איברים.

```
datatype ('a, 'b)sum =
      In1 of 'a
       | In2 of 'b;
```

הגדרנו אופרטור סוג ושני בנאים, בשם In1, In2. הבנאים מוגדרים להיות:

$$In1:\alpha\to(\alpha,\beta)sum$$
$$In2:\beta\to(\alpha,\beta)sum$$

הסוג $(\alpha,\beta)sum$ הוא למעשה איחוד של שני הסוגים α ו- β. האיחוד (union) מאפשר לנו לקחת ערכים מסוגים שונים ולשים אותם במקום שבו בד"כ מותרים איברים רק מסוג אחד, למשל רשימה.

לדוגמא, אם הסוג הינו sum(int, string) אז כל איבר ברשימה יכיל מספר שלם או מחרוזת, למשל:

```
val x = [In1(3), In2("hi")];
> val x = [In1 3,In2 "hi"] : (int,string) sum list
```

דוגמא

ניצור סוג שהוא מעין רשימה, היכולה להכיל איברים משני סוגים כלשהם:

```
datatype ('a, 'b) two_kinds_list =
      emptylist |
      in1 of 'a * ('a, 'b) two_kinds_list |
      in2 of 'b * ('a, 'b) two_kinds_list;
```

דוגמא לשימוש:

```
in2("33", in1(4, emptylist ));
> val it = in2 ("33",in1 (4,emptylist)) : (int,string) two_kinds_list
```

יצרנו רשימה לה איבר מסוג string ואיבר מסוג int.

דוגמא

נגדיר את הסוג switchlist('a, 'b) שהוא רשימה שבה כל האיברים במקומות האי זוגיים הם מסוג 'a וכל האיברים במקומות הזוגיים הם מסוג 'b. למשל: ["a", 1, "b", 2] : (string, int) switchlist

הסוג יוגדר בצורה הבאה:

```
datatype ('a,'b)switchlist =
      emptylist
        | midnode of 'a * 'b * ('a,'b)switchlist
        | tailnode1 of 'a
        | tailnode2 of 'a * 'b;
```

אנחנו בעצם בכל שלב: או מוסיפים שני איברים – אחד מסוג 'a ואחד מסוג 'b, או מוסיפים איבר שהוא סוף הרשימה, שיכול להיות ריק, או להכיל איבר מסוג 'a בלבד.

שימוש בסוג שהגדרנו כדי לייצר את הרשימה שבדוגמא:

```
midnode("a", 1, tailnode2("b", 2));
```

נדגים כעת שני מימושים של פונקציה, הלוקחת את הרשימה הנ"ל, ומפצלת אותה לשתי רשימות, רשימה אחת מסוג 'a list ורשימה שניה מסוג 'b list.

הפונקציה הראשונה מבצעת שני מעברים על הרשימה. במעבר הראשון היא אוספת את האיברים הזוגיים, ובמעבר השני היא אוספת את האיברים האי זוגיים.

הפונקציה השניה מבצעת את המטלה על ידי מימוש רקורסית זנב ומעבר יחיד על הרשימה.

בשני המקרים, סוג הפונקציה הינו:

```
splitswitch = fn : ('a,'b) switchlist -> 'a list * 'b list
```

המימוש הראשון:

```
local
        fun odd_list(emptylist) = []
          | odd_list(midnode(odd,even,nextnode)) = odd ::
odd_list(nextnode)
          | odd_list(tailnode1(odd)) = odd :: nil
          | odd_list(tailnode2(odd, even)) = odd :: nil;

        fun even_list(emptylist) = []
          | even_list(midnode(odd,even,next)) = even ::
even_list(next)
          | even_list(tailnode1(_)) = nil
          | even_list(tailnode2(odd, even)) = even :: nil;

in
      fun splitswitch (given_list : ('a,'b)switchlist )
            = ( odd_list(given_list) , even_list(given_list) );
end;
```

המימוש השני: (רקורסית זנב)

```
local
      fun
            splitter( midnode (aa , bb , rest) , resA, resB )
                  = splitter(rest, aa :: resA, bb :: resB) |
            splitter (tailnode1 a, resA, resB)
                  = ( a :: resA, resB) |
            splitter (tailnode2 (a, b), resA, resB)
                  = ( a :: resA, b :: resB);
in
      fun
            splitswitch(emptylist) = ( [], [] ) |
            splitswitch(l) = splitter(l, [], []);
end;
```

7. חריגות – Exceptions

פונקציות יכולות להיכשל בפעולתן. למשל, פונקציה המחלקת שני מספרים תכשל אם המספר המחלק הוא 0. כאשר מתרחשת תקלה בפונקציה אנחנו מעוניינים לדווח על כך לזה שקרא לפונקציה.
דרך אחד לדיווח על תקלה, למשל, היא לקבוע כי הפונקציה מחזירה datatype שיכיל ערך אחד אם קיבלנו תוצאה, וערך אחר אם התחרשה שגיאה. פתרון זה עובד, אולם הוא מסרבל את התוכנית.
בדומה לשפות אחרות, שפת ML מכילה מנגנון exceptions כדי לשלוח דיווח על שגיאה שלא דרך מנגנון ההחזרה של הפונקציות.
אני מניח כי אופי המנגנון אינו זר לקורא.

כל החריגות שאנחנו שולחים הינן מהסוג exn. הגדרת חריגות על ידי יוצר הבנאים exception, למשל:

```
exception Failure;
exception DivideByZeroError;
exception Problem of int;
```

הבנאים שיצרנו:

```
Failure : unit -> exn
DivideByZeroError : unit -> exn
Problem : int -> exn
```

שליחת ותפיסת הודעות נעשית על ידי raise, handle. חריגות מפעפעות עד שמישהו תופס אותן.

7.1. העלאת חריגות

העלאת חריגות באופן הבא:

```
raise Ex;
```

אם הביטוי Ex הוא מסוג exn וגם Ex מתפענח לערך e, אזי הפקודה לעיל תיצור Exception Packet המכיל את e. Packets הם אינם ערכים בשפת ML. הפעולות היחידות המותרות עליהם הן: raise, handle.

7.2. טיפול בחריגות

תחביר:

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

לדוגמא: הפונקציה drink מנסה לקרוא לפונקציה beer, ואם הקריאה נכשלת היא קוראת לפונקציה cola:

```
exception AccessDenied;
fun beer(age) =
      if (age < 18) then raise AccessDenied
      else "Beer is good, m'k?";

fun cola(age) = "Cola Rules";

fun drink(age) = beer(age) handle AccessDenied => cola(age);
```

הערה חשובה: ה-handlers חייבים להחזיר ערך מאותו סוג כמו שהפונקציה מחזירה באופן רגיל.
אחרת נקבל compile error.

<u>דוגמא</u>

הפונקציה הבאה מקבלת מספר שלם ומחשבת את העצרת שלו.
חישובי הביניים מועברים על ידי exceptions. זו אינה הדרך המומלצת לפתרון בעיות תיכנותיות, בהן אנו משתמשים ב-exceptions כדי להודיע על שגיאה, אולם היא מציגה את השימוש ב-exceptions ואת תפיסתן.

```
fun e_fact n =
let
      exception E of int;
      fun
            aux_fact(0) = raise E(1)
            | aux_fact(n) = aux_fact(n-1) handle E g => raise E(g*n);
in
      (0 * (aux_fact n)) handle E g => g
end;
```

7.3. חריגות סטנדרטיות

שפת ML מגיעה עם מספר שגיאות המוגדרות מראש:

- Div - נזרק במקרה של חלוקה ב-0.
- Chr - נזרק אם בקריאה ל-chr(k) אזי $k < 0$ או $k > 255$.
- Match - נזרק אם, למשל, הערך שהפונקציה קיבלה הוא לא מהסוג שהיא ציפתה לקבל.
- Bind - נזרק אם לערך E אין את התבנית של P בהצהרה val P = E;
- Subscript - נזרק אם יצאנו מגבולות מערך, מחרוזת או רשימה.

8. Curried Functions

שפת ML מציגה דרך נוספת (ומעוותת) להסתכל על פונקציות המקבלות פרמטרים.

פונקציה ב-ML יכולה לקבל פרמטר בודד בלבד.

כאשר פונקציה מקבלת מספר פרמטרים, יש לנו שתי דרכים להסתכל על כך:

- הדרך האינטואיטיבית: הפונקציה מקבלת למעשה tuple של איברים.
- דרך נוספת: נסתכל על הפונקציה כמחזירה פונקציה בעצמה.

אל הפונקציה $f:(\alpha*\beta)\to\gamma$ נוכל להתייחס כאל $f':\alpha\to(\beta\to\gamma)$.

פונקציה מצורה כזו נקראת Curried Function.

מבחינת תחביר:

Curried Function מוגדרת בצורה הבאה: הגדרת fun שלאחריה מספר ארגומנטים המופרדים על ידי רווחים (ללא סוגריים) מגדירה curried functions.

קריאה לפונקצית curried נעשית על ידי קריאה מהצורה: E E1 E2 … En.

הקריאה מתפענחת בתור: (…((E E1) E2) …) En.

דוגמא: ניקח פונקציה המקבלת שתי מחרוזות ומשרשרת את הראשונה לפני השניה. פונקציה זו איננה curried function.

```
fun regular_prefix(pre, post) = pre ^ post;
```

הטיפוס של הפונקציה הינו

```
> val regular_prefix = fn : string * string -> string
```

ניתן לקרוא לפונקציה כך:

```
regular_prefix("Dr. ", "Yanz");
```

ונקבל כתוצאה את המחרוזת "Dr. Yanz".

נכתוב כעת פונקציה זו כ-curried function:

```
fun regular_prefix2 pre post = pre ^ post;
```

במקרה זה סוג הפונקציה הינו:

```
> val regular_prefix2 = fn : string -> string -> string
```

קיבלנו כאמור פונקציה המחזירה פונקציה. מייד נתעמק בנושא זה.

ניתן לקרוא לפונקציה כך:

```
regular_prefix2 "Dr. " "Yanz";
```

נקבל תוצאה זהה לתוצאה שראינו לעיל. מה בכל זאת ההבדל בין הפונקציות? נביט בפונקציה הבאה:

```
fun prefix pre = fn post => pre ^ post;
```

פונקציה זו מקבלת פרמטר בשם pre ומחזירה פונקציה המקבלת פרמטר בשם post, ומשרשרת את pre עם פרמטר זה. הטיפוס של פונקציה זו הינו:

```
> val prefix = fn : string -> string -> string
```

נשים לב שהוא זהה לזה של regular_prefix2.

קריאה לפונקציה יכולה להיות:

```
val DrPrefix = prefix("Dr. ");
DrPrefix("Nir");
> val it = "Dr. Nir" : string
```

אולם יותר חשוב מכך היא שניתן לקרוא לפונקציה כך:

```
(prefix "Dr. ") "Nir";
```

וגם כך:

```
prefix "Dr. " "Nir";
```

למעשה prefix ו-regular_prefix2 מבצעות את אותה פעולה בדיוק.

כאן אנו חוזרים לנקודת המוצא שלנו – אל כל פונקציה המקבלת מספר פרמטרים ניתן להתייחס כאילו היא פונקציה המקבלת פרמטר בודד ומחזירה פונקציה.

General-Purpose Functions .8.1

השימוש העיקרי שלנו ל-curried function יהיה על מנת ליצור פונקציות כלליות.
Curried functions מאפשרות לנו בעצם ליצור פונקציות אחרות, שישמשו אותנו לביצוע פעולות.
אנחנו נקרא ל-curried function ולא נספק לה את כל הפרמטרים, ואז התוצאה תהיה למעשה פונקציה בה נוכל להשתמש בהמשך.

נגדים כעת מספר General-Purpose Functions לפי רעיון זה.

Sections .8.1.1

הפעלת אופרטור infix על אחד מהאופרנדים, הימני או השמאלי, והשארת האופרנד השני לא מוגדר.

```
fun secl x f y = f(x,y);
> val secl = fn : 'a -> ('a * 'b -> 'c) -> 'b -> 'c

val title = (secl "Dr. " op^);
> val title = fn : string -> string

title "Nir";
> val it = "Dr. Nir" : string
```

כיצד נפענח ונדע להבין את הסוג של secl?

כאמור, מכיוון שאנחנו מניחים כל פעם שיש לנו פונקציה המקבלת פרמטר יחיד ומחזירה פונקציה בעצמה, הרי שאם יש לנו, כמו במקרה זה, 3 פרמטרים + הערך שהפונקציה מחזירה, אז יש לנו פונקציה המחזירה פונקציה המחזירה פונקציה המחזירה עוד פונקציה (פחות או יותר ☺).

בצורה לגמרי טכנית: קיבלנו 3 "פרמטרים", נכתוב 3 חצים:

$$\rightarrow \qquad \rightarrow \qquad \rightarrow$$

כעת, הטיפוס של $f(x,y)$ הינו: $a'*b' \rightarrow c'$, ומכאן: הטיפוס של x הינו a' והטיפוס של y הינו b', ומכאן:

$$\underset{x}{a'} \rightarrow \underbrace{\left(a'*b' \rightarrow c'\right)}_{f} \rightarrow \underset{y}{b'} \rightarrow \underset{\text{returned value of } f(x,y)}{c'}$$

8.1.2. הרכבה

```
infix o;
fun (f o g) x = f(g x);

val sqrt4 = Math.sqrt o Math.sqrt;

sqrt4(16.0);
```

8.1.3. מיון סוג סדור – דוגמא לקבלת פונקציות כארגומנטים

נגדיר את הפונקציה הבאה (curried function) המממיינת רשימה לפי פונקצית השוואה שהיא מקבלת.

```
fun insort lessequal =
      let
            fun ins(x,[]) = [x]
            | ins(x, y::ys)=
                  if lessequal(x,y) then x::y::ys
                  else y::ins(x,ys);

            fun sort [] = []
            | sort(x::xs) = ins(x, sort xs);
      in
            sort
end;
```

סוג הפונקציה:

```
> val insort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
```

דוגמאות לשימוש:

```
insort (op<=) [5,3,7,5,9,8];
> val it = [3,5,5,7,8,9] : int list

insort (op>=) [5,3,7,5,9,8];
> val it = [9,8,7,5,5,3] : int list
```

8.2. דוגמאות לשאלות בנושא

נגדיר את הפונקציה הבאה:

```
fun foo a b = ( (a, b), (b, a) );
```

מה יחזיר הביטוי foo(1, 2.5)?

<u>תשובה שגויה</u>: הערך המוחזר יהיה $\left((1,2.5),(2.5,1)\right)$. הסיבה שתשובה זו שגויה: הפונקציה הינה curried function. העברנו אל הפונקציה tuple שייכנס רק אל האיבר a בעוד b ישאר לא מוגדר, ולכן תשובה זו איננה הנכונה.

<u>התשובה הנכונה</u>: הערך המוחזר יהיה $fn: a' \to \left(\left((\text{int}*\text{real})*a'\right)*\left(a'*(\text{int}*\text{real})\right)\right)$

9. Sequences, Infinite Lists

שפת ML תומכת ב-Lazy Lists, למרות שהשימוש בהן אינו נפוץ בשפה.
Lazy List היא רשימה שאיבריה אינם מחושבים עד לרגע בו התוכנית צריכה אותם בפועל.
בצורה כזו, אנחנו מסוגלים להגדיר רשימות אינסופיות של איברים.
נשים לב לסכנות בשימוש ברשימות כאלה: צורת התכנות המוכרת לנו מתבססת על כך שכל פעולה על הנתונים תסתיים בזמן סופי. במקרה של רשימות אינסופיות, זה איננו בהכרח המקרה.
מותר לנו לבצע פעולה של חיבור שתי רשימות אינסופיות איבר אחרי איבר. אסור לנו לעבור על רשימה אינסופית ולחפש את האיבר הכי קטן בה, למשל.

קיימות שפות כגון Haskell שבהן מבוצע lazy evaluation, ובהן מבנים לוגיים כגון lazy lists הם נפוצים. שפת ML אינה שפת lazy evaluation ולכן כדי ליצור רשימות אינסופיות אנחנו מייצגים את זנב הרשימה כפונקציה, ובכך מעכבים את ה-evaluation שלו.

בדומה לרשימות ב-ML, Sequence היא או ריקה (ואז ערכה Nil) או מכילה ראש וזנב.
הרשימה הריקה היא Nil ולרשימה לא ריקה הצורה Cons(x, xf) כאשר xf זוהי פונקציה המחשבת את זנב הרשימה.

נגדיר את הסוג כך:

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

נכתוב פונקציות שיאפשרו לנו לקבל את ראש הרשימה ואת זנב הרשימה. כאמור, זנב הרשימה עלול (בהתאם לפונקציה היוצרת אותו) להיות אינסופי:

```
exception Empty;
fun
      head(Cons(x,_)) = x
      | head(Nil) = raise Empty;

fun
      tail(Cons(_, xf)) = xf()
      | tail(Nil) = raise Empty;
```

דוגמא: הפונקציה הבאה יוצרת סדרה של טבעיים המתחילה מ-k המתקבל כפרמטר ונמשכת עד אינסוף:

```
fun from k = Cons(k, fn() => from(k+1) );
```

דוגמא לשימוש:

הפונקציה הבאה מקבלת סדרת מספרים שלמים, ומחזירה לנו את האיבר הבא בסדרה המתחלק ב-11.

```
fun div11(x : int seq) =
      if (head(x) mod 11 = 0) then head(x)
      else div11(tail(x));
```

אם נכתוב, למשל,

```
div11(from 12)
```

נקבל את המספר הבא בסדרה הגדול מ-12, כלומר נקבל 22 במקרה זה.

10. משתני התייחסות – reference types

משתני התייחסות הם משתנים השומרים כתובות. הם המקבילים למשתנים של שפת C, פסקל וכדו'.
התפקיד שלהם בשפת ML הוא בעיקר לשמש ליצירת מבני נתונים מקושרים (כגון hash וכדו').
משתני התייחסות אלו מאפשרים תכנות אימפרטיבי בשפת ML.

משתנה התייחסות בשפת ML מייצג כתובת בשטח האכסון של המחשב. לכל כתובת בזיכרון המחשב (store) יש ערך אותו ניתן להחליף. כמו כן, משתנה ההתייחסות בעצמו הוא ערך. אם x הוא מסוג τ אז הכתיבה ref x יוצרת משתנה מסוג $\tau - \text{ref}$.
הבנאי ref יוצר משתני התייחסות. כאשר הוא מופעל על ערך v, הוא מקצה כתובת חדשה שהערך ההתחלתי בה הוא v, ומחזיר משתנה התייחסות לאותה כתובת.
למרות ש-ref הוא פונקציה של שפת ML, זוהי איננה פונקציה במובן המתמטי, מכיוון שבכל פעם ש-ref נקראת מוקצה כתובת חדשה.
הפונקציה !, כאשר מפעילים אותה על משתנה התייחסות, מחזירה את הערך שלו.
ההשמה $E_1 := E_2$ מפענחת את E_1 (החייב להיות משתנה התייחסות) ומעדכנת אותו, על ידי הערך E_2.
מבחינת תחביר, := זוהי פונקציה ו-$E_1 := E_2$ זהו ביטוי. ערך הביטוי המחוזר הוא ().

דוגמא לשימוש:

```
val p = ref 5 and q = ref 2;
> val p = ref 5 : int ref
> val q = ref 2 : int ref

(!p, !q);
> val it = (5,2) : int * int

p := !p + !q;
> val it = () : unit

(!p, !q);
> val it = (7,2) : int * int
```

מכיוון שמשתני התייחסות הם ערכים, הם יכולים להיות שייכים ל-n-יות, רשימות וכדו'.

לדוגמא, בהמשך לדוגמא הקודמת:

```
val refs = [q, p, q];
> val refs = [ref 2,ref 7,ref 2] : int ref list

q := 123;
> val it = () : unit

> refs;
val it = [ref 123,ref 7,ref 123] : int ref list
```

האיבר הראשון והשלישי מייצגים את אותה כתובת כמו q ואילו האיבר השני מכיל את אותה כתובת כמו p.

פעולות נוספות על משתני התייחסות:

שפת ML מאפשרת לנו ליצור משתני התייחסות המתייחסים למשתני התייחסות.

```
val refp = ref p and refq = ref q;
> val refp = ref (ref 7) : int ref ref
> val refq = ref (ref 123) : int ref ref
```

וכעת נוכל לכתוב:

```
!refq := !(!refp);
> val it = () : unit

(!p, !q);
> val it = (7,7) : int * int
```

במקרה זה refp, refq מתפקדים כמו מצביעים בשפות אימפרטיביות.

שיוויון בין משתני התייחסות: האופרטור = מוגדר בין כל סוגי משתני ההתייחסות. שני משתני התייחסות מוגדרים להיות שווים אם הם מתייחסים אל אותה כתובת.