פרל Perl

מדריך למתכנת

מהדורה שניה

נכתב ע"י שמואל פומברג
בתאריך 20.5.2008

http://www.shmuelfomberg.com/perlhebtut/

תוכן עניינים

1.מילות פתיחה

1.1. למה פרל?

פרל (Perl) היא שפה המיועדת במקור לעיבוד טקסטים. נשמע מעט? לא אם זוכרים שכל האינטרנט מורכבת, בעצם, מטקסטים. הוסף לכך פקודות רבות המשותפות עם שפת C, המון מודולי הרחבה שמקלים את החיים וכו', ויש לך נוסחה מנצחת.
דבר נוסף זה זמן הפיתוח. לכתוב תוכנית בפרל (נורמאלית, לא כזו שרק כותבת "שלום") לוקח במקרים רבים עשירית מהזמן שייקח לכתוב תוכנית C מקבילה. פרל רצה, למעשה, תחת כל סביבה, תחת כל מערכת הפעלה. אם אתה צריך להריץ על יוניקס, לינוקס, חלונות, או מקינטוש, תוכל לעשות זאת.

1.2. למי המאמר מיועד?

המאמר מיועד למתחילים, בעלי רקע בסיסי בתכנות, בשפת כלשהי. (כגון שפת C, או Pascal) לא נסביר כאן מהו משתנה, מהו סוג משתנה, מהו מערך, טבלת-האש או פונקציה, מה ההבדל בין קיר לעיר, או כל דבר אחר. אני מניח שאתה כבר עברת על דברים כאלה.
אז בואו נתחיל.

1.3. איך לומדים את השפה הזאת?

קודם כל, יש את המדריך שאתה קורא כרגע. זו כבר התחלה טובה.
בגדול, יש שני מקורות מהם אפשר ללמוד פרל: אינטרנט וספרים.
שמות של ספרים למתחילים אתה יכול למצוא בפרק 8, תחת הכותרת "ספרים". יש כמובן ספרים למתקדמים, וספרים על נושאים ספציפיים בשפה, אבל בוא נשאיר את זה לכשתגיע לרמה הזאת. (בפרק 8 יש עוד כמה הפניות לעזרה, אבל בוא נשאיר את הפרק לזמנו.)
ובאינטנט, יש הרבה מקורות מהם אפשר ללמוד. האתרים המרכזיים הם perl.com, cpan.com ועוד.
ואם אתה מתעקש על חינוך פורמלי, יש קורסים. באתר www.perl.org.il יש הפניות לקורסים בארץ.
ולבסוף, הדבר הכי יעיל ללמידה זה סבלנות ונכונות לניסויים. אם אתה לא מבין את ההסבר על איזו פונקציה, הדרך הכי קלה היא פשוט לנסות אותה. ואז לשנות את הפרמטרים, ולנסות שוב. אם אחרי כמה ניסיונות כאלה עדיין לא הבנת, תשאל אנשים שמבינים, למשל ברשימת התפוצה. (www.perl.org.il)

1.4. מה אני צריך כדי לתכנת בפרל?

השפה נכתבת בכל עורך טקסט פשוט. אם זה פנקס-הרשימות של חלונות, או pico של יוניקס, לא משנה. אפשר גם להשתמש בעורך של שפות אחרות, אם הוא לא מפריע. העיקר שלא יתווספו לקובץ שמתקבל תווי בקרה נוספים. מה שאתה רואה, זה מה אנחנו רוצים לקבל בקובץ.
בשביל להריץ תוכנית פרל, אתה צריך את התוכנה.
בהתקנות יוניקס\לינוקס התוכנה בד"כ כבר מותקנת בברירת מחדל.
עבור חלונות, ניתן להוריד מהאינטרנט בחינם את הגירסה של פרל עבור חלונות המופצת ע"י חברת ActiveState בכתובת:

http://www.activestate.com/Products/ActivePerl/

עבור כל סביבה אחרת, חפש באתר :

http://www.cpan.org/ports

מה שבטוח, זה לא עולה כסף .

1.5. תנאי רישיון

המדריך מופץ תחת GNU FDL, (גירסא 1.2) שזה אומר שמותר לך להדפיס, להפיץ ולשכפל את המדריך, תחת התנאים הבאים: המדריך מופץ בשלמותו, ללא שינויים. מופיע על הכריכה או במקום בולט אחר קישור לדף האינטרנט המקורי של המדריך. (שזה http://www.shmuelfomberg.com/perlhebtut/) אם ברצונך לשים את המדריך על אתרך, נא צור קשר איתי. תודה. (shmuelfomberg@gmail.com)

1.6. אז בואו נתחיל כבר

פתח בעורך החביב עליך קובץ חדש הנקרא, נגיד, first.pl. שים לב לסיומת - לא בכל מקום היא חובה (למשל, יוניקס לא מתחשב בסיומות), אבל תמיד טוב לסמן את הקבצים שלך. כתוב את התוכנית הבאה:

```
#!/usr/bin/perl -w
use strict;
print "Hello World!\n";
```

השורה הראשונה אומרת למחשב שעליו לקרוא לתוכנית פרל (במיקום המצויין בשורה) לתרגם את הקובץ. המתג (switch, ניקרא גם דגל בעיברית) "w-" מורה לפרל לפעול במצב של אזהרות - הקומפילר יזהיר מפני דברים שהוא לא היה אומר אם לא היה את המתג הזה. השורה השניה היא שורת "use". בפרל use מקביל ל-`include, pragma` ב-C ועוד כל מיני. דרך פקודות use אפשר להורות לפרל להשתמש במודולים שונים, להרשות שימוש רק בגירסאות מתקדמות יותר של פרל ממספר גירסה מסויים, או להורות לפרל להכנס למצבים מסוימים. במקרה שלנו, ה-strict אוכף היצמדות לחוקים נוקשים של כתיבת התוכנית, למשל מצב זה אוסר הגדרה ספונטנית של משתנים. נדבר על כך עוד בהמשך. בתור מתחילים (וגם בעתיד בתור מתקדמים) מומלץ *מאוד* להשתמש במתג ה-"w-" וב-"use strict" שכן הם ביחד יתפסו הרבה טעויות שאתם תעשו ויעזרו לכם בדיבוג התוכניות שלכם. השורה השלישית אומרת לכתוב "Hello World", כאשר ה-"\n" בסוף הוא התו המסמן מעבר שורה. פשוט. כדי להריץ ממש את התוכנית הראשונה שלך, כתוב בשורת הפקודה של הטרמינל שאתה עובד איתו:

```
chmod a+x first.pl
./first.pl
```

לא נעבור כאן על הסבר מפורט של פקודות אלו שכן זהו שיעור פרל ולא יוניקס. אם הנך עובד תחת חלונות, אזי בהנחה שהנתיב אל פרל הוא "C:\perl\bin\perl", אזי ההרצה מחלון ה-DOS של חלונות (ניקרא גם חלון הפקודות או ה-Command Prompt) תהיה ע"י הפקודה:

```
C:\perl\bin\perl -w first.pl
```

2.סוגי משתנים והגדרתם

2.1. כללי

בפרל קיימים ארבע סוגי משתנים: סקלר, מערך, טבלת האש וקובץ. הסוגים נבדלים אחד מהשני באמצעות התו שמקדים אותם. לדוגמא: $m הוא סקלר. (אם עכשיו אתה ממלמל "בייסיק", אני לא אתווכח)
קיימים שלושה דרכים להגדיר משתנה: מקומית, גלובלית (עליה נדבר בפרק אחר) וספונטנית.

2.2. הגדרה ספונטנית

הגדרה ספונטנית היא הגדרת משתנה תוך כדי קוד. אם באמצע הקוד תכתוב את השורה:

```
$m=5;
```

אזי פרל תגדיר משתנה חדש, סקלר בשם $m ותשים בתוכו את המספר 5.
היתרון היחיד שאפשר לחשוב עליו זה שהכתיבה חופשית, ולא צריך להגדיר כל משתנה, אלא אפשר להמציא משתנים תוך כדי כתיבה. החסרונות הם שזה מכוער, זה לא קריא, אתה עלול לטעות בכתיב, (שאז פרל ימציא משתנה חדש, ולך תבין מה קרה לערך של המשתנה הקיים) קשה לדעת איפה המשתנה ייהרס ואיפה הוא ימשיך לחיות, ועוד.
בקיצור, אל תעשה את זה. יותר טוב, תמנע את זה מעצמך. תכתוב בתחילת הקובץ:

```
use strict;
```

ואז פרל יהיה מנוע מלהמציא משתנים חדשים. כשתנסה להמציא משתנה, הוא ייעצר (בשלב הקומפילציה) ויגיד שהמשתנה לא קיים.

2.3. הגדרת משתנה מקומי

זה לא טוב להמציא משתנה באמצע קוד. אז איך מצהירים על משתנה? פשוט:

```
my $m;
```

זוהי הצהרה רגילה של סקלר. כמו כל שורת int i ב-C. והיא גם מצייתת לכללים זהים. אם תשים את השורה בתוכנית הראשית, (ולא בתוך בלוק כלשהו) אזי זהו משתנה גלובלי. אם תשים את המשתנה בתוך בלוק, אזי הוא יושמד בסוף הבלוק.
עוד דוגמאות:
דוגמא להגדרה עם הצבה:

```
my $m=5;
```

דוגמא להגדרה של כמה משתנים בבת אחת:

```
my ($m, $s, $gfsd);
```

דוגמא בה מוגדרים שני משתנים, באחד מוצב הערך 5, בשני מוצבת מחרוזת.

```
my ($m, $s)=(5,"Moshe");
```

שים לב שכל שורה מסתיימת בנקודה-פסיק. (;) נקודה-פסיק מסמנת סוף שורה\פקודה בפרל, בדיוק כמו ב-C.

2.4. משתנים סקלריים

בשפות תוכנה אחרות, יש כל מיני סוגי משתנים. שלמים, נקודה-צפה, בוליאניים, תווים ומחרוזות. בפרל קיים רק את הסקלר. (Scalar) הסקלר מכיל את כל המספרים, ואת כל המחרוזות, לא משנה באיזה אורך או רמת דיוק. הסקלר מאופיין ע"י סימן הדולר ($) בתחילתו.
דוגמאות:

```
$m=5;
$m2="Have a nice day";
$zfdg='a';
$fnlpi=3.14157265;
$stam="7";
$van=$stam+$m;
```

שימו לב לאחרון: אני מחבר את המחרוזת "7" עם המספר 5. מה שקורה זה תרגום אוטומטי. אם מנסים להפעיל פעולה אריתמטית על מחרוזת, פרל בודק מה יש בפנים. אם המחרוזת היא בעצם מספר, אזי היא מתורגמת למספר ועליו מתבצעת הפעולה. אם המחרוזת מכילה גם מספר וגם תוים אחריו, פרל בשקט (אלא אם כן השתמשת ב-"w-"!) יוריד את התוים וישאיר את המספר. אם המחרוזת מכילה תוים בלבד, היא תתורגם לערך 0 (אפס).

לייצוג ערכים בוליאניים, בפרל 0 זה שקר, כל דבר אחר זה אמת. מחרוזות - מחרוזת ריקה זה שקר, כל מחרוזת מלאה אחרת זה אמת.

הערה - המחרוזת "0" היא שקר, כיוון שהיא מתורגמת למספר 0, שהוא שקר. אפשר לעקוף את זה, ולהוציא את המחרוזת "0". שים לב לרווח לפני האפס. הרווח מונע מהמחרוזת להיתרגם אוטומטית לאפס, ובבדיקה בוליאנית המחרוזת תהיה שווה לאמת. אם תפעילו על המחרוזת פעולה אריתמטית, המחרוזת תיתרגם לאפס. ככה מוציאים אפס-אמת.

דוגמא לחתיכת קוד חסרת תועלת:

```
#!/usr/bin/perl -w
use strict;
my ($a,$b,$c,$d)=(1,2,3);
$d = $a+$b*$c/$a;
print "And the number is: $d !\n";
```

כמו שאתם כבר מנחשים, כל הפעולות האריתמטיות הרגילות פועלות כרגיל.

הערה - בקטע הקוד הצבנו ערך לתוך $d אחרי ההצהרה. מה היה בתוך $d אחרי ההצהרה ולפני ההצבה?

התשובה היא, שאז היה בתוך $d את הערך undef, או בעברית, לא-מוגדר. זו הדרך של פרל להגן על המתכנת מפני השתמשות במשתנה שלא אותחל. אם תנסה לעשות פעולה כלשהי על undef, תקבל הודעת שגיאה. מצד שני, אפשר להציב undef לתוך סקלר, ואפשר לבדוק האם הערך שבתוך סקלר מוגדר, בעזרת הפונקציה defined.

חתיכת קוד:

```
#!/usr/bin/perl -w
use strict;
my $m;
if (defined($m)) {
    print $m;
} else {
    print "Undefined!";
}
```

לבסוף, שים לב שגם undef בבדיקה בוליאנית הוא שקר.

אופרטורים אריתמטיים:

תוצאה	שם	אופרטור
2 + 3 = 5	חיבור	$a + $b
4 - 3 = 1	חיסור	$a - $b
2 * 3 = 6	כפל	$a * $b
17 % 3 = 2	מודולו (שארית)	$a % $b
8 / 4 = 2	חילוק	$a / $b
2 ** 3 = 8	חזקה	$a ** $b
$a = $a -1	הפחתה-עצמית	$a--, --$a
$a = $a +1	ייסוף-עצמי	$a++, ++$a

2.5. מערכים

אוקי, יש לנו משתנים סקלרים. עכשיו צריך לאגד אותם למערכים. המערך מסומן בשטרודל בתחילתו, (@) ומכיל כל כמות של משתנים סקלרים, ורק משתנים סקלרים. אם תנסה להכניס מערך לתוך מערך, תקבל מערך אחד ארוך. (אפשר להכניס דברים אחרים לתוך מערך, אבל באמצעות מצביעים - מה ששייך לספר אחר)

```
my
@array=(4,5,7,3,3,71,"Samba!",undef,3.14,$m,@old_array,83
);
```
מה שיצא מהדוגמא הזאת, זה מערך אחד גדול, המכיל את מספרים, מחרוזות, ערך שבתוך $m כל הערכים שבתוך @old_array ואפילו undef אחד.
מערכים בפרל הם בעלי גודל משתנה - כלומר, אפשר להוסיף לסופם, לתחילתם או לכל מקום באמצע איברים נוספים, ואפשר גם להוציא איברים.

```
my @arr = (1,2,3);
my $avar = 5;
push (@arr, $avar++); # Add to the end. @arr=(1,2,3,5);
unshift (@arr, $avar++); # Add to the beginning.
@arr=(6,1,2,3,5);
$avar = pop (@arr); # Remove from the end.
@arr=(6,1,2,3), $avar=5.
$avar = shift(@arr); # Remove from the beginning.
@arr=(1,2,3), $avar=6
```

שים לב שהסימן '#' מציין הערה, הנמשכת עד סוף השורה. (שלא כמו ב-C, פה הערות לא יכולות להפרס על כמה שורות, אלא כל הערה היא שורה בודדת)
גישה לערכים בתוך מערך: אפשר לגשת לאיברי המערך בבודדים, או לקבוצה שלמה, בצורה של תת-מערך.

```
$arr[3]=7;
@arr[4,5]=(32,34);
```

שים לב שבדוגמא הראשונה, $arr[3] מתחיל עם סימן דולר, ולא שטרודל. זה כיוון שבדוגמא זו אנו מתייחסים לאיבר ספציפי בתוך המערך, שהוא בפני עצמו סקלר, ולכן הוא מתחיל בסימן דולר. בדוגמא השניה, אני מתחיל בסימן השטרודל, כיוון שאני מתייחס לחתיכה מהמערך, שהיא מערך קטן בפני עצמה, ולכן מגיע לה שטרודל.
אני רוצה להזכיר כאן דוגמא מהסעיף הקודם:

```
my ($a,$b,$c,$d)=(1,2,3);
```

אם תעיף מבט מקרוב, תראה שבעצם מוגדרים פה לא איברים בודדים, אלא מערך, בעל ארבעה איברים. (בעצם לא מערך, אלא רשימה, אבל זה סיפור אחר) ולתוך המערך הזה מוצבים שלושה איברים. מה שיוצא זה ששלושת המשתנים הראשונים מקבלים ערך, והאחרון נשאר undef. נראה את הדוגמא הזאת שוב, עם הבדל קטן:

```
my ($a,@b,$c,$d)=(1,2,3);
```

הפעם @b הוא מערך, ולא סקלר. מה שיקרה הפעם זה: $a=1, @b=(2,3) ו-$c=$d=undef. זה קורה בגלל ש-@b הוא מערך, ולכן ההצבה מכניסה אליו את כל האיברים שנותרו, ולא משאירה ל-$c, $d איברים.
דוגמא אחרת:

```
$a=@arr;
```

כן, סקלר שווה למערך. מה שיוצא, זה שהסקלר מקבל את גודל המערך. אם היו במערך חמישה איברים $a יקבל את הערך 5. יש עוד שתי דרכים לקבל את מספר האיברים במערך:

```
$size = scalar(@arr);
$size = $#arr +1;
```

בשתי השורות $size יקבל את אותו הערך כמו בדוגמא הקודמת. שים לב לתוספת 1 בשורה השניה. זה כי "$#arr" נותן את מספר האיבר האחרון המערך. ומכיוון שהמיספור מתחיל מאפס, אז צריך להוסיף 1 כדי להגיע למספר האיברים במערך. מה יקרה אם המערך ריק? תנסה לבד.
הערה – בעמוד הקודם ציינו שהסימן '#' מסמל תחילת הערה. בכל זאת זה משמש גם לגודל מערך. תחייה עם זה.
ככה שולפים את האיבר האחרון במערך:

```
$a = $arr[$#arr];
```

לבסוף, פקודת splice לטיפול במערך. פקודה קצת מסובכת אך שימושית לעיתים, המסוגלת להחליף את pop, push וכל שאר הפקודות של הכנסת\הוצאת\החלפת איברים במערך. הנה דוגמאות לשימוש בפונקציה:

```
my @arr1 = (1,2,3,4,5); my @arr2=(6,7,8); my @arr3;
```

```
@arr3=splice(@arr1,2,2,@arr2);
```
הפקודה הזאת מורידה את האיברים 3,4 מתוך @arr1, מכניסה במקומם את איברי @arr2, והאברים שהוסרו מוכנסים ל- @arr3 ומה שיוצא זה:
```
@arr1 = (1,2,6,7,8,5), @arr2=(6,7,8), @arr3=(3,4)
```
הכנסת והוצאת איבר בעזרת splice:
```
splice(@arr1,3,0,2.5); # @arr1=(1,2,6,2.5,7,8,5)
my $val=splice(@arr1,5,1); # @arr1=(1,2,6,2.5,7,5),
$val=8
```
רק נסביר, ש-0 מסמל לפני האיבר הראשון, 5 אחרי האיבר החמישי, 1- זה לפני האיבר האחרון.

2.6. טבלת האש

או בלעז, Hash-Table. זהו מבנה נתונים דומה למערך, רק שבמקום להשתמש במספר כדי לציין מקום של איבר במערך, לכל איבר יש מפתח, שהוא יכול להיות מחרוזת או מספר (אבל לא מצביע) שהוא המציין של האיבר. (שהוא סקלר כלשהו) לסוג נתונים זה יש את הסימן % בתחילת שם המשתנה.

דוגמא להגדרה של האש:
```
my %h=( 1 => "elf", "Zambura" => 5, 5=>7, "another"=>7);
```
זוהי טבלת האש עם ארבעה איברים, עם המפתחות ("1","Zambura",5,"another") ועם הערכים (7,5,7,"elf") כדי לקבל את הערכים הללו משתמשים בפקודות:
```
my @ar1=keys(%h); # @ar1 gets (1,"Zambura",5,"another")
my @ar2=values(%h); # @ar2 gets ("elf",5,7,7)
```
למעשה, אם תנסו את זה באמת, סביר שלא תקבלו בדיוק את המערך שכתבתי, אלא שהאברים יהיו בסדר אחר. זה בגלל שטבלת האש לא מאכסנת את האיברים בצורה סידורית, אלא לפי מפתחות. ואין דרך לדעת באיזה סדר הם יצאו.

כדי לגשת לאיבר או איברים מההאש לפי מפתח/ות:
```
$m=$h{"Zambura"};
@ar=@h{1, "another"};
```
וכדי להכניס איבר חדש:
```
$h{"new one"}=3.14;
```
אגב, כמו תמיד בטבלאות האש, המפתחות חייבים להיות ייחודיים. זה אומר, שאם בדוגמא למעלה אני אכתוב "another" במקום "new one", אזי הערך הישן שתחת another יידרס (שזה 7) ובמקומו יהיה 3.14.

בשביל מה כל הסיפור הזה טוב? האש משמש להרבה דברים, כגון שמירה על מאפיינים שונים. אפשר לשמור מאפיינים שונים באותו האש, וכל מי שצריך מאפיין כלשהו, ניגש אליו ישירות באמצעות השם שלו, בלי לגעת במאפיינים אחרים. עוד שימוש זה לפברק מבני מערכים (struct של שפת C) כמה שזה נשמע מפתיע, לא קיימים בפרל מבנים, וכדי בכל זאת לפברק אותם פשוט משתמשים בהאש.

2.7. הקשרים ומשתנים

כל פעולה בפרל, מתבצעת תחת הקשר מסוים. הקשר יכול להיות סקלרי או רשימה, והוא נקבע על פי המשתנה אליו התוצאה צריכה להיות מוצבת. הנה דוגמא.
```
$acla=localtime();
@arr=localtime();
```
שתי הפקודות עושות את אותו הדבר, רק שהראשונה מחזירה מחרוזת של זמן, שנראית ככה:
"Thu Oct 13 04:54:34 1994" והשנייה מחזירה את המערך:
```
($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst)
```
זה שימושי מאוד, כי ככה פונקציה יכולה להחזיר את התוצאה שמבוקשת, אבל שם רוצים יותר מזה, אז שמים אותה בהקשר מערך, והיא יכולה להחזיר יותר מערך יחיד.

למרות זאת, פונקציות בד"כ מחזירות תשובה מסוג יחיד, (סקלר או רשימה) בלי אפשרות לבקש משהו אחר. (אם כי פונקציות המנתחות מחרוזות כן נוטות לדואליות)

אם רוצים להכריח פונקציה להחזיר בהקשר סקלרי, משתמשים בפקודה scalar. דוגמא:

```
@arr=("Today","is",scalar(localtime));
```

אם לא היינו שמים את ה-scalar, אזי localtime היתה רואה הקשר רשימה, ושופכת את כל המערך הגדול שלה. ככה קיבלנו רק את המספר.

2.8. תוכנית דוגמא

מכיוון שעד כה כל מה שראינו היה סקלרים, מערכים והאשים, (אפילו לא מחרוזות!) אה, ואת הפקודה print, אז תסלח לי שהתוכנית קצת מטופשת. תתרגל.

```
#!/usr/bin/perl -w
use strict;
my ($a,$b,$c,$d)=(73,22,"Oops");
my (@arr1, @arr2, @arr3);
my (%h1, %h2);
@h1{"a","b","c"}=($a, $b, $c);
@arr1=keys(%h1);
@arr2=values(%h1);
@arr3=%h1;
push(@arr1, "klick");
push(@arr2, "klack");
%h2 = ("first"=>1, "second"=>2, "third"=>3, "forth"=>4);
$h2{"fifth"}=5;
splice(@arr3, 0, scalar(@arr1), @arr1);
$h1{pop(@arr1)}=pop(@arr2);
$c=7;
$d = $a+$b*$c/$a;
print "And the number is: $d !\n";
print "Hash table: ", %h2, "\n";
print "Keys: ", @arr1, " Values: ", @arr2, "\n";
```

3. מבני בקרה

3.1. ביטויים בוליאניים

כמו שכבר הסברנו בפרק סוגי משתנים, סעיף משתנים סקלרים, בפרל 0, המחרוזת הריקה ("") ו-undef הם שקר, כל דבר אחר זה אמת.

פרל תומכת בכל האופרטורים הבוליאניים המוכרים, אם בסגנון C, (&&, ||, !) או בסגנון פסקל. (And, Or, Not)

אופרטורים לוגיים:

פעולה	אופרטור פסקל	אופרטור C
And	$a and $b	$a&&$b
Or	$a or $b	$a || $b
Not	not $a	! $a

אופרטורי השוואה: בפרל יש אופרטורים שונים להשוואה בין מספרים ולהשוואה בין מחרוזות. (טוב, בשפות אחרות כדי להשוות בין מחרוזות יש פונקציות מיוחדות)

פעולה	אופרטור מחרוזת	אופרטור מספר
האם שווה	$a eq $b	$a == $b
האם לא שווה	$a ne $b	$a != $b
האם גדול	$a gt $b	$a>$b
האם גדול-שווה	$a ge $b	$a>= $b
האם קטן	$a lt $b	$a<$b
האם קטן-שווה	$a le $b	$a<= $b
השוואה	$a cmp $b	$a<=>$b

אופרטור ההשוואה מחזיר: 1 אם $a>$b, מחזיר 0 אם $a==$b, ומחזיר 1- אם $a<$b.

3.2. פקודת *if* וניגזרותיה

כן, ניגזרותיה. כלל גדול בפרל אומר, "יש יותר מדרך אחת לעשות את זה". ובפרט, יש יותר מדרך אחת לעשות if. אבל נתחיל ברגילה, בסדר? הנה:

```
if (condition1) {
    Set of commands...
} elsif (condition2) {
    Set of commands...
} else {
    Set of commands...
}
```

אלה שני משפטי תנאי רגילים, משורשרים.

יש גם וורסיות אחרות של משפט ה-if. הפקודות הבאות זהות:

```
if ($m==3) {
    print "Avukado";
}
print "Avukado" if ($m==3);
print "Avukado" unless ($m != 3);
```

3.3. פקודת *while*

טוב, פה פרל זהה לשפת C. כמעט. הנה דוגמא:

```
while ($line=<>) {
    $count_lines++;
    if (substr($line,1) eq '#') {
```

```
        next;
    }
    if ($line eq "End\n") {
        last;
    }
    print $line;
}
```

אז מה יש לנו כאן? בשורה הראשונה יש את אופרטור ה-<>. זה אופרטור שקורא שורה מה-stdin, ומציב אותה לתוך `$line`. אם לא נשארו שורות - הוא יחזיר undef, מה שיתן שקר, והלולאה תסתיים. נסביר בפרק מאוחר יותר על האופרטור הזה.
מיד אחרי שקראתי את השורה, אני סופר אותה. שיהיה. אח"כ אני בודק האם התו הראשון של השורה הוא "#". אם כן, אזי אני מדלג ועובר לשורה הבאה. אם השורה היא "End", הסתיים הקובץ, ואני שובר החוצה מהלולאה. ובסוף, אם לא קרה שום דבר דרסטי, אני מדפיס את השורה לפלט הסטנדרטי (STDOUT).

3.4. לולאת *for*

לולאת for היא בהחלט כמו המקבילה של ב-C. כמעט.

```
my ($m, $i);
$m=1;
for ($i=1; $i<5; $i++) {
    $m = $m*$i;
}
```

יש ללולאות גם פקודות בקרה:

```
my ($m, $i, $j) = (0,0,0);
for ($i=1, $m=1; $i<5; $i++) {
    $j++;
    redo unless ($j>=$i);
    if ($i+$m>30) {
        last;
    }
    next if ($i==1);
    $m=$m*$i/$j;
}
```

פקודת next מקפיצה את ההרצה לסוף הלולאה, ומתחילה סיבוב חדש של הלולאה. כלומר, `$i` מתקדם באחד, והלולאה שוב מתחילה לרוץ. פקודת last שוברת את הלולאה, והתוכנית ממשיכה לרוץ. פקודת redo היא פקודה מיוחדת. היא מקפיצה את ההרצה לתחילת הלולאה. זה אומר ש-`$i` לא מתקדם, אבל `$j` שממוקם שורה מתחתיו, כן יתקדם. לא ראיתי את זה בשום שפה אחרת, אבל דוגמא מעניינת להוראה תמצאו בפרק על טיפול בקבצים.
מקרה אחר של for היא ה-foreach. הנה דוגמא:

```
foreach $count (10,9,8,7,6,5,4,3,2,1,"BOOM!") {
    print "$count\n";
}
```

בכל סיבוב של הלולאה, `$count` מקבל אחד מהערכים של הרשימה שאחריו, שמודפס בפקודה. אם הרשימה היא בעצם מערך, אזי `$count` בעצם נהפך לערך. לא נתמקד במוזרות הזאת, רק נראה דוגמא:

```
my @arr=(1,2,3,4,5);
foreach $val (@arr) {
    $val=$val*2+1;
}
```

כאן הערכים בתוך המערך `@arr` ישתנו. מוזר, אך אמיתי ויעיל.

3.5. פעולות על תנאי

כשפרל בודק משפטים לוגיים, הוא עושה זאת בחלקים. לדוגמא, אם יש את הביטוי הבא:

```
$x or $m++
```

אם $x הוא אמת, אזי $m לא יקודם. זה בגלל שפרל כבר יודע שהערך של הביטוי הוא אמת, ואין לו צורך לבדוק את חלקו השני.

משתמשים בזה בפרל המון, פשוט במקום לעשות if. הנה פתיחת קובץ טיפוסית:

```
open(my $fh, ">", "zorba.txt") or die "Cann't open that
file!\n";
```

הפקודה open פותחת קובץ, בשם zorba.txt, תחת המשתנה $fh. אם היא מצליחה, היא מחזירה אמת, ואז פרל לא ממשיכה לבדוק את התנאי. אם open מחזירה שקר, אז פרל ממשיך לבדוק את התנאי, ומגיע לפקודה die "string", שמוציאה את הודעת השגיאה ועוצרת את ההרצה.

נכון, זה יכול להיות מכוער, ואפשר לכתוב בעזרת זה דברים לחלוטין בלתי קריאים. אבל זה פרל, ואפשר להשתמש בזה במקום המון פקודות if. תגיד בעצמך מה יותר קריא, הפקודה שלמעלה או פקודה מקבילה בסגנון C?

```
if (open(FH, ">", "zorba.txt")==0) {
    die "Cann't open that file!\n";
}
```

בקיצור, אחרי שמתרגלים לזה, זה נחמד.

3.6. תוכנית דוגמא

כרגע יש לנו, כאבני בנין את המשתנים, ואת מבני הבקרה. התוכנית עכשיו יכולה להיות קצת יותר אינטילגנטית. (אבל לא בהרבה)

```
#!/usr/bin/perl -w
use strict;
my ($key, $val);
my %h;
while ($key=<>) {
    defined($val=<>) or die "Odd number of line?!\n";
    chomp($key);
    chomp($val);
    $h{$key}=$val;
}
foreach $key (keys(%h)) {
    print $key, "->", $h{$key}, ", ";
}
print "\n";
print $h{"message"}, "\n";
my $i ;
for ($i=1; $i<$h{"loop count"}; $i++) {
    last if ($h{"loop count"}*2-$i<5);
    print $h{"loop msg"}, "\n";
}
```

התוכנית משתמשת בפקודה שעדיין לא הזכרנו - chomp. זוהי פקודה שמורידה מהקלט את סימני סוף השורה. אם לא נשים אותה, התוכנית גם תעבוד, אבל נקבל כל מיני אזהרות. (בגלל שאנחנו משתמשים ב-"w-" בשורה הראשונה של התוכנית, שאומרת לו להזהיר אותנו) זה יקרה בכל פעם שננסה להשתמש בקלט כמספר, ואז הוא צריך לתרגם אותו למספר, ישמיט את סימני סוף השורה, יזהיר אותנו שהוא עשה את זה, אבל ימשיך לרוץ.

מה שהתוכנית עושה זה לקרוא קובץ, וכל זוג שורות להכניס בתור מפתח וערך לתוך טבלת האש. אח"כ היא מדפיסה את טבלת ההאש שהתקבלה, מדפיסה את הערך שתחת המפתח "message", ואז רצה בלולאה לפי "loop count" ומדפיסה את "loop msg". קובץ קלט לדוגמא:

```
<Start input file>
5
Oops.
loop count
17
And the winner is
me.
loop msg
AhAhAhAh!
message
Resistance is futile
<End input file>
```

שים לב שאין חשיבות לסדר הזוגות. ממילא הכל נשפך אח"כ לתוך האש ומתערבל שם. שים גם לב לכך שמספר השורות צריך להיות זוגי. אם הוא לא זוגי, (יכול להיות שהעורך שלך אוהב להוסיף שורה ריקה בסוף הקובץ) אז פשוט תוסיף בעצמך שורה ריקה, וכך יהיו לך שתי שורות ריקות בסוף הקובץ.

הרצת התוכנית: אם נקרא לקובץ הקלט פשוט "in3", לקובץ של הסקריפט "chap3.pl", אזי ביוניקס השורה הבאה תפעיל את הקובץ:

```
./chap3.pl <in3
```

בחלונות, בהנחה שפרל היא בנתיב החיפוש:

```
perl chap3.pl <in3
```

בהצלחה.

4.טיפול במחרוזות

לפרל יש כוח עצום, בהשוואה לכל שפה אחרת, לטיפול במחרוזות. בעזרת האופרטורים והפקודות, קלות הפעולה בפרל היא ללא תחרות. רק לחשוב על כך שבפרל מחרוזת נחשבת סקלר, לעומת כל שפה אחרת, בה מחרוזת היא מערך או אובייקט, נותן איזו נקודה למחשבה.

4.1. אופרטורים של מחרוזות

```
my ($a, $b, $c) = ("Abra", "kadabra", 3);
print $a . $b, "\n"; # prints "Abrakadabra"
print $a x $c, "\n"; # prints "AbraAbraAbra"
```

4.2. מחרוזות ומשתנים

כדי לשלב משתנה בתוך מחרוזת, לא צריך פקודה מיוחדת. פשוט עושים את אחד מהשניים:

```
$temperature = 37;
$str = "The temp. today is $temperature degrees";
$str = "The temp. today is " . $temperature . " degrees";
```

באותה דרך, אפשר לשלב מערך, איבר המערך, כמה איברים ממערך, טבלת האש שלמה או חלק ממנה, וכו'. כל דבר שמתחיל ב-$, @ או %, יכול להיכנס:

```
@temps=(23,53,25,34,11,7,0);
$str = "The temps in the whole week where @temps degrees,
each day";
$str = "The temp. yesterday was $temps[6] degrees";
$str = "The temps yesterday and the day before were
@temps[6,5] degrees";
```

אם אתה רוצה להכניס את התווים $, @ או % עצמם למחרוזת, פשוט שים \' לפניהם.

```
print "\$temp = $temperature\n";
```

אם אתה רוצה לעצב את המחרוזת בצורה מיוחדת, אתה תמיד יכול להשתמש בפקודה sprintf, המוכרת משפת C, ונתמכת בפרל.

4.3. ביטויים רגולריים

אוקי. לאט-לאט. תירגע. תנשום עמוק – ותתחיל.

מה זה ביטויים רגולריים? ביטויים רגולריים (Regular Expressions, regexp) הם דרך מסודרת לתאר מחרוזת. אפשר גם לומר, שזו דרך לשאול שאלות על מחרוזת. כמו שבמספר אתה יכול לשאול האם הוא גדול משלוש, במחרוזת אתה יכול לשאול האם המחרוזת מכילה "חתול".

ללמד ביטויים רגולריים באופן מלא, גם ספר לא מספיק. ללמד באופן חלקי, צריך הסבר באורך גדול יותר מהחוברת הזאת. אני אתן כאן כמה עקרונות ודוגמאות, שיתנו לך את הבסיס לשימוש נורמאלי בכוח הזה שפרל נותנת לך. מומלץ להתעמק ולמצוא דוגמאות אחרות, כי פה טמון כלבלב גדול.

אנחנו נעבוד בסעיף הזה בעזרת תוכנית דוגמא:

```
#!/usr/bin/perl -w
use strict;
my $str = "string";
print "Found!\n" if ($str =~ m/regexp/);
```

הדבר היחידי שאתה לא מכיר בתוכנית זה התנאי של ה-if. מה שאני שואל שם, זה האם הסקלר `$str` מכיל את המחרוזת "regexp". התשובה היא לא.

קודם כל – תנסה לשחק עם המחרוזות, כדי לקבל אמת. (כלומר, תחליף גם את "string" וגם את "regexp", כדי לראות מתי אתה מקבל אמת ומתי שקר) קח לך עשר דקות. אני אחכה.

עכשיו, נשנה את המחרוזת למשהו מעניין יותר:

```
my $str = "Semuel casts: Abra Kadabra! sugoi"
```
מעכשיו בכל פעם נחליף בתכנית את regexp לשאלה אחרת.
הביטוי הרגולרי הפשוט ביותר, הוא המחרוזת המבוקשת עצמה:

Abra Kadabra

כלומר, בתוכנית יהיה כתוב:
```
print "Found!\n" if ($str =~ m/Abra Kadabra/);
```
וזה יהיה אמת. המחרוזת שלנו באמת מכילה "Abra Kadabra".
עכשיו אפשר להתחיל לשחק:
הנקודה (.) מסמנת כל אות. אם למשל, במקום d לא אכפת לנו שיהיה z ,J ,5 או סימן כלשהו ^, (אות, מספר או סימן כלשהו) אנחנו יכולים להחליף אותה בנקודה:
```
$str =~ m/Abra Ka.abra/
```
המחרוזת באמת מכילה את התת-מחרוזת "Abra Ka", שאחריה יש תו כלשהו יחיד, (במקרה הזה יש את התו "d") ואחריו יש את התת-מחרוזת "abra". לכן, יחזיר אמת.
גם המקרים הבאים יחזירו אמת:
```
my $str = "Semuel casts: Abra KaZabra sugoi";
my $str = "Abra Ka5abra ahoi";
my $str = "blaBLAblaAbra Ka^abraapchiEWE!";
```
כמו שאתה רואה, לא חשוב לשאלה הזאת מה יש לפני המחרוזת המבוקשת, מה יש אחריה, ואיזה תו בדיוק עומד במקום הנקודה. כל עוד המחרוזת עומדת בתנאי, יוחזר אמת.
ננסה משהו אחר. נגיד שבמקום ה-b השני אנחנו רוצים שיהיה או b, או z או 5, ורק אלה. כלומר, נשים בתוך הסוגריים של ה-if את הביטוי הבא:
```
m/Abra Ka.a[bz5]ra/
```
יוחזר אמת. במחרוזת באמת יש את התת-מחרוזת "Abra Ka", אחריה יש תו כלשהו, ("d") אחריו יש "a", אחריו יש אחד מהתווים "z" ,"b" או "5", ("b") ואחריו יש "ra".
רגע, בעצם התכוונתי, שיהיה כל דבר *חוץ* מ b, או z או 5:
```
Abra Ka.a[^bz5]ra
```
יוחזר שקר. כי במקום שאנו מתעקשים שלא יהיה "b", ולא "z" ולא "5", ויש "b". אי לכך, המחרוזת לא תואמת לתנאי, ולכן שקר.
הסימן + מסמן אחד-או-יותר, על הדבר שלפניו. למשל, איפה שיש כרגע K, אני מוכן שיהיה יותר מ-K יחיד: (כלומר, לקבל גם את המחרוזת "Abra KKKagakra")
```
Abra K+a.a[^bz5]ra
```
בעצם, למה להגביל אותנו ל-K. נשלב בין שני הקודמים, ונסכים לקבל גם שבאותו מקום יוכל להיות גם M, T ו-R (כלומר, גם "Abra MTRTKTMRagakra" מתקבל)
```
Abra [KTMR]+a.a[^bz5]ra
```
או סתם כל אות גדולה מה-ABC:
```
Abra [A-Z]+a.a[^bz5]ra
```
הסימן * מסמל אפס או יותר. זוכרים את הנקודה שהכנסנו? אז בואו נעשה אותה אפס-או-יותר. כלומר, שם יכול להיות כל דבר שהוא, בכל אורך, או כלום:
```
Abra [A-Z]+a.*a[^bz5]ra
```
זהירות עם הכוכבית. זה ת'כלס יכול לבלוע לך קבצים שלמים. {איש אחד נכנס עם פיל לקולנוע. השומר אומר לו, אדוני, אתה לא יכול להיכנס עם פיל לקולנוע. האיש מסתובב והולך. אחרי חמש דקות הוא חוזר עם הפיל, ששתי פרוסות לחם תקועות לפיל באוזניים. השומר אומר לו, אדוני, אתה לא יכול להיכנס עם פיל לקולנוע. הוא עונה, אתה לא תגיד לי מה לשים בסנדוויץ' שלי!} אז תיזהרו שלא יהיה לכם פה סנדוויץ' פיל, בסדר?
הסימן כובע (^), בנוסף לפעילותו בתוך הסוגריים המרובעות, גם מסמן תחילת מחרוזת, או אם זוהי מחרוזת עם כמה שורות טקסט בפנים, אז תחילת שורה. בדוגמא הבאה, מחפשים משהו שמתחיל ב-a+[A..Z]bra, אחריו לא משנה מה, שמסתיים בשורה חדשה ו-ra[bz5^]a.
```
Abra [A-Z]+a.*^a[^bz5]ra
```
אגב, די פופולרי הצרוף הזה, ".*^". זה בגלל הקושי לסמן שורה חדשה, או שאתה לא בדיוק יודע מה יש בסוף השורה.
הנה סתם קשר 'או'. אני מוכן שיהיה באותו המקום במחרוזת או 'bra' או 'adr'.
```
A(bra|adr) [A-Z]+a.*^a[^bz5]ra
```

בלי הסוגריים, זה עובד רק על אות בודדת: (בסוף, יש r או z)

```
A(bra|adr) [A-Z]+a.*^a[^bz5]r|za
```

בעצם, אני לא בטוח בגלל שאני רוצה את ה-r הזאת בסוף. (כלומר, אפס-או-אחד)

```
A(bra|adr) [A-Z]+a.*^a[^bz5](?r)a
```

טוב, די. אין לדבר הזה סוף. מה שקיבלנו כרגע, שאנחנו רוצים מחרוזת שמתחילה ב-A, אחריה יש bra או adr, אח"כ רווח, אח"כ אחת או יותר מהאותיות הגדולות באנגלית, (A..Z) אח"כ את האות a, אח"כ משהו, לא משנה כמה ומה. אח"כ שורה חדשה, a, ואז סימן כלשהו שיכול להיות כל דבר חוץ מ-b ,z או 5, ואז אולי r ואולי לא, ובסוף עוד a. ברור, לא?

רק שים לב, שהמון תווים נחשבים לתווי בקרה בביטויים רגולריים, ואם אתם רוצים דווקא לחפש אותם, פשוט תשימו סלש-כפול (\\) לפניהם. למה כפול? כי אל תשכחו שמה שאתם כותבים זה בסופו של דבר מחרוזת, ובמחרוזת סלש מרמז על תו מיוחד. סלש-כפול נהפך אחרי העיבוד של המחרוזת לסלש רגיל, שאומר לביטוי הרגולרי שפה מדובר בתו עצמו, ולא באיזה תו בקרה.

בנוסף לכל מה שאמרנו, יש גם דרכי קיצור לתיאור של תווים. במקום לכתוב [A-Za-z0-9], (שזה כל האותיות באנגלית, פלוס כל הספרות) אפשר להשתמש בקיצור \w, שאומר בדיוק את אותו הדבר.

וזוהי טבלת הקיצורים:

משמעות	הגדרה	צרוף
אות גדולה, קטנה, מספר או קו תחתי	[a-zA-Z_0-9]	\w
רווח לבן (רווח, שורה חדשה, טאב וכו')	[\t\n\r\f]	\s
סיפרה	[0-9]	\d
הכול חוץ מאות\מספר\קו תחתי	[^a-zA-Z_0-9]	\W
הכול חוץ מרווח לבן	[^ \t\n\r\f]	\S
הכל חוץ מסיפרה	[^0-9]	\D
מתקיים כאשר מצד אחד שלו יש \w, ומהצד השני \W. מסמן קצה מילה.	קצה מילה	\b

ניתן כמה דוגמאות, בעזרת הפקודה split. הפקודה מקבלת מחרוזת וביטוי רגולרי, ושוברת את המחרוזת בכל מקום שהיא מצאה רצף המתאים לביטוי הרגולרי.

```
split("\\|","Start|middle|end"); # gets
("Start","middle","end")
split("\\||d","Start|middle|end"); # gets
("Start","mi","","le","en","")
split("[td]+","Start|middle|end"); # gets
("S","ar","|mi","le|en","")
m/<a [^<>]*href="[^"]*"[^<>]*>[^<>]*<\/a>/
```

האחרון הוא תנאי המחפש לינק של HTML בתוך מחרוזת. נשתמש בו בהמשך.

4.4. פקודת התאמת מחרוזת

הפקודה מחפשת תת-מחרוזת (המיוצגת ע"י ביטוי רגולרי) בתוך מחרוזת, ומחזירה את המחרוזת אם מוצאת. כמו כן, אפשר לבקש חתיכות מהמחרוזת. אשר להריץ את הפקודה שוב ושוב על אותה המחרוזת, כדי לקבל בכל פעם את תת המחרוזת הבאה. דוגמא:

```
while ( $text =~ m/Semuel is a(?n) (\w+) guy/g ) {
    print $1,"\n";
}
```

הפקודה שבתוך התנאי של פקודת ה-while משמעותו, לחפש בתוך `$text` את הביטוי המובא בין שני הסלשים. אם היא לא מצאה, היא תחזיר undef, שהוא שקר, והפקודה תסתיים. אם מצאה, היא תחזיר בכל פעם את המחרוזת שהיא מצאה, ומחרוזת מלאה זה אמת, שתבצע את הפקודה. כשיגמרו ההופעות של תת המחרוזת בתוך המחרוזת, היא תחזיר undef, ובא לציון גואל.

שים לב לכך שבכל הקוד הוספתי פרמטר g לפקודה. זה אומר לו שיחפש בתוך המחרוזת שוב ושוב, עד שנגמרות לו התשובות, כמו שמוסבר בפסקה הקודמת. בלי g, הלולאה שלמעלה תרוץ לעד, שכל פעם היא מוצאת את אותו הדבר.

עכשיו, מאיפה צץ $1 ?! שימו לב לסוגריים סביב ה- '\w+'. הסוגריים אומרים, שאני מעוניין בקטע הזה של המחרוזת בנפרד. אז הערך מושם ב- $1. אם יש יותר מסוגריים יחיד, אזי שמים את הבאים ב- $3 ,$2 וכו'. הסוגריים עם ה-n בפנים לא נחשבים, כי זה חלק מהביטוי הרגולרי. אגב, זו פקודה שמתנהגת שונה תחת הקשר רשימה. בהקשר רשימה, היא מחזירה רשימה של כל תת המחרוזות המתאימות לביטוי. דוגמא:

```
@arr = $text =~ m/Semuel is a(?n) \w+ guy/g;
```

דוגמא ללולאה. אם $contect מכיל קובץ html שלם, אז כדי לקבל את הלינקים שלו, עושים:

```
while( $context =~ m/<a
[^<>]*href="([^"]*)"[^<>]*>([^<>]*)<\/a>/g ) {
    print "The url is: $1\n";
    print "The text is: $2\n";
}
```

תשאל, למה אני משתמש בכל מקום ב-' *[<>^]', במקום בסתם ' *.'. התשובה היא שהאלגוריתם שעובד פה הוא חמדן. זה אומר שאם אני אשתמש ב-" *.", הוא ייקח מהלינק הראשון של הקובץ עד האחרון, תחת אחד ה-' *.' (סנדוויץ' פיל). ככה, אני מכריח אותו שלא לקחת את התווים '<>', מה שמגביל אותו שהוא לא יכול להתפשט ליותר מלינק אחד, כי זה מכריח שימוש ב-'<>'.
אפשר להוסיף אופציות לפקודה, כגון האם להתחשב באותיות גדולות וקטנות ועוד. האופציות יתווספו אחרי הסלש השני של הפקודה. למשל:

```
$text =~ m/abra/i;
```

4.5. פקודת החלפה

דומה לפקודת ההתאמה, רק שבמקום למצוא תת מחרוזת בתוך מחרוזת, פה אנחנו גם מחליפים את תת המחרוזת בביטוי אחר. דוגמא:

```
my $text = "This is a long text, written slowing in the
evening";
$text =~ s/long/short/;
$text =~ s/written \w+ (\w+)/scratched $1 fast/;
```

בפקודה הראשונה, אני פשוט מחליף את המילה "long" במילה "short". בפקודה השניה, אני מחפש שלושה מילים, שהראשונה מהם היא "written", ומחליף אותם במילים "scratched fast", והמילה האחרונה מועברת לצד השני של הפקודה, ומשובצת שם. התוצאה תהיה:
"This is a short text, scratched in fast the evening"

4.6. חלק ממחרוזת

עכשיו נעזוב קצת את השדות הזוהרים של הביטויים הרגולריים. ניקח פקודות פשוטות, לא משהו מתוחכם - פשוט קבלת חלק ממחרוזת. בלי ביטויים רגולריים ובלי סיפורים. תן לי מהמחרוזת, מתו מספר 5, שלושה תווים:

```
$newstr = substr($str,5,3);
```

מצד שני, אם נוסיף לפקודה עוד פרמטר, שהוא עוד מחרוזת, הפקודה תחתוך את הקטע מהמחרוזת, תכניס במקומו את הקטע החדש, (ואם הם באורכים שונים, היא תשנה את האורך של המחרוזת) ותחזיר את הקטע החתוך. תראה:

```
my $text = "This is a long text, written slowing in the
evening";
print $text, "\n";
print substr($text,5,5,"xxmmxxmmxx"), "\n";
print $text, "\n";
```

4.7. מציאת מיקום בתוך מחרוזת

אם אתה מחפש איפה האות g נמצאת בתוך מחרוזת, אז משתמשים בפקודת ה-index. דוגמא:

```
my ($str, $pos) = ("Kenguru");
```

```
$pos = index($str,"g");
```
אפשר לעשות את זה על יותר מתו אחת, ואפשר גם להגיד לו מאיפה להתחיל לחפש. הנה קטע קוד פופולרי:
```
my $str = "Here we are today, there were the greens";
my ($pos,$old_pos)=(0,0);
while (($pos = index($str,"re",$old_pos))>=0) {
    print "Found \"re\" at $pos!\n";
    $old_pos = $pos+1;
}
```
הפונקציה rindex היא בדיוק אותו הדבר, רק שהיא מחפש מהסוף, ולא מההתחלה. אם תחליפו בקוד למעלה את index ב-rindex, (וכמובן, תשחקו עם הערכים ההתחלתיים של שני משתני המיקום) הוא יוציא בדיוק את אותו הפלט, רק בסדר הפוך. מהסוף, להתחלה.

4.8. חיבור מחרוזות

מי שקופץ ואומר, "אמרנו שיש אופרטור נקודה שמחבר מחרוזות!", אז צודק. זה טוב אם אנחנו רוצים לחבר שתי מחרוזות, ביחד. אבל אם יש לנו רשימה של מחרוזות, אז במקום לעשות for שבתוכו אופרטור הנקודה, יש לנו פקודה בשביל זה:
```
my
@strarr=("Here","we","are","today,","there","were","the",
"greens");
print join('_',@strarr);
```
זה ידפיס: "Here_we_are_today,_there_were_the_greens". שים לב לקו התחתי - זה מה שאמרתי לו להכניס בין המחרוזות. זו הפקודה ההפוכה ל-split. שם חתכנו לפי איזה ביטוי, פה אנחנו מחברים בעזרת מחרוזת.

4.9. חיתוך שורה חדשה

אולי פעם קראת שורות מקובץ, וכל פעם טרחת לחתוך את סימני שורה-חדשה (n\). פה, אתה פשוט משתמש בפקודת chomp:
```
my $line;
$line = <>;
chomp($line);
```
זוכר את האופרטור "<>"? הזכרנו אותו בעבר. זה אופרטור שקורא שורה מה-stdin. (שזה מהמקלדת, או מקובץ) אז אני לוקח פה שורה מה-stdin, ומכניס אותה ל-line. אולי יש בפנים את סימן שורה-חדשה, אולי לא. לא יודע.

chomp לוקח את המחרוזת, מחפש בפנים סימני שורה-חדשה, וחותך אותם. לא משנה אם אתה עובד על חלונות או על לינוקס. זה יעבוד. בכל פעם שתראה בתוכנית פרל קריאה מקובץ, תראה תמיד מיד אחריה את chomp. יעיל בטרוף.

4.10. סיכום מחרוזות

לא, אלה לא כל הפקודות של מחרוזות. יש המון. וגם לא תיארתי פה את כל מה שהפקודות מסוגלות לעשות. (למשל, לפקודה החלפה יש טונות של אופציות) אבל אלה הפקודות הבסיסיות ביותר והנפוצות ביותר, ואיתם תוכל לעשות משהו מועיל. וזה, בסופו של דבר, מטרת המסמך זה.

4.11. תוכנית דוגמא

יש לנו משתנים, לולאות ומחרוזות. למען האמת, אנחנו יכולים לכתוב עכשיו תוכניות סבירות בהחלט. למשל, תוכנית שעובדת על התגים של קובץ html:
```
#!/usr/bin/perl -w
use strict;
my @taglist=();
```

```
my ($line, $tag, $stag);
while ($line=<>) {
    chomp($line);
    while($line=~m/<([^<>]*)>/g) {
        $tag=$1;
        $tag=~s/^\s+//;
        $tag=~s/\s+$//;
        ($stag)=($tag=~m/\w+/);
        if (substr($tag,0,1) ne "/") {
            print "Starting tag($#taglist): $tag\n";
            push(@taglist, $stag);
        } else {
            while (($#taglist>0) and ($stag ne
pop(@taglist))) {}
            print "Ending tag($#taglist): $tag\n";
        }
    }
}
```

התוכנית קוראת קובץ עם תגים, ועוקבת אחרי התחלה וסוף של כל תג. היא מתחילה בהגדרת ארבעה משתנים, ומתחילה לקרוא את הקלט שורה-שורה. עבור כל שורה, התוכנית מחפשת את התגים בשורה. (אל דאגה, התוכנית לא תיתקע על התג הראשון ותמצא אותו שוב ושוב, אלא בכל פעם שורת ה-while תחפש את התג הבא, עד שיגמרו התגים במחרוזת ואת הלולאה תסתיים) התג שנמצא מאוכסן בתוך $1. (שים לב לסוגריים בתוך פקודת ההתאמה) אני מציב את הערך לתוך $tag, מוריד רווחים בהתחלה ובסוף, ומכניס את המילה הראשונה בתג לתוך $stag. אם אין את הסימן "/" בתחילת התג, זה אומר שמדובר בתחילת תג. אני מוציא הודעה מתאימה, הכוללת גם את גודל המערך שלי ואת התג עצמו, ודוחף את התג לתוך רשימת התגים שלי. אם יש את הסימן "/" בתחילת התג, זה אומר שזה סוף תג. אני מוריד מהמערך איברים עד שאני מגיע לתג המתאים, ומוציא שוב הודעה מקבילה על סוף תג.

התוכנית מניחה שכל התגים הם באותיות קטנות, מכילים מילה אחת לפחות, ושהסגירה שלהם מתחילה באותה המילה. אם יהיה סגירה של תג בלי ההתחלה שלו, התוכנית תיגמר מאוד מהר. כמו כן התוכנית מניחה שאין שבירה של תג בין שתי שורות.

5.פונקציות והוראות

5.1. משתני ברירת מחדל

פרל עובדת דרך משתני ברירת מחדל. תכירו אותם: _@ , _$, שהם המערך וסקלר ברירת המחדל. הכי טוב להראות דוגמא. ארבעת זוגות הפקודות הבאות זהות:

```
$m = shift(@_);
$m = shift;
print $_;
print;
@_=split($_);
split;
chomp($_);
chomp;
```

כזכור, הפקודה shift לוקחת מערך ומורידה לו איבר בתחילתו. אם אתה לא אומר לפקודה לאיזה מערך אתה מתכוון, היא מניחה שאתה מדבר על _@. גם print, אם אתה לא אומר לה מה להדפיס, מדפיסה את _$. וגם chomp, כשלא אומרים לה לאיזה משתנה לבצע, היא מניחה שמתכוונים ל- _$. אבל split היא בכלל מקרה מיוחד. אם לא אומרים לה את מה לחתוך, היא מניחה שמתכוונים ל- _$. אם לא אומרים לה גם איפה לשים את המערך שהיא הרגע יצרה, היא שופכת אותו לתוך _@. (למעשה, ההתנהגות הזאת לא מומלצת, כי בטעות אתה עלול לדרוס בקלות את _@. אם תשתמש בה, תקבל אזהרה)

במבט ראשון זה מוזר. במבט שני זה מוזר. במבט שלישי, יש לזה פוטנציאל. אם כל הפונקציות והאופרטורים יעבדו ככה, לא נצטרך בכלל משתנים בקטעי קוד מסוימים.

מצד שני כתבתי כבר אלפי שורות קוד, וחוץ מהמקומות שמחייבים זאת, לא השתמשתי בהם.

בתיעוד של כל פונקציה (לדוגמא: perldoc -f shift) כתוב בפרוטרוט מה הפונקציה רוצה לקבל, ומה יקרה אם היא לא תקבל את זה. לא כל הפונקציות משתמשות במשתני בירת המחדל, לכן יש לעיין במדריך לצרכן לפני השימוש.

5.2. כתיבת פונקציות

פונקציה היא חתיכת קוד, המקבלת את הפרמטרים שלה בתוך המערך הדיפולטי, _@, ומחזיר לתוכנית הקוראת מערך או משתנה, (דרך המערך או המשתנה הדיפולטי) בעזרת הפקודה return. סימן הזיהוי של סוג הפונקציה שאני אראה כאן הוא התו '&', אבל אף אחד לא משתמש בו כי אנחנו שמנים ועצלנים. (ופרל בד"כ מסוגל לנחש בעצמו שאתה מתכוון לקרוא לפונקציה בשם זה)

השם הרשמי של הסוג הזה הוא List-Operator, זה כי היא מקבלת מערך בתור קלט.

דוגמא:

```
sub myChomp {
    $_=shift;
    chomp($_);
    return $_;
}
my $text="And the winner is\n";
print myChomp($text);
chomp($text);
print $text, "\n";
```

הפקודה shift מוציאה את האיבר הראשון מתוך _@ ושמה אותו בתוך _$. הפקודה chomp, שכבר הכרנו אותה בפרק הקודם, בלי פרמטרים היא עובדת על _$. ואז, אנו מחזירים את _$ לתוכנית הקוראת. (תנסה גם להוסיף את התו '&' לפני myChomp בשורה השביעית, ותראה שפשוט לא תהיה לזה השפעה)

ננסה עוד דוגמא:

```
sub myTime {
    my ($msg1, $i, $msg2) = @_;
    print $msg1, (localtime)[$i]+1, $msg2, "\n";
}
```

אם נכתוב בגוף התוכנית:

```
myTime("Today is the ",6," day of the week");
```

נקבל: (בהנחה שמדובר ביום שלישי)

```
Today is the 3 day of the week
```

שים לב שפה לא החזרנו שום דבר. קורה. פונקציה לא חייבת להחזיר משהו. יכולה להחזיר מערך, סקלר, או כלום. ניקח שוב את הדוגמא האחרונה:

```
sub myTime {
    my ($msg1, $i, $msg2) = @_;
    print $msg1, (localtime)[$i]+1, $msg2, "\n";
    return ($msg1, (localtime)[$i]+1, $msg2);
}
```

פה היא מחזירה מערך, המכיל את שתי ההודעות שהתקבלו ואת הערך של localtime המבוקש.

אבל אם נכתוב:

```
my $ans = myTime("Today is the ",6," day of the week");
```

התוצאה היא ש- $ans=3. זה בגלל שהחזרנו מערך, ואם עושים סקלר שווה מערך, מקבלים בסקלר את גודל המערך. כדי שהפונקציה תדע מה התוכנית רוצה ממנה, משתמשים בשאלה wantarray.

ככה:

```
sub myTime {
    my ($msg1, $i, $msg2) = @_;
    print $msg1, (localtime)[$i]+1, $msg2, "\n";
    if (wantarray) {
        return ($msg1, (localtime)[$i]+1, $msg2);
    } else {
        return join(" ", $msg1, (localtime)[$i]+1,
$msg2);
    }
}
```

אם רוצה מערך, נותנים לו מערך. אם רוצה סקלר, נותנים לו סקלר. בדיוק כמו ש-localtime יודע מה רוצים ממנו, ככה הפונקציה שלך.

הערה: בטח כבר הפנמת שהמשתנים עוברים דרך מערך. זה אומר, שבעצם אפשר להעביר רק מערך יחיד. הנה צורות קריאה נכונות ולא נכונות:

```
myFunc1($str, $i, @kaskas); #right.
myFunc2($str, @kaskas , $i); #wrong.
```

באפשרות השניה, כאשר תעשה בתחילת הפונקציה:

```
my ($str, @kaskas , $i) = @_;
```

אזי $str יקבל את שלו, אבל @kaskas ישתה את שאר המשתנים, ו- $i יקבל undef. לכן, ככה אפשר להעביר לפונקציה כמה סקלרים, אך רק מערך אחד, בסוף הרשימה.

אתה בטח עכשיו אומר, אבל יש פונקציות שמקבלות מערך בהתחלה, ואח"כ סקלרים ומערכים נוספים, כמו splice. זה נכון, אבל הם עובדים קצת אחרת. לא ניכנס לנושא, כי זה גורר נושאים אחרים כמו מצביעים. נסתפק בפונקציות רגילות, כאלה.

ולסיום, משהו קטן: דרך פשוטה לעשות ערכי ברירת מחדל לפרמטרים:

```
sub write_person {
    my ($first, $last, $age, $address) = @_;
    $age ||= 23;
    $address ||= "the galaxy";
    my $msg = "$first $last is $age years old ";
    $msg .= "and lives in $address";
```

```
    return $msg;
}
print write_porson("John", "Rahul", 17), "\n";
```
הפונקציה רוצה לקבל ארבעה סקלרים - שני ערכים חובה, ושניים אופציונליים. כל ערך שלא יתקבל יהיה undef, ולכן יוכנס לו את הערך הדיפלטי.

5.3. הוראות לקומפילר

כשמריצים סקריפט פרל, הוא עובר שני שלבים: קומפילציה והרצה. אפשר לתת הוראות לקומפילר של פרל, ולהסביר לו מה אתה רוצה. לדוגמא, השורה מהתוכנית בפרק הראשון:
```
use strict;
```
הפקודה אומרת לקומפילר, שמפה והלאה הוא עובד במצב strict, מה שאומר כל מיני הגבלות. ההוראה תהיה תקפה עד סוף הבלוק בו היא נכתבה. נדגים על הוראה אחרת:
```
my $i;
for ($i=1; $i<=10; $i++) {
    use integer;
    print $i/3, "\n";
}
print "Final: ", $i/3, "\n";
```
ההוראה use Integer אומרת לקומפילר שמפה והלאה, להשתמש בקוד עבור מספרים שלמים. זאת אומרת, שהתוצאה תהיה 0,0,1,1,1,2,2,2,3,3. אבל ברגע שנגמרת הלולאה, הפקודה לא תקפה יותר, וההדפסה האחרונה תהיה 3.333. כשאתה כותב use strict בתחילת התוכנית, אתה בעצם מחיל אותו על כל התוכנית. זה בגלל שהוא נכתב בהתחלה, מחוץ לכל בלוק. אם אתה רוצים לבטל את ההוראה לקטע מסוים, אזי תשתמש בפקודה no. לדוגמא: no strict.
```
no strict;
for (; $bla<28; $bla+=3) {
    print $bla, "\n";
}
```
אוטומטית יוגדר משתנה חדש, $bla, יאותחל ל-undef, ויורץ בלולאה. בסיבוב הראשון נקבל שתי התראות שאנחנו עושים פעולות על משתנה לא מאותחל. אחרי הפעם הראשונה ש- $bla מקודם בשלוש, הערך שלו יהיה שלוש, והלולאה תמשיך לרוץ. זה שימושי, אם אתה רוצה לעשות באיזה חלק תוכנית דברים שלא יעברו ב-strict, אבל יעשה לך חיים פשוטים. כמובן שלא למדנו (ולא נלמד) במדריך הזה שום דבר כזה, אבל שתדע שהאופציה קיימת. הוראות נוספות במערכת:
```
use integer;
no integer;
```
מורה לקומפילר לעבוד עם מספרים שלמים בלבד.
```
use constant PI => 3.14;
print PI *10, "\n";
```
הגדרת קבוע. שים לב שכאשר משתמשים בקבוע, לא צריך את סימן ה-$. אפשר גם להגדיר קבוע מערך:
```
use constant ARRAY => ( 1,2,3,4 );
print ARRAY;
print((ARRAY)[3]);
```
הגדרת הקבוע כופה הקשר רשימה מצד אחד, אך לא מערך. (תעיף מבט על ההסבר על המערכים בפרק 2, זה מוזכר) כלומר, כאשר הגדרנו את PI, בעצם הגדרנו רשימה שמה יש מספר אחד, והוא 3.14. ואז בפקודת ה-print הסתכלנו עליו בהקשר סקלרי, מה שנותן את האיבר האחרון (והיחיד, במקרה הזה) של הרשימה. מצד שני, כשרצינו להציג את האיבר השלישי של ARRAY, כדי לקבל מערך "אמיתי" היינו צריכים לכלוא את הקבוע בסוגריים. הצץ בעזרה של פרל:
```
perldoc constant
```
עבור עוד הרבה דוגמאות.

5.4. תוכנית דוגמא

זוהי תוכנית דוגמא, המקבלת רשימת מחרוזות ומחזירה אותם אחרי חיתוך הרווחים (או כל רווח-לבן) בהתחלה, הרווחים בסוף, וצמצום רווחים רצופים לרווח יחיד. כל זה, ללא שימוש בביטויים רגולריים.

```
#!/usr/bin/perl -w
use strict;
my @strs = ("first last", " somespace more space ", "\t\t
include\ttabs\t \t as well\t \t", " ");
my @str = @strs;
foreach my $s (@str) {
    print "'$s'\n";
    $s = remove_leading_space($s);
    print "'$s'\n";
    $s = remove_trailing_space($s);
    print "'$s'\n";
    $s = reduce_space($s);
    print "'$s'\n";
    print "======================\n";
}
sub remove_leading_space {
    my $s = shift;
    my $i=0;
    my $l = length $s;
    while ($i < $l) {
    my $c = substr($s, $i, 1);
    last if (not is_white_space($c));
    $i++;
}
substr($s, 0, $i, '');
    return $s;
}
sub remove_trailing_space {
    my $s = shift;
    my $i=0;
    my $l = length $s;
    while ($i < $l) {
        my $c = substr($s, - $i -1, 1);
        last if (not is_white_space($c));
        $i++;
    }
    substr($s, -$i, $i, '');
    return $s;
}
sub reduce_space {
    my $s = shift;
    my $i = 0;
    while ($i < length($s)-1) {
        my $c = substr ($s, $i, 1);
        if (not is_white_space($c)) {
            $i++;
```

```
            next; # skip if not white space
        }
        $c = substr ($s, $i+1, 1);
        if (not is_white_space($c)) {
            substr($s, $i, 1, ' '); # make sure it is a
space
            $i+=2;
            next; # skip two if next is not white space
        }
        substr($s, $i, 1, ''); # cut off one white space
        #$i++;
    }
    return $s;
}
sub is_white_space {
    $_=shift;
    return (($_ eq " ") || ($_ eq "\t") || ($_ eq "\f")
|| ($_ eq "\n") || ($_ eq "\r"));
}
```

הנה תוכנית שעושה את אותו הדבר, רק בעזרת פקודות החלפה.

```
my @str = @strs;
foreach my $s (@str) {
    print "'$s'\n";
    $s =~ s/^\s+//;
    print "'$s'\n";
    $s =~ s/\s+$//;
    print "'$s'\n";
    $s =~ s/\s+/ /g;
    print "'$s'\n";
    print "======================\n";
}
```

והנה תוכנית שלישית שעושה את אותו הדבר, גם בלי ביטויים רגולריים, תוך ניצול תכונה מיוחדת של פקודת split כשהיא פועלת עם פרמטר חיתוך רווח.

```
for my $s (@str) {
    $s = join ' ', split ' ', $s;
    print "'$s'\n";
}
```

6.טיפול בקבצים

6.1. הכוונת קלט\פלט

אחת הדרכים הקלות לטפל בקבצים, זה להזין אותם לתוכנית דרך הקלט\פלט הסטנדרטי.
התוכנית תקרא את הקלט כאילו המשתמש כתב אותו, בעזרת האופרטור <>, ותוציא את הפלט בעזרת print רגיל, שינותב לקובץ אחר. דוגמא להרצת תוכנית כזו:

```
./myprog.pl <inFile >outFile
```

התוכנית תקבל את הקלט מתוך הקובץ inFile, ותוציא את הפלט לקובץ outFile. והיופי בזה, שהתוכנית בכלל לא יודע שהיא עושה את זה. מבחינתה, היא מקבלת הקלדות מהמשתמש, ומוציאה לו על המסך מידע.
אתה יכול לקחת את תוכנית הדוגמא שבסוף הפרק, להוריד את כל הוראות פתיחת וסגירת הקבצים, ופשוט לחתוך את המילים "InFile" ו-"OutFile". ואם נקרא לתוכנית LinkProcess.pl, אזי אפשר להריץ:

```
./LinkProcess.pl  < urls.txt  > urls.html
```

6.2. משתנה מטיפוס קובץ

הכר את טיפוס הנתונים הרביעי של פרל: קובץ (File Handler). לטיפוס זה אין שום סימן לפניו, ובאופן מסורתי המשתנים נכתבים עם אותיות גדולות. כמו כן, סוג משתנה זה אינו דורש הגדרה, אלא אפשר פשוט להתחיל להשתמש בו, כאילו אין strict.
לפתיחת קובץ משתמשים בפקודת open. (מפתיע, אך אמיתי) אפשר לפתוח קובץ לקריאה ולכתיבה. ככה:

```
open(FH, "> wwwest.txt");
```

עכשיו תשכח ממנו. למה? כי מה פתאום יש לנו משתנה שלא הגדרנו? ועוד הוא גלובלי, כלומר אם תשתמש באותו שם משתנה בפונקציות אחרות, אתה דורס את המשתנה הקודם. ואם אתה מאלה שכותבים פונקציות רקורסיביות, אז בכלל אי אפשר להשתמש בזה.
אני גם מבקש שתשכח שאפשר לכתוב open עם שני פרמטרים. אין שום סיבה להשתמש בזה היום, אלא רק בצורות עם שלושה פרמטרים. אם אתה רואה open עם שניים, אוטומטית תחליף לשלושה, כמו שתראה בסעיפון הבא.

6.3. פתיחת קובץ

אז הנה, ככה פותחים קובץ:

```
open(my $fh, ">", "wwwest.txt");
```

הפקודה הזאת פותחת את הקובץ wwwest.txt לכתיבה. אם הקובץ לא קיים, הוא נוצר.
שים לב שהגדרנו את $fh בעזרת פקודת my, ולכן הוא משתנה מקומי כמו כל משתנה אחר.
בד"כ מוסיפים לפקודת open גם פקודת die:

```
open(my $fh, ">", "wwwest.txt") or die "Cann't open
wwwest.txt!\n";
```

פקודת die היא ההתראה (Exception) של פרל. בניגוד לפקודת exit (שגם קיימת) שפשוט מפסיקה את הרצת התוכנית, die מוציאה הודעת שגיאה מסודרת, וסוגרת את התוכנית. אפשר לתפוס את סגירת התוכנית, ולהמשיך אותה בכל זאת, אבל זה סיפור אחר.
בכל מקרה, הפקודה שלמעלה מנסה לפתוח\ליצור את הקובץ לכתיבה. אם היא מצליחה, היא מחזירה אמת, ואין צורך להתקדם לפקודה הבאה. אם היא לא מצליחה, היא מחזירה שקר, ואז פקודת die פועלת ומפסיקה את ההרצה.
פתיחת קובץ לקריאה:

```
open(my $fh, "<", "wwwest.txt") or die "Can't open
wwwest.txt!\n";
```

פתיחת הקובץ להוספה (Append):

```
open(my $fh, ">>", "wwwest.txt") or die "Can't open
wwwest.txt!\n";
```

6.4. אופרטור <>

הזכרתי את האופרטור הזה בעבר. מה שהוא עושה, זה לקרוא מקובץ שורה. ככה:

```
$line = <$fh>;
```

אם לא כתוב מחזיק-קובץ בין סימני הגדול-קטן, אזי הוא קורא מתוך ה-stdin, שזה הקלט הסטנדרטי. זה יכול להיות מהמקלדת, או מקובץ, תלוי איך מריצים את הקובץ. הנה דוגמא טיפוסית לקוד:

```
my %config;
while (my $line = <$fh>) {
    chomp $line;
    next if ($line =~ m/^#/); # skip comments
    my ($key, $value) = split '=', $line;
    $config{$key} = $value;
    print "Got config: $key=$value\n";
}
```

פקודת ה-while מפעילה את אופרטור ה-<>, שקורא שורה מתוך מחזיק קובץ $fh שככל הנראה פתחנו קודם לקריאה, ומכניס את השורה (כולל סימן סוף השורה) לתוך המשתנה $line. כל עוד שיש מה לקרוא מהקובץ, המשתנה הוא אמת, ולולאת ה-while ממשיכה לרוץ.

שורה שניה - עושה chomp ל- $line - מה שמוריד את סימן סוף השורה מהמשתנה. כי הוא בד"כ לא באמת מעניין אותנו.

שורה שלישית - בודקת האם הקלט מתחיל בסימן '#'. אם כן - זוהי שורת הערה ומדלגים לשורה הבאה.

אנחנו יודעים שבקובץ יש שורות של פרמטרים, בפורמט של מפתח=ערך. אז שוברים את השורה לשניים לפי ה-"=". הראשון מפתח, השני ערך, ואנחנו מכניסים את שניהם להאש כלשהו. בתקווה שגם נעשה עם ההאש העם ההאש הזה משהו אח"כ.

אם תריץ את האופרטור <> בהקשר רשימה, אז הוא יקרא את כל השורות, ולא רק אחת.

```
my @lines;
@lines=<>;
```

אחרי זה, כל שורות הקלט יהיו בתוך המערך @lines.

6.5. כתיבה לקובץ

פה אין הרבה הפתעות - בשביל לכתוב לקובץ משתמשים בפקודה print. כמו שעד עכשיו כתבנו לתוך הפלט הסטנדרטי, ככה נכתוב לכל קובץ פתוח אחר:

```
open(my $fh, ">", "tmp.txt") or die "AHHHHH!\n";
print $fh "The time is: " , scalar(localtime()), "\n";
```

שים לב שאין פסיק בין FH לבין המחרוזת הראשונה. ככה זה. (ואם תשים פסיק, זו תהיה טעות.

פקודת ה-print תנסה להדפיס את מחזיק הקובץ. דווקא חוסר הפסיק אומר לו שפה יש מחזיק קובץ, ולא משהו להדפסה)

6.6. סגירת קובץ

בצורה מפתיעה ביותר, בשביל לסגור קובץ משתמשים בפקודת close.

```
close($fh);
```

6.7. אבל הקובץ שלי הוא

אבל הקובץ שלי הוא לא סתם שורות! הוא <הכנס כאן פורמט זה או אחר>!

ובכן, אם זה פורמט מוכר, רוב הסיכויים שיש כבר מודול ב-CPAN שמטפל בו. הצץ בפרק 7 להסברים על חיפוש והתקנת מודולים משם.

אם זה קובץ בינארי לא מוכר, ואין מודול שיעשה לך את העבודה, תצטרך להשתמש בפקודות binmode, read, write לקרוא ולכתוב מהקובץ הזה. הנה דוגמא של תוכנית קטנה שעוברת על קובץ בינארי ועושה checksum עליו:

```
open my $fh, "< :raw", "binfile.dat" or die "can not open
file";
my $buf = '';
my $checksum = 0;
while (read( $fh, $buf, 100 )) {
    my @numbers = unpack "C*", $buf;
    for my $t (@numbers) {
        $checksum = ( $checksum + $t ) % 256;
    }
}
print "File Checksum is $checksum\n";
```

6.8. תוכנית דוגמא

```
#!/usr/bin/perl -w
use strict;

open(my $InFile, "<", "urls.txt") or die "Error IN!\n";
open(my $OutFile, ">", "urls.html") or die "Error OUT\n";

print $OutFile "<HTML><HEAD><TITLE>";
print $OutFile "Semuel's Link Page";
print $OutFile "</TITLE></HEAD><BODY>";

while(my $line = <$InFile>){
    chomp $line;
    if ($line=~m/^http/i) {
        print $OutFile "<A HREF= $line > $line
</A><br>\n";
    } else {
        print $OutFile "$line <br>\n";
    }
}

print $OutFile"</BODY></HTML>";

close($InFile);
close($OutFile);
```

התוכנית קוראת מקובץ טקסט המכיל רשימת לינקים, וכותבת קובץ html המכיל את אותה הרשימה, רק שעכשיו הלינקים הם באמת לינקים, (השורות שהם לא לינק עוברות דרך התוכנית, ואפילו מקבלות תג של סוף שורה בסופן) ואפשר להשתמש בקובץ דרך גלשן אינטרנט. (זה למעשה תוכנית ישנה שפעם עבדתי איתה. היה לי קובץ טקסט מסכן שם זרקתי את כל הלינקים שרציתי לשמור, והתוכנית הזאת הפכה את קובץ הטקסט למשהו שמיש)

7.מודולים

7.1. כללי

עד עכשיו השתמשנו ביכולות הבנויות של פרל. אחלה יכולות, אבל מה הלאה?
כדי להרחיב את היכולות של פרל, משתמשים במודולים. מודולים הם קבצי פרל, שאתה מוסיף לתוכנית שלך. זה כמו ה-include של C. מישהו כבר ישב וכתב קוד המטפל בפעולות מסובכות. לא צריך להמציא את הגלגל מחדש בכל פעם. (למרות שזה כיף)
קיימים אלפי מודולים עבור פרל. מודולים עבור תקשורת מול מסדי נתונים, מודולים עבור תקשורת מול האינטרנט, מודולים לניתוח קבצים, מודולים עבור מבני נתונים מסובכים. תבחר נושא, ובטח יש בשבילו מודול. כל המודולים נכתבו ע"י אנשים פרטיים, ולכן קיימים מודולים עם באגים, קיימים מודולים בהם המחבר נעלם, אבל רובם תקינים, והמחברים בד"כ מתקנים את הבאגים, אם וכאשר נמצאים כאלה. אז כנס לאתר search.CPAN.org, ותרגיש חופשי למצוא מודול שעושה בדיוק את מה שאתה צריך.

7.2. שימוש במודולים

כדי להשתמש במודול, תשתמש בפקודה use. ככה:

```
use LWP::Simple;
```

כאן אני מבקש מהקומפילר שיחבר לתוכנית מודול. המודול הספציפי הזה הוא לגישה לדפים באינטרנט. בתוך המודול כתוב איזה פונקציות ואילו משתנים הוא ייתן לי. למשל, המודול הזה נותן פונקציה הנקראת בפשטות get, שמקבל URL, ומחזירה את הדף שבאותו מיקום.

```
$content = get("http://www.google.com/");
```

אחרי הפקודה הזאת, בתוך $content יהיה כל הדף הראשי של גוגל.

כאשר עושים use כמו למעלה, אזי המודול נותן לך איזה פונקציות שהוא רוצה. אבל יש פונקציות שהוא ייתן לך רק אם תבקש במיוחד:

```
use LWP::Simple qw{$ui};
```

כאן אתה אומר לו, אני יודע שיש לך בפנים את $ui, תן לי אותו!

7.3. מודולים כאובייקטים

אין בפרל אובייקטים אמיתיים. במקום זה, משתמשים במודולים בתור אובייקטים. לדוגמא:

```
use FileHandle;
```

זהו מודול שנותן גישה נוחה יותר לקבצים מאשר האופציות הפנימיות של פרל. (במיוחד אם אתה משתמש בכמה קבצים בו זמנית) כדי להגדיר באמצעותו מחזיק-קובץ, עושים:

```
$fh=new FileHandle;
```

ואז, כדי לפתוח את הקובץ:

```
$fh->open("> tmp.txt");
```

או פשוט:

```
$fh=new FileHandle("> tmp.txt");
```

שים לב רגע להבדל בין מודולים רגילים למודולים אובייקטים. מודולים רגילים מוסיפים לך עוד פונקציות לסביבת ההרצה, כמו בדוגמא שבסעיף הקודם. לפני שקראנו למודול לא היתה פקודת get, ואילו עכשיו יש. לעומת זאת, מודולים שהם אובייקטים לא מוסיפים לך פונקציות, אלא מחכים שתעשה להם new ודרך האובייקט תעבוד. כמובן, יש מודולים שעובדים בשני הדרכים, גם מוסיפים פונקציות וגם עובדים כמו אובייקט, אבל זו זכותם...

7.4. מודולים פופולריים

יש בעיות נפוצות יותר, ויש בעיות נפוצות פחות. בהתאם, יש מודולים שכולם משתמשים, ויש מודולים שבאים לתת מענה לבעיה ספציפית נידחת. הבאתי פה רשימה של המודולים היותר פופולריים, בהנחה שתזדקק להם בשלב זה או אחר.

תיאור	שם מודול
גישה לאינטרנט (בפרוטוקול http)	LWP::Simple
גישה למסדי נתונים	Dbi
גישה לתיבת הדואר (על יוניקס)	Mail::Box
כתיבת תוכניות CGI	CGI
גישה נוחה לקבצים	FileHandle
גישה למשתני הסביבה	Env
מידע על איך פרל קומפלה	Config

7.5. תוכניות דוגמא

```
#!/usr/bin/perl -w
use strict;
use FileHandle;
my $handle = new FileHandle;
$handle->open("+> output") or die "File won't open!\n";
$handle->print("A line in the river\n");
print $handle "And a line under the river\n";
seek($handle, 0, 0);
print "I printed: ", <$handle>;
$handle->close();
```

זוהי תוכנית המשתמש במודול FileHandle. אפשר לראות שמשתמשים במשתנה $handle גם בתור מחזיק קובץ רגיל, וגם כמו אובייקט. (כמו שאפשר לראות בשתי ההדפסות)
עוד הערה, זה אנחנו פותחים את הקובץ באמצעות סימון מיוחד, '+>'. זה פתיחה עבור כתיבה וקריאה גם יחד. זה מאפשר לנו לעשות את פקודת ה-seek בתוכנית. התוכנית תפתח קובץ חדש (או תדרוס קובץ קיים) בשם output, תדפיס לתוכו שתי שורות, תקפוץ חזרה לתחילת הקובץ, ותדפיס את מה שהודפס לתוכו.

```
#!/usr/bin/perl -w
use strict;
use LWP::Simple;
my $url="http://www.altavista.com/web/results?".
        "q=perl&kgs=0&kls=0&avkw=aapt";
my $i=1;
my ($content, $next_url);
while ($url) {
    $content=get($url);
    $i++;
    print $url . "\n";
    open(my $fh, ">", "altavista$i.html");
    print $fh $url, "\n";
    print $fh $content;
    close($fh);
    if ($content=~m|<a href="([^<>"]+)"[^<>]*>\[Next
&gt;&gt;\]</a>|) {
        $next_url=$1;
        $url="http://www.altavista.com".$next_url;
        sleep(10);
    } else {
         $url=undef;
    }
}
```

תוכנית המוציאה את כל הדפים שאלטא-ויסטא מוצאת על פרל. (היא שומרת את דפי אתר האלטא-ויסטא עצמם) אפשר לקחת את זה עוד צעד, ולהדפיס את ה-URL שנמצאו, (ע"י עוד פקודת while בסקריפט שמחפשת לינקים בתוך הדף) אבל זה תעשה בעצמך. התוכנית עובדת בעזרת LWP::Simple, המספקת לנו פקודת get, שמביאה דף מהאינטרנט. יש גם פקודת getstore, שמביאה דף מהאינטרנט ושומרת אותו לדיסק. יש עוד הרבה, אבל נסתפק בזה.

8.אינטרנט

8.1. הקדמה

בראשית היה האינטרנט. ובאינטרנט היו שרתים, והשרתים שלחו ללקוחותיהם קבצים סטטיים מהדיסק הקשיח, ויהי ערב ויהי בוקר יום אחד.

ויאמר העם: לא טוב לנו שהקבצים סטטיים. ויברא אלוהים את ה-Common Gate Interface, (או בקיצור CGI) ותשקוט הארץ שלוש שנים.

ויתאנה העם על ה-CGI, כי איטי הוא, ואוכל משאבים. ויחר אף אלוהים, ויברא את ה-ASP ואת ה-JSP ואת ה-Templates, למען ענותם ולא יוכלו דבר עוד איש אל אחיו. וישב כל תכנת בנישה הנוחה לו, ויעסיקו עצמם במלחמות דת איש עם אחיו, ולאלוהים היתה המנוחה.

8.2. פרוטוקול *HTTP* על קצה המזלג

קודם כל יש את השרת, שמחכה ללקוחות. בצד השני יש את הלקוח, שרוצה איזה משאב מהשרת. הלקוח מתחבר לשרת, ושולח בקשה. הוא אומר, שלום שרת. אני לקוח מסוג (אינטרנט אקספלורר, מוזילה..) ומעוניין בקובץ (`/users/list.html`) הנמצא על שרת בשם (my.domain.com) והבקשה היא בשיטת (Get, Post). אני מכיר את השפות (אנגלית, עברית, ...), ותומך בעוד כמה דברים. (אתה לא חושב שאני אפרט הכל, נכון?)

השרת מקבל את הבקשה, וחושב עליה. האם מותר למשתמש הנ"ל לראות את המשאב הזה? האם המשאב הזה בכלל קיים? מאיפה אני משיג את המידע הדרוש?

אוקי. עכשיו נגיד שההרשאות בסדר, והשרת מוכן להעביר את המידע ללקוח. רגע. לאט. קודם כל צריך לשלוח ללקוח מידע על מה הוא הולך לקבל.

שלום לקוח יקר. בקשר לבקשתך, היא (מאושרת, אין לך הרשאות, לא קיימת, יש לי בעיה פנימית ואני לא יכול לספק לך את המידע כרגע). המידע הזה הוא באורך (מספר) בתים, ובקידוד (כלשהו). המידע לא השתנה מאז (אתמול, לפני שנתיים).

עכשיו השרת שולח פעמיים אנטר ("`\n`") כדי לומר ללקוח שפה נגמרת הכותרת ומתחיל המידע האמיתי, ושולח את המידע.

עכשיו לשיטות בקשה: יש שתי שיטות. (עיקריות) GET ו-POST.

שיטת GET, אם פעם ראית כתובת בדפדפן שלך שנראית ככה:

http://my.domain.com/some/page.html?page=1&cat=normal

זה שיטת .GET כל הפרמטרים מופיעים פשוט בכתובת, כאשר סימן שאלה מסמן את תחילת הפרמטרים, ו-& מפריד בין פרמטרים .

בשיטת POST אתה לא רואה את הפרמטרים בשורת הכתובת. הם מועברים בהחבא. היתרון הוא שזה נקי יותר, וכן אפשר להעלות קבצים בעזרת השיטה.

מבחינת המתכנת שיושב וכותב את התוכנית בשרת, רוב הכלים המודרניים מחביאים את השיטה של הבקשה, ונותנים מנשק אחיד לקבלת המידע.

8.3. המודול *CGI*

זוהי הצורה הוותיקה ביותר לכתיבת תוכניות על השרת. בפשטות, מדובר בקובץ הרצה רגיל, שהשרת מריץ בכל פעם שהוא מתבקש לספק את המידע. השרת מאכיל מצד אחד את הסקריפט בנתונים, (דרך הסביבה, או דרך פרמטרים, או דרך ה-stdin) ומה שהסריפט מוציא הוא שולח למשתמש.

ומכיוון שאנחנו שמנים ועצלנים, יש לנו את המודול CGI שעושה את כל העבודה השחורה בשבילנו. דוגמא לסקריפט:

```
#!/usr/bin/perl -w
use strict;
use CGI;
use CGI::Carp 'fatalsToBrowser';
```

```
my $q = new CGI;
my $user_name = $q->remote_user();
my $filename = $q->param('filename');
print $q->header;
print "<html><head><title>Your file</title></head><body>
Hello $user_name !<br>
You have requested file: $filename !<br>
</body></html>";
```

זהו סקריפט CGI פשוט, המקבל את שם המשתמש מהזיהוי משתמש הבסיסי של השרת, מקבל בתור פרמטר (ולא משנה לנו איך הפרמטר הזה התקבל) שם של קובץ, ומדפיס למשתמש את השאילתה חזרה.

שורה אחר שורה: בהתחלה יש לנו את הקידומת הרגילה של כל תוכנית פרל, עם use strict. סטנדרטי.

השורה use CGI נותנת לנו להפעיל את המודול, ואילו `use CGI::Carp 'fatalsToBrowser'` אומר לתוכנית, שאם יש שגיאה פאטלית, (התוכנית מסתיימת בצורה לא תקינה) אז שישלח הודעה יפה ומסודרת למשתמש, ולא משהו שהוא לא יכול לקרוא.

השורה הבאה `my $q = new CGI;` יוצרת אוביקט חדש. כבר פה המודול משתלט על העניינים, קורא את כל הפרמטרים הרלוונטיים, משתני הסביבה וכל מה שהוא צריך, והוא כבר מוכן לעבודה.

וכבר בשתי השורות הבאות אנחנו מפעילים אותו. הראשונה שואלת מה שם המשתמש שניגש לקובץ הזה (בהנחה שהזיהוי (Authentication) הבסיסי של השרת פועל) והשניה מקבלת פרמטר בשם filename שהמשתמש שלח עם הבקשה.

עכשיו כשאני יודע מי המשתמש ומה הוא רוצה, אני יכול לומר לו את זה. אבל קודם כל – צריך לשלוח קידומת לתשובה. ואת זה עושה השורה `print $q->header;`. שים לב ש-CGI לא מוציא מידע ישירות למשתמש, אלא רק מכין את התגובה הרלונטית, ונותן לך לשלוח אותה. ככה הוא עובד.

עכשיו כשגם את הקידומות שלחתי, הגיע הזמן לשלוח למשתמש את קובץ ה-HTML עצמו. סיימנו. רק שים לב שבתשובה יש גם את השם של המשתמש וגם את הקובץ שהוא רצה. את הקובץ עצמו לא שלחנו לו. שימשיך לחלום.

8.4. טפלייטים

כמו שראית בסעיף הקודם, ה-CGI פשוט מדפיס את קובץ ה-HTML. זה בסדר כל עוד מדובר במקרה פשוט, בו כל ההדפסה המיועדת נמצאת בסוף הקובץ. אם מדובר על מקרה מסובך יותר, בו צריך לחשב דברים תוך כדי הדפסה, יכול לצאת סקריפט מכוער מאוד, כאשר קטעי HTML וקטעי קוד מעורבבים בצורה מטורפת.

כדי להפריד את הקוד מה-HTML, הומצאו הטמפלייטים. בסגנון העבודה הזה, קובץ ה-HTML נשמר בנפרד, והקוד רץ בנפרד. ובתוך קובץ ה-HTML יש סימונים מיוחדים, שמתורגמים לפני שליחת הקובץ ללקוח. הדוגמא הבאה משתמשת בחבילה הנקראת Template Toolkit:

```
#!/usr/bin/perl -w
use strict;
use CGI;
use CGI::Carp 'fatalsToBrowser';
use Template;

my $q = new CGI;
my $user_name = $q->remote_user();
my $filename = $q->param('filename');
print $q->header;

my $tamplate_directory = "/home/semuel/tt";
my $config = {INCLUDE_PATH => $tamplate_directory};
```

```
my $template = Template->new($config);

my $vars = {
      filename => $filename,
      user_name => $user_name,
};

# specify input filename, or file handle, text reference,
etc.
my $input = 'print_that.html';

# process input template, substituting variables
$template->process($input, $vars) || die $template-
>error();
```

והקובץ print_that.html מכיל:

```
<html><head>
<title>Your file</title>
</head><body>
Hello [% user_name %] !<br>
You have requested file: [% filename %] !<br>
</body></html>
```

עכשיו נסביר. עד הקטע של הדפסת הכותרת, (print $q->header) הכל בדיוק אותו הדבר. (חוץ משורת ה-use Template)

שלושת השורות הבאות, זה סידור המנוע של הטמפלייט לקראת העבודה. קודם אני אומר לו איפה כל הטמפלייטים נמצאים, ואז מייצר אובייקט Template חדש.

אבל זה לא הכל. הטמפלייט צריך לדעת גם איפה נמצא קובץ ה-HTML, ומה להכניס לתוכו. את שם הקובץ הוא פשוט מקבל בתור פרמטר, ואת מה להכניס לתוכו הוא מקבל בצורת האש, המכיל זוגות שם וערך. ובסוף יש שורת process, שמקבלת את כל המידע ומדפיסה (בעצמה) את הפלט ללקוח.

עכשיו אם תסתכל על הקובץ print_that.html, תראה משהו דומה מאוד לסוף של ה-CGI מהסעיף הקודם. רק במקום $user_name יש [% user_name %]. המנוע של הטמפלייט בעצם מדפיס את כל הקובץ ללקוח, עד שהוא נעצר בסוגריים המרובעים הללו עם האחוזים. פה במקום להדפיס, הוא מסתכל בהאש שנתנו לו, מוצא מפתח שנקרא user_name, ומכניס במקום הסוגריים את הערך המתאים.

זו כמובן דוגמא פשוטה מאוד, המשלבת רק שני משתנים בתוך קובץ HTML. השפה של Template Toolkit היא מאוד רחבה, וכוללת משפטי תנאי, הכללת קבצים אחרים, ועוד הרבה דברים טובים. כמובן יש טמפלייט אחרים (הבדיחה טוענת שכל תכנת פרל כתב פעם מנוע טמפלייט משלו. למזלי אני ניצלתי מגורל זה. אני עצלן מדי. ;) שכל אחד עובד קצת שונה, אבל זה כבר מחקר נפרד.

רק שים לב, שעדיין מדובר פה על תוכנית CGI. רק שבמקום להדפיס בעצמנו את ה-HTML נתנו למנוע טמפלייט לעשות זאת בשבילנו.

8.5. דפים משולבי קוד

בוא נסה לעבוד על הבעיה מכיוון אחר. במקום שתוכנית ה-CGI תדפיס את קובץ ה-HTML בחתיכות, בוא נהפוך את היוצרות. הקובץ עצמו יהיה קובץ ה-HTML, ובתוכו נשתול חתיכות קוד.

הגישה הזאת ממומשת, למשל, ע"י ASP. ברוב המקרים לא עובדים פה תחת CGI, אלא בסביבה אחרת שמראש יודעת שמדובר בקבצי HTML שבתוכם מושתל קוד. על סביבה כזאת נדבר בסעיף הבא. בנתיים, דוגמא:

```
<html><head>
<title>Your file</title>
```

```
</head><body>
Hello <% $r->connection->user() %> !<br>
You have requested file: <% $ARGS{filename} %> !<br>
</body></html>
```

זאת דוגמא הכתובה ב-Mason, שמהווה את הסביבה בה הקובץ הזה רץ. מכיוון שהסביבה מראש יודעת שמדובר בקובץ שכולל בתוכו פרל, היא מראש קוראת את כל הפרמטרים לתוך ARGS% ושולחת כותרת מתאימה. ועכשיו כמו במקרה עם הטמפלייט, הסביבה שולחת את הקובץ ללקוח, אבל את מה שתחום בתוך <% %> היא מריצה, ושולחת את התוצאה.
גם פה אפשר לעשות את כל מה שמנוע טמפלייטים עושה, רק שהסגנון הוא יותר פרלי.
הגישה הזאת מועדפת בד"כ ע"י מתכנתים שכותבים את האתר שלהם בעצמם, כיוון שהם כותבים את הלוגיקה בפרל בתוך ה-HTML. הגישה של הטמפלייט מועדפת בד"כ כאשר יש איש אחד שכותב את ה-HTML (ולא יודע פרל) ואיש אחר כותב את הפרל. ככה לזה שלא יודע פרל אין קטעי פרל שהוא לא מבין תקועים בתוך הקובץ.

8.6. ומה עוד

בסעיף השלישי הסברתי על CGI. בכל פעם שקובץ CGI מבוקש, השרת מריץ אותו ומדפיס את הפלט. הרצת תוכנית וסגירתה, כידוע, דורש הרבה ממערכת ההפעלה. היא צריכה לטעון פרל עצמו, לטעון את הקובץ, לקמפל אותו, להקצות לו זיכרון, להפעיל אותו ובסוף גם לסגור אותו. שלא לדבר אם הקובץ רוצה להתחבר למסדי נתונים ומשאבים אחרים.
מצד שני, היתרון הוא שהשרת לא צורך יותר זיכרון מהרגיל, (כי ה-CGI מקבל זיכרון משלו ממערכת ההפעלה) לא קשור רגשית אל ה-CGI, (אם קורת תקלה קולוסאלית, ה-CGI קורס באופן מרהיב, אך השרת לא אכפת לו. הוא רק משדר ללקוח שקרתה תקלה) ובכל פעם שה-CGI נסגר הוא לוקח עימו את כל הזיכרון שלו, כולל כל השגיאות והבעיות שיכלו לצוץ.
אבל בגלל שבכל זאת אנחנו רוצים לעבוד מהר יותר, המציאו את mod_perl. זה מודול של Apache, שנותן לו להריץ תוכניות פרל מבלי להצטרך לטעון אותם בכל פעם מחדש. יש אינטרפרטר אחד של פרל, שבכל פעם מקבל קובץ אחר להפעלה. הוא גם זוכר את הקבצים שהוא כבר הריץ, ככה שהשרת לא צריך לקמפל כל פעם מחדש את הסקריפט, אלא פשוט מריץ אותו שוב.
היתרונות: קודם כל מהירות. דבר שני, השרת יכול עכשיו לזכור דברים, כמו למשל אפשר לפתוח פעם אחת קישור למסד נתונים, ולהשתמש באותו הקישור לכל אורך חיי השרת.
בונוס נוסף: בגלל שפרל משולב עמוק לתוך Apache, אז יש גישה להרבה דברים שבתור CGI לא היה לנו. למשל, אפשרות לשנות את ה-url המבוקש, לפני שהשרת מתחיל ליצר את הפלט ללקוח. ועוד כל מיני דברים מתוחכמים.
החסרונות: צורך הרבה זיכרון. בגלל שפרל עכשיו הוא חלק מהשרת, השרת מחזיק אותו כל הזמן בזיכרון. מה שעושה אותו מנופח יותר, אפילו אם עכשיו הוא שולח איזה איקון ללקוח.
כמו כן, הוא זוכר דברים. שאלה אופיינית: למה השרת עושה <משהו> בפעם הראשונה שאני ניגש אליו, אבל בפעם השניה והלאה הוא עושה <משהו אחר>. התשובה היא כי יש לו משהו בזיכרון שנשאר מהפעם הראשונה.
לסיכום, שווה להשתמש בזה. יש המון יתרונות וחסרונות, אבל אם תכתוב את התוכנית כיאות, (ותתקל כמה פעם בבעיות, ותפתור אותם) ותשתמש בכמה טיפים שיחפו על החסרונות, רק תרוויח.

9.עברית בפרל

9.1. הקדמה

חסר לנו משהו בכל הסיפור. הממ... עברית! לעבוד באנגלית זה נחמד, אך אנחנו בישראל. אנחנו רוצים לקרוא, לכתוב ולעבד עברית!
לרוע מזלנו, המחשב בד"כ עובד משמאל לימין. כדי לשכנע אותו לעבוד מימין לשמאל, דורש עבודה. כדי לשכנע אותו לעבוד דו-כיווני, זה בכלל מסוכן.
אבל קודם כל אתה צריך כמה מושגי יסוד.

9.2. מושגי יסוד

בתחלה היה האסקיי. (ASCII) אסקיי זוהי טבלה הממפה מספר (מ-0 עד 127) לתו, ומכילה את כל הספרות, האותיות הגדולות והקטנות באנגלית, סימנים נפוצים, ועוד כמה תווים שונים.
ואז בא אדם חכם (?) ואמר, בואו נוסיף פה עברית. טבלת האסקיי היא רק עד 127, כלומר רק 7 ביט, ולנו יש בכל בית (Byte) שמונה ביטים, כלומר עד 255, בואו נדחוף את האותיות העבריות מעל 127, והכל יהיה טוב.
צצו עם זה שלוש בעיות: אחת, שגם הצרפתי, הגרמני, הספרדי והטיוואני חשבו ככה. וכולם שמו את האותיות שלהם באותו תחום. (128-255) כלומר, לאותו מספר היו כמה אפשרויות לתצוגה.
הבעיה השניה היא שבגלל שיהודים לא יכולים להסכים אפילו על השעה, היו כמה יהודים ששמו את האותיות העבריות בתחום ההפקר, וכל אחד שם אותם במיקום שונה. כלומר, לאותו תו היו כמה אפשרויות למספרים.
הבעיה השלישית היא שיופי שלימדנו את המחשב את האותיות העבריות, עדיין נשאר לנו ללמד אותו לדבר מימין לשמאל.
בסופו של דבר, הטבלאות השונות הסתדרו במה שנקרא דפי-קידוד. (Code-Page) לכל שפה יש טבלה משלה, המקצה מספר לכל תו. כל דף מזוהה ע"י מספר. בעברית למשל יש את iso-8859-8 ואת CP1255. כל אחד משניהם מצה מקום לעברית, אבל במקום שונה מהדף השני.
חסרון אחד ברור של השיטה הזאת הוא שאי אפשר להציג כמה שפות באותו המסמך. כי כל השפות דורכות אחת על השניה. חסרון אחר זה שמישהו (מי?) צריך לומר למחשב באיזה דף קידוד אנחנו רוצים להשתמש.

9.3. יוניקוד

לעזרתנו בא איגוד היוניקוד. (UniCode) איגוד היוניקוד שם לו למטרה להקצות מספר אחד לכל אות שקיימת, ולא משנה באיזו שפה. כמובן שעבור זה צריך יותר משמונה ביטים. אז מה.
האיגוד יצר טבלה ענקית, הכוללת את כל הסימנים בכל השפות. כמובן, שום דבר לא מושלם, ויש כמה מקרי קצה, (של תו שיש לו כמה מספרים, ומספר שיש לו כמה תווים) אבל בגדול, זה עושה את חיינו טובים יותר.
עדיין, הטבלה הזאת לא מיתרגמת ישירות לדף-קידוד. וזה מהסיבה בפשוטה שהיא גדולה מדי. ולתו בודד יש רק שמונה ביטים, לא יותר.
אז מה עושים? יש שתי כיוונים עיקריים:
שיטה אחת אומרת: מהיום, לתו יש יותר משמונה ביטים. מהיום, יהיה לו 16 ביט. או אפילו 32 ביט. אלה שיטות הנקראות ucs2 ו-ucs4, בהתאמה.
היתרון שלהם זה שלכל תו יש מספר ביטים קבוע. זה חשוב בטיפול בקבצים, שאם אתה רוצה לגשת לתו השלושים ושמונה, לא מעניין אותך מה יש לפניו. אתה פשוט קופץ לבית ה-38 כפול 2 או 4. החיסרון הוא שלכל תו יש מספר ביטים קבוע. אם אני רוצה לכתוב רק אנגלית, אני עדיין מבזבז 4 בתים על כל אות, בעוד שבאסקיי זה היה רק בית אחד על כל אות.
שיטה אחרת אומרת: אנגלית נשארת בית אחד לכל תו. רק אם אני אצטרך, אני אוסיף בתים. ולזה נשתמש בביט השמיני של התו. אם הוא דלוק, (כלומר, ערכו 1) אזי אני מוסיף גם את הבית הבא לתו. אם הוא כבוי, אני נשאר עם בית אחד לתו אחד. שיטה זאת נקראת utf8.
היתרון הוא בקומפקטיות, אם אני כותב אנגלית, יש לי בית אחד לכל תו. החיסרון הוא באורך תו משתנה. אני כבר לא יכול לדעת מראש איפה נמצא התו ה-38.

9.4. חזרה לפרל

כמו בהרבה מקרים, בפרל בחרו לשמר את הקיים. כלומר, מה שהיה עד כה, ימשיך לעבוד בלי שינוי. כלומר, אם התוכנית שלך כתובה באנגלית בלבד, וכן הקלט פלט שלה הוא באנגלית, הכל יעבוד כמו קודם.

יותר מזה. ברירת המחדל של פרל היא להניח שהתוכנית שלך כתובה בקידוד הנקרא latin1, שזה אנגלית פלוס סימנים נוספים. אם תנסה לדחוף כמה אותיות בעברית, אז בתוך מחרוזת זה כנראה יעבוד, מחוץ למחרוזת אלוהים יודע.

כדי לומר לפרל שאתה רוצה לעבוד יוניקוד בתוך הקוד, תכתוב:

```
use utf8;
```

ועכשיו אתה רק צריך לדאוג שהעורך שאתה משתמש בו באמת שומר את הקוד שלך בפורמט utf8. אחרת, מה עשינו?

כדי לומר לפרל שאנחנו רוצים לכתוב לקובץ ביוניקוד, יש את הפקודה binmode. ככה:

```
binmode(STDIN, ":utf8");
```

וזה יגיד לפרל להתייחס לקלט שלו כמקודד ב-utf8. אפשר להפעיל את אותה הפקודה על הפלט ועל כל קובץ פתוח. כדי לפתוח קובץ עם קידוד כבר, אפשר לכתוב:

```
open(my $fh, "<:encoding(cp1255)", "file");
open(my $fh, ">:encoding(UTF-8)", "file");
open(my $fh, "<:utf8", "file"); # utf8 have it's own
shortcut
```

הרעיון הוא להודיע לפרל באיזה קידוד המידע שהוא מקבל, והוא אוטומטית ימיר אותו לקידוד פנימי שלו (utf8). אני לא מגלה מה הפורמט הפנימי הזה (utf8), כדי שלא תסתמך עליו בהמרות אלא תודיע ספציפית לפרל מה הקידוד. בנוסף, צריך להודיע לפרל מה הקידוד הרצוי לפלט, והוא אוטומטית ימיר מהפורמט הפנימי הלא ידוע (utf8) לפורמט המבוקש.

כאשר פרל מקדד מחרוזת לפורמט הפנימי שלו, הוא מסמן את המחרוזת באמצעות דגל מיוחד. מרגע זה והלאה, כל פעולה המבוצעת על המחרוזת, עם זה חיתוך או ביטוי רגולרי, מודעת לכך שמדובר במחרוזת בה כל תו יכול להיות עם יותר מתו אחד, ומתייחסת בהתאם.

שים לב שהדגל הזה מדבק. אם תחבר שתי מחרוזות, אחת עם הדגל והשניה בלי, אז פרל ימיר את המחרוזת השניה לפורמט הפנימי שלו, ואז יחבר. אם היה לך במחרוזת השניה מידע שהוא לא אסקי, רוב הסיכויים שתקבל סימנים מוזרים.

אז תזכור - להודיע לפרל באיזה קידוד כל מחרוזת כמה שיותר מוקדם.

ואם אתה רוצה ידנית להעביר חתיכת טקסט מקידוד אחד לשני, יש מודול הנקרא Encode, שיעשה עבורך את העבודה:

```
use Encode qw{from_to encode decode};
# from iso-8859-8 to Perl's internal encoding (adding the
flag)
$string = decode("iso-8859-8", $octets);
# from Perl's internal encoding to iso-8859-8 (removing
the flag)
$octets = encode("iso-8859-8", $string);
# from legacy to utf-8 - "in-place"
from_to($data, "iso-8859-8", "utf8");
```

השורות לעיל ממירות לפורמט הפנימי של פרל ובחזרה לקידוד המקורי. בנוסף אפשר להמיר מידע מקידוד לקידוד, בלי ליצור מחרוזות ביניים, בעזרת הפקודה from_to. הפקודה, בניגוד ל-decode, אינה מפעילה את דגל הקידוד על המחרוזת.

כמובן, אפשר להמיר מכל קידוד לכל קידוד. כדי לקבל את רשימת הקידודים שהמודול הזה מכיר, עשה:

```
@all_encodings = Encode->encodings(":all");
```

9.5. זרימת טקסט

שים לב שלא דיברתי בכלל על העניין של כיוון הכתיבה. סיכמנו רק איך מייצגים תו בודד.

במשך השנים קמו ונפלו כל מיני שיטות לבקרה על כיוון זרימת הטקסט. בסופו של דבר, השיטות מתחלקות לשתי ענפים: עברית ויזואלית ועברית לוגית.

עברית ויזואלית טוענת, שאם המחשב מציג הפוך את המילים, אזי אנחנו פשוט נהפוך מראש את המילים. ואז, כאשר המחשב יציג אותם הפוך, אז הפוך-על-הפוך שווה ישר. (שיטה מאוד יהודית, אם יורשה לי להעיר)
עברית לוגית טוענת, שאת המידע צריך לשמור ישר, ולא הפוך. תן לשכבות אחרות במערכת לעבוד עבורך ולהתיישר לבד.
אנחנו בעד עברית לוגית. זה גם עוזר לעיבודים על הטקסט. כלומר, תחשוב למשל על מערך של מילים בעברית. בעברית ויזואלית האות הראשונה תהיה בסוף, ובעברית לוגית האות הראשונה תהיה בהתחלה. אם תריץ sort על המערך, באיזה משני המקרים המחשב יעשה מה שהתכוונת?
בפרל אפשר לתרגם בין עברית ויזואלית לעברית לוגית ע"י:

```
use Locale::Hebrew qw/hebrewflip/;
$visual = Locale::Hebrew::hebrewflip($logical);
```

ובכן, נותרה עדיין השאלה של איך משלבים עברית ואנגלית באותה השורה.
בעברית ויזואלית העניין למעשה די פשוט. ההנחה הבסיסית היא שהטקסט מוצג כאילו היה אנגלית, ולכן אפשר לחזות את ההתנהגות של הרכיב המציג. לעומת זאת, בעברית לוגית ההנחה היא שהרכיב המציג יודע איך לסדר את הטקסט. אבל איך הוא יסדר? לפי אילו כללים? נפתח פה פתח להמון אפשרויות. רק דוגמא קטנה:

```
Hello לאומש! How are you?
```

למעלה המידע לפי סדר הבתים בזיכרון. עכשיו הרכיב המציג צריך לתרגם. שתי האפשרויות הבאות תקפות:

```
Hello !שמואל How are you?
Hello שמואל! How are you?
```

האם הסימן קריאה שייך לטקסט באנגלית או לעברית? הרכיב המציג מחליט.
גם פה, במשך השנים קמו ונפלו המון אלגוריתמים. גם ביוניקוד ניסו להכניס אלגוריתם גלובלי שיטפל בבעיה. רק מה לעשות שכל אחד מימש מהאלגוריתם של יוניקוד רק מה שנראה לו הגיוני. ושוב, חזרנו לאי תאימות הישנה והמוכרת.
בכל מקרה, יש מימוש חופשי של האלגוריתם של יוניקוד הנקרא FriBidi. אני מקווה שאם תצטרך תשתמש בו, ולא תמציא עוד אלגוריתם שלא תואם לשום דבר אחר.

9.6. זהירות - הגדרות

מעבר לכל מה שדיברנו, פרל יורש הגדרות ממערכת ההפעלה. זה אומר, שאם מערכת ההפעלה מכוונת ל-utf8, אז פרל יניח שהקלט והפלט הסטנדרטיים שלו הם utf8. (אבל לא קבצים שהוא קורא וכותב לדיסק, אגב. לא יודע למה) הקוד עצמו עדיין מוגדר להיות אסקיי טהור.
בכל מקרה, אם אתה לא רוצה שיהיו אי נעימויות, ושהקוד יעבוד אותו הדבר בכל סביבה, ואם אתה עובד מחוץ לאסקיי הטהור, אזי כדאי שתגיד מפורשות לפרל באיזה קידוד מגיע כל מידע, ובאיזה קידוד הקוד עצמו. אחרת, במוקדם או במאוחר יהיו צרות.
מצד שני אתה יכול לספר לפרל שמדובר במידע בינארי, ולעשות מה שמתחשק לך, בזהירות.

9.7. ביטויים רגולריים

בסופו של דבר, זה מה שאנחנו רוצים לעשות. לנתח טקסט. אבל איך שואלים האם תו מסוים הוא בעברית? בשביל זה יש את מחלקות התווים. הנה:

```
use utf8;
$str = "Hello עולם!";
$str =~ m/(\p{InHebrew}+)/;
print $1, "\n";
```

אתה יכול לנחש מה התוכנית תדפיס. כמובן, יש מחלקת-תווים כזו לכל שפה. וכמובן, אם $str מסומנת בדגל הקידוד, (כמו בקוד שלפנינו) ביטוי רגולרי המכיל \w יתפוס עברית, אנגלית, וכל אות בכל שפה שהיא בעולם.
כמו כן אתה יכול פשוט לכתוב עברית בתוך הביטוי הרגולרי, בדיוק כמו שהייתה כותב באנגלית.

9.8. תוכנית דוגמא

תוכנית קטנה שמשחקת עם עברית:

```
#!/usr/bin/perl -w
```

```
use strict;
use warnings;
use Encode;

binmode(STDOUT, ":utf8");
my $orig_str = "some עברית writen";
print "undecoded: ", (join ", ", $orig_str =~ m/\w+/g)
,"\n";
my $dstring = decode("utf8", $orig_str);
print "decoded: ", (join ", ", $dstring =~ m/\w+/g)
,"\n";
my $cpstring = encode("cp1255", $dstring);
print "re-encoded ($cpstring): ", (join ", ", $cpstring
=~ m/\w+/g) ,"\n";
```

כדי להריץ את התוכנית אתה צריך להעתיק אותה לתוך עורך התומך ב-utf8, ולשמור אותה בקידוד זה.

ככה הפלט נראה בסביבת ההרצה שלי:

```
undecoded: some, writen
decoded: some, עברית, writen
re-encoded (some ????? writen): some, written
```

הסביבה בה אני עובד היא komodo, התומך בפלט utf8.

אני מתחיל ממחרוזת המקודדת כ-utf8, אבל בהתחלה פרל לא יודע מזה. אז הביטוי הרגולרי הראשון לא מזהה את העברית בתור אותיות. מצד שני המחרוזת המקודדת (decoded) מוצגת כמו שצריך, ובה הביטוי הרגולרי זיהה את האותיות העבריות כאותיות.

ומה זה ה-"????"? לקחנו את המחרוזת והמרנו אותה לקידוד עברית-חלונות. למחרוזת החדשה אין את הסימון של פרל שזו מחרוזת בפורמט הפנימי שלו. עכשיו התוכנית מדפיסה את המחרוזת לפלט הסטנדרטי, שמסומן כ-utf8. פרל מסתכלת על המחרוזת, רואה תווים שהם לא אסקי.

מניחה שמדובר בקידוד latin1, וממירה מהקידוד הזה ל-utf8.

אגב, ככה נראה הפלט של אותה התוכנית בקופסת דוס אצלי במחשב:

```
undecoded: some, writen
decoded: some, ???ש?¿?ס?, writen
re-encoded (some ?????????? writen): some, writen
```

בערך. במקום ה-ש וה-ס היו תווים מוזרים, שלא רצו לעבור בהעתק\הדבק.

איך זה יראה אצלך תלוי בסביבה שלך ובאם הצלחת לשמור את הקובץ בקידוד המתאים.

10. תרבות וסביבה

ככלל, מתכנתי פרל הגיעו לשפה הזאת במקרה. גם אני. במהלך הלימודים לתואר עבדתי באיזו חברה, שם נתנו לי ספר על פרל, משימה וקדימה תתחיל לכתוב. מה הנקודה שלי? שרוב המתכנתים בשפה רואים בה כלי שימושי ואפילו כייפי, אבל לא ילכו לשבור למישהו את האף כי הוא אמר עליה דברים רעים.
הפרק הזה מטרתו לעזור לך להתמצא בסביבה ולהבין כמה ביטויים נפוצים.

10.1. מלחמות דת

מלחמות דת (Religious Wars) הם וויכוחים עקרים שבד"כ מבוססים על העדפות אישיות. איזה עורך אתה מעדיף? vi או emacs? איזו שפה טובה יותר? C++ או Java?
לדעתי זו תופעה סוציולוגית מרתקת, שעדיין לא היה לי הזמן והכוח לרדת לפשרה. אנשים מתקוטטים על שטויות, על שאלות חסרות תשובה ועל העדפות אישיות כאילו אין מחר. למרבה המזל, תכנתי פרל בד"כ לא נכנסים לויכוחים אלו, כיוון שפרל זו השפה בה הם מתכנתים, ולא המזבח לו הם סוגדים. מצד אחד זה טוב, (הסביבה שקטה יותר והשכנים לא צועקים) מצד שני זה רע, כי המתכנת לא יגן על פרל ולא יפיץ את הבשורה ברבים. זה מה יש.

10.2. גישה

יש יותר מדרך אחת לעשות את זה.
או בראשי תיבות: TIMTOWTDI. בד"כ, פרל נותנת לך שבע דרכים לעשות את אותה הפעולה. חלק מהדרכים איטיות יותר, חלק מהירות יותר. חלק יפות וחלק לא.
הכלי הנכון למשימה.
כמו שאמרתי, תכנתי פרל הגיעו לפרל במקרה, ולכן אם יש לך שאלה, ומישהו חושב שזה רעיון טוב יותר לעשות את העבודה בכלי אחר או בשפה אחרת ולא בפרל, הוא יגיד את זה. כי פרל, כמה שאנחנו מספטים את השפה, היא רק עוד כלי. כלי חזק, אבל כלי.
קרא את המדריך המזורגג - RTFM.
העזרה של פרל היא די מקיפה. במקרה של שאלה פשוטה, לפעמים אנשים יעדיפו לשלוח אותך לקרוא את העזרה מאשר פשוט לענות על השאלה. ידוע גם כסינדרום "תן לאיש דג והשבעת אותו ליום. תן לו חכה ונתת לו מקצוע". כמובן שאני נגד זה, ומעדיף לתת את התשובה ולמטה בהערה להפנות לעזרה לקריאה נוספת. כל אחד והגישה שלו.
גולף.
מכיוון שיש יותר מדרך אחת לעשות כל דבר, יש דרכים עם קוד קצר ויש עם קוד ארוך. לשחק גולף משמעו לחפש את התשובה הקצרה ביותר. ברוב המקרים במשחקים האלו נוצר קוד מכוער ביותר, כי כל מה שמעניין את המשחקים זה שהוא יהיה קצר, ולא שיהיה מובן. ולמה לבזבז שתי אותיות על סוגריים אם אפשר בלי?

10.3. שיתוף ורשיונות

מתכנתים הם עצלנים אך גאוותניים. שונאים לכתוב קוד שכבר מישהו כתב משהו דומה פעם, אבל אוהבים להראות קוד שהם כתבו לאחרים. כדי לעזור למתכנתים להשאר ככה, נולד האתר CPAN. המאגר האולטימטיבי של מודולים של פרל. תוכל למצוא שם אלפי מודולים, בכל נושא שבעולם, בהם אתה יכול להשתמש באופן חופשי, כדי להשאר עצלן אך מועיל. ואם כתבת משהו משלך, ואתה מעוניין לשתף את העולם בבשורה, אתה חופשי להעלות מודול משלך.
רגע, רשיונות. אם תשים לב, רוב המודולים מצהירים שהם מופצים תחת רשיון "זהה לפרל עצמה". הרשיון של פרל הוא אינו הרשיון של GNU. ב-GNU אסור לך לשלב את התוכנה בתוכנה מסחרית. בפרל מותר. בפרל גם מותר לעשות שינויים ולהפיץ אותה כחבילה חדשה, כל עוד אתה לא טוען שזאת החבילה הרשמית של פרל. הרעיון הוא שתוכל לעשות מה שאתה רוצה, כל עוד אתה לא פוגע בקהילה. (ע"י הפצת חבילות "מזויפות" וכד')
הרשיון הוא חלק חשוב ממה שעשה את השפה למה שהיא כיום. שב תקרא אותו, הוא לא נושך.

10.4. גרסאות של פרל

השפה מתפתחת במשך השנים. כיום, היא נמצאת בגרסא 5, ולא נדבר על גרסאות קודמות, כי זה ישן ודפוק מדי. אבל גם בתוך גרסא 5 יש גרסאות שונות. ואלו הן:

גרסא 5.004: הגרסא הראשונה בסדרה של 5. תמצא אותה בעיקר במחשבי יוניקס ישנים, עם מערכת הפעלה שלא שודרגה כבר הרבה זמן.

גרסא 5.6: רוב המבנים המודרנים של השפה כבר קיימים. זוהי הגרסא המינימלית לשימוש בפרל כיום, אך גם התמיכה בגרסא הזאת הולכת ונעלמת.

גרסא 5.8: פרל מודרנית. כוללת תמיכה ביוניקוד, ב-Threads.

גרסא 5.10: בגרסא זאת הכיוון הוא שיפור נוחות השימוש בשפה. הוסיפו פקודות שמחליפות פקודות בעיתיות ישנות, הוסיפו כמה אופרטורים שעושים את החיים קלים יותר וכו'.

גרסא 6: פרויקט עתידי ושאפתני, בו מנסים לכתוב את השפה עצמה מחדש, להיפטר מכל מיני ירושות שפרל סוחבת מימי קדם ולהכניס תמיכה מסודרת באובייקטים. מעורבים בפרויקט הרבה שמות גדולים בעולם הפרל. (כמו לארי וול (Larry Wall), ההוא שכתב את פרל מלכתחילה)

11. מידע נוסף ועזרה

11.1 למה צריך עוד מידע

במסמך קטן זה ניסיתי לכלול את הבסיס של הבסיס של פרל. ניסיתי לכלול מספיק בשביל לתת לך, הקורא, כלים מספיקים עבור כתיבת תוכניות. אבל זה סתם מסמך. קיימים בפרל כל כך הרבה דברים שאפילו לא נגעתי בהם. אם אני אנסה אפילו לגעת בכל נושא, אפילו קצת, אני אסיים עם ספר בגודל מלא. בנוסף, גם נושאים שכתבתי עליהם, בד"כ לא כיסיתי הכל. למעשה, כיסיתי רק טיפה. אם תקרא את התיעוד של כל דבר שהזכרתי במסמך זה, על הפונקציה הכי זניחה שבעולם, אני מבטיח לך חגיגית שתמצא לפחות שלושה דברים שלא הזכרתי עליו.
ספר אני לא יכול לכתוב, אבל אני יכול להפנות אותך לספרים ולמקורות שונים, בהם תוכל למצוא מידע נוסף, לקרוא על נושאים שונים, וללמוד.

11.2 ספרים

הספר הרשמי של עולם הפרל הוא Programming Perl, בהוצאת O'Reilly. (כרגע במהדורה השלישית) הספר הזה הוא אחד הספרים המקיפים ביותר על יסודות התכנות בפרל. והוא אפילו מצחיק. אם אתה צריך לקנות ספר אחד, זה הספר.
הספר השני הוא Learning Perl, מאותה ההוצאה. נחשב גם לספר טוב, יותר בשביל מתחילים בעולם התכנות. מעולם לא קראתי אותו, כך שקשה לי להמליץ.

11.3 רשימות תפוצה

פרל מתברכת בקהילה גדולה של תכנתים, שישמחו לעזור לך בכל בעיה. למה שלא תצטרף לרשימת התפוצה הישראלית?

http://www.perl.org.il/

11.4 ערוצי צ'אט באינטרנט

אה, כאן יש קצת בעיה. אנשי פרל בערוצי צ'אט, בד"כ חסרי סבלנות למתחילים. הם בד"כ ישלחו אותך לקרוא את התיעוד, במקום לתת תשובה. אז אם אתה מחפש מקום לברר איזה חלק של התיעוד לקרוא כדי לפתור בעיה, אולי זה באמת המקום לשאול אותה...

11.5 התיעוד

עם פרל מגיע תיעוד. זהו התיעוד העדכני ביותר לכל נושא ופונקציה. התיעוד כתוב מאוד מתומצת, ואם אתה מחפש דוגמאות, לא יהיו שם כאלה, בד"כ. מה שכן, כל אופציה הקיימת בפונקציה תהיה כתובה שם.
כדי להעלות תיעוד של פונקציה, למשל chomp, כתוב בשורת הפקודה:

```
perldoc -f chomp
```

שם תראה תיעוד מפורט על מה הפונקציה עושה, איזה פרמטרים היא מקבלת, מה היא מחזירה, מה קורה אם היא לא מקבלת פרמטרים, וכו'. כדי להעלות תיעוד של מודול, כתוב בשורת הפקודה:

```
perldoc LWP::Simple
```

כדי לחפש מילים כתוב בשורת הפקודה:

```
perldoc -q keyword
```

ותקבל חזרה את רשימת המקומות בהם מילה זו נכתבה. קיים גם אתר שמכיל את התיעוד של פרל עצמה, אם אתה (משום מה) מעדיף לקרוא דפים יפים באינטרנט במקום טקסט מחלון דוס:
http://perldoc.perl.org/
ובאתר של CPAN אפשר לקרוא את התיעוד של המודולים השונים .

11.6. פרקי קריאה

אם יום אחד יהיה לך הרבה זמן, המון ריכוז וטונות סבלנות, אתה מוזמן בזאת לשבת ולקרוא את הפרקים למטה מהתיעוד. כל אחד מהפרקים הללו מפרט נושא אחד בפרל, בצורה שלמה ומקיפה. (ומכוערת) אז כדי שתדע איפה לחפש, הנה רשימת הנושאים: (עשה לעצמך טובה, תנסה קודם לקרוא את המדריכים הנגמרים בסימון "tut", כמו perlretut, perlreftut. אלה חדשים יחסית, ומסבירים בצורה טובה יותר מהרגילים)

נושא	מילת קוד
רשימת פרקי עזרה	perl
מדריך לביטויים רגולריים	perlretut
ביטויים רגולריים	perlre
משתנים מיוחדים בפרל	perlvar
הסינטקס של פרל	perlsyn
אופרטורים	perlop
פונקציות	perlsub
הפונקציות הפנימיות של פרל	perlfunc
מצביעים	perlreftut
פרמטרים עבור פרל עצמה. (לדוגמא, "-w")	perlrun
הוראות לקומפילר ומודולים סטנדרטים	perlmodlib
הסבר על הודעות שגיאה	perldiag

כדי להעלות הסבר על אחד מהנושאים כתוב:

```
perldoc subject
```

כדי לקבל את הרשימה המלאה:

```
perldoc perl
```

קריאה מהנה.

12. סיכום

12.1. תודות

אני רוצה להודות:
לגבור, על המוטיבציה לכתוב מסמך זה.
לעופר קיי, על תיקון השגיאות הגסות שעשיתי במהלך הכתיבה.
לשלמה יונה, על תמיכה מורלית, ועל פרק העברית.
לעודד רזניק, מחברת RAZ מערכות, על הנספח "מלכודות למתכנת המתחיל".
לשאר חברי קבוצת דיון פרל ישראל, על אספקה שוטפת של בעיות ופתרונות.
לצה"ל, על שסיפק מהנדס משועמם עם המון זמן פנוי שיכול לשבת ולכתוב מדריכים.
לחברת Siftology מנוחתה עדן, שהכירה לי את עולם הפרל.
וכמובן, להורי, שהביאוני עד הלום. :-)

12.2. ביבליוגרפיה

O'Reilly's Programming Perl, 2nd edition .

12.3. על מסמך זה

מדריך זה נכתב ע"י שמואל פומברג.
המדורה ראשונה שוחררה בתאריך 20.9.2003, והמהדורה השניה והנוכחית שוחררה בתאריך 20.5.2008.
דף הבית שלו למציאת הגרסא העדכנית ביותר הוא:
http://www.shmuelfomberg.com/perlhebtut/
ליצירת קשר, לעזרה ולתיקון שגיאות שמצאת במסמך, שלח מייל לכתובת:
shmuelfomberg@gmail.com