Bardzo krótki kurs Perla

... i trochę trobiazgów, między innymi make

Perl

- Practical Extraction and Report Language
- Opracowany w 1986 (Larry Wall)
- Następca (?) AWK i sed, poprzednik (?) Pythona, od którego jest szybszy i brzydszy.
- Składniowo nawiązujący trochę do AWK, C, powłoki (od której wziął na przykład odwrotne apostrofy).

Perlowe struktury danych

- Skalary (nazwa od $)

```
$val = "napis"
$val = 10.1
```

- Tablice (wektory, nazwa od @)

```
@studenci=("ala","ola","zbyś")
print $studentci[0]    # ala
studenci[3] = "beata"  # nowy element
@studenci=()           # zerowanie tablicy
```

- Tablice asocjacyjne (nazwa od %):

```
%emp =("Julie", "President",  "Mary", "VP");
print $emp {"Julie}; #wypisze  president
```

```
$emp{"John"} = "controller";
%emp =(); # empty hash
```

Operatory dla napisów

- Konkatenacja: znak `.`
- `print "hello"."world"` daje `helloworld`.
- Operator powtarzania `str x num` — wielokrotna konkatenacja napisu.

Operator porównania wartości skalarnych

operacja	**liczba**	**napis**
równe	`==`	`eq`
różne	`!=`	`ne`
mniejsze	`<`	`lt`
większe	`>`	`gt`
mniejsze równe	`<=`	`le`
większe równe	`>=`	`ge`

Wzorce

- Dopasowanie wzorców:

```
if ($var =~ m/pattern/) {...}
if ($var  !~ m/pattern/) {...}
```

- Podstawienie: `$var1 =~ s/pattern/replacement/;`

Funkcje

- Definicja funkcji użytkownik:

```
sub subname {
  Statement_1;
  Statement_2;
  Statement_3;
}
```

- Jak w `bash`-u: funkcja zwraca wartość zwracaną przez ostatnią jej instrukcje.

Wywoływanie funkcji

- `$var = myroutine( paramters);`
- `@array = myroutine( parameters);`

Argumenty

- Po wywołaniu podprogramu podprogramu następuje lista argumentów w nawiasach.
- Automatycznie przypisują się one do zmiennej `@_`
- Zatem kolejne argumenty to : `$_[0]`, `$_[1]`, ...
- Zmienna `@_` jest lokalna w procedurze
- Przekazywanie tablic (rownież asocjacyjnych) to przekazywanie referencji do tablic.

Jak łatwo opanować Perla?

- Wiemy jak działa `awk`, `sed`, `find`?
- Jeżeli tak, to programy w każdym z tych trzech języków można przetłumaczyć automatycznie na Perla (i poczytać).
- Służą do tego programy `a2p`, `s2p` oraz `find2perl`.

Przykładowy program w awk

```
BEGIN {x=0}
{Tab[Nr] = $0}
END {
   for (i=NR; i>0; i--)
      print Tab[i];
}
```

Wynik translacji do Perla

```
$, = ' ';                   # set output field separator
$\ = "\n";                  # set output record separator

$X = 0;

while (<>) {
    chomp;       # strip record separator
    $Tab{$Nr} = $_;
}

for ($i = $.; $i > 0; $i--) {
    print $Tab{$i};
}
```

Inny prosty przykład

- Program w awk

```
{print $2 " --- " $1}
```

- Translacja do Perla

```
# ...
while (<>) {
    ($Fld1,$Fld2) = split(' ', $_, 9999);
    print $Fld2 . ' --- ' . $Fld1;
}
```

Narzędzia wspomagające uruchamianie programów – kompilator

- Opcje kompilatora (dla `gcc` to opcje `-Wall`, `-pedantic`, `-ansi`).
- Makrodefinicje przydatne w debugowaniu: `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`.
- Makrodefinicja `assert` pozwala na testowanie warunków.
- Gdy zdefiniowana jest stała `NDEBUG` wówczas `assert` kompiluje się do pustego kodu.
- Stałą tę (i wszystkie inne) możemy zdefiniować w programi (`#define NDEBUG`) lub wywołując kompilator z opcją `-DNDEBUG`.

Debuger `gdb`

- Aby zeń korzystać, trzeba kompilować program z opcją `-g`.
- Uruchamiamy pisząc `gdb <nazwa-pliku-wykonywalnego>`.
- Polecenie `help` spowoduje wyświetlenie listy dostępnych poleceń.

Korzystanie z gdb

- Uruchamiamy program poleceniem `run`.
- Badanie zmiennych realizujemy za pomocą polecenia `print <nazwa zmiennej>`
- Polecenie `list` wyświetla fragment programu.
- Do wstawiania punktów przerwania służy polecenie `breakpoint` (można napisać `help breakpoint`).

Punkty przerwania

- Przerwanie w pewnej linii programu: `break 20`.
- Wznawiamy wykonywanie za pomocą polecenia `cont`.
- Za pomocą `condition` można uzyskiwać punkty przerwania „warunkowe” – tj. działające jedynie przy spełnionym warunku.
- Polecenie `ignore` pozwala zignorować pewną ilość przerwań w danym punkcie.

Narzędzia analizujące kod

- Program `ctags` – tworzy indeks funkcji (i innych obiektów) podając miejsca, w których dana funkcja jest definiowana.
- Program `cxref` – analogicznie, ale dla wszystkich symboli w programie.
- Program `cflow` – drukuje drzewo wywołań funkcji.

Działanie programu `cflow`

- Ponumerowane wiersze w raporcie, dla każdej funkcji wypisane funkcje w niej wywołane.
- Pierwsze wystąpienie funkcji w raporcie wraz z dokładnym opisem, dalsze wystąpienia – tylko numer linii.
- Przykładowy wydruk:

```
1     main: int(), <licznik.c 8>
2             fopen: <>
3             printf: <>
4             CzytajSlowo: int(), <licznik.c 150>
5                   getc: <>
6                   ungetc: <>
7             AnalizujSlowo: void(), <licznik.c 195>
8                   DodajSlowo: void(), <licznik.c 50>
9                         TworzSlowo: struct*(), <licznik.c 70>
10                              malloc: <>
```

```
11                          ZapiszNazwe: char*(), <licznik.c 136>
12                                  malloc: <>
13                                  strlen: <>
14                                  strcpy: <>
15                  CzyKluczowe: int(), <licznik.c 263>
16                          tolower: <>
17                          strcmp: <>
18                  AnalizujSlowo: 7
19              DodajSlowo: 8
   (...)
```

Program `ctags`

- Wytwarza dane w formacie akceptowalnym przez różne edytory (`vi`, `emacs`, `joe`, ...) lub do czytanie przez człowieka (opcja `-x`)
- Potem w `vim` można pisać: `:tag <nazwa-funkcji>` i przenosimy się do jej definicji.
- Umożliwia znajdywanie miejsc, w których dany obiekt jest definiowany (inteligentniejsze niż wyszukiwanie)
- Domyślnie tworzy plik `tags`, dla wielu plików wejściowych.
- Działa dla wielu języków, między innymi dla C, C++, Pascal, Perl, AWK, Python, PHP, Java, Tcl, język powłoki, ... (jest ich w sumie ponad 26)

Wynik progamu `ctags -x`

```
AnalizujSlowo function  195 licznik.c   void AnalizujSlowo(char *slowo) {
CzyKluczowe   function  263 licznik.c   int CzyKluczowe(char *slowo) {
CzytajSlowo   function  150 licznik.c   int CzytajSlowo(char *tab) {
DodajSlowo    function   50 licznik.c   void DodajSlowo(int typ, char *slowo)
DodajZmienna  function   31 licznik.c   void DodajZmienna(char *nazwa, int wa
MAX           macro       6 zadanie5.c  #define MAX 200 // Maksymalna dl
```

Program `crefs`

- Podaje linie (i plik) w której jest definicja, jak również inne wystąpienia danej nazwy.
- Linie z definicją oznaczane są znakiem `*`, znak `=` oznacza, że zmienna jest modyfikowana.
- Przykładowy wynik:

```
wart     licznik.c  TworzZmienna  85*   90
wartosc  licznik.c  DodajZmienna  31*   36    40
znak     licznik.c  CzytajSlowo  153*  155=  157   157   157
                                158=  158   158   160   164
                                165=  166=  166   168   173
                                176   176   176   176   176
                                177   178=  181   184
```

Profilowanie kodu

- Do profilowania kodu może służyć polecenie `prof` lub `gprof`
- Wykonuje on pewien program i następnie generuje raport mówiący o tym, ile i gdzie czasu program spędza.
- Nie zaszkodzi skompilować z opcją `-p` lub `-gp`.

Program `make`

- Pozwala określać zależności pomiędzy plikami i uaktualniać te pliki, które naprawdę tego wymagają.
- Danymi do programu `make` jest plik tekstowy zawierający zadania do wykonania (nazywane regułami).
- Polecenia zapisane są w języku powłoki.
- Domyślną nazwą pliku z zadaniami jest `makefile` (lub `Makefile`).

Format pliku `makefile`

- Plik jest ciągiem wpisów o następującej postaci:

  ```
  plik docelowy: pliki potrzebne do jego utworzenia
  <TAB> lista poleceń
  ```

- Pliki oddzielane spacjami, polecenia średnikiem
- Można pisać polecenia w różnych linijkach używając `;\`.

Znaczenie pliku `makefile`

- Za pomocą reguł określamy **co** należy robić, aby otrzymać plik (lista poleceń) ...
- ... i **kiedy** to robić:
  a) wtedy, gdy tworzony plik nie istnieje lub
  b) któryś z plików nań wpływających po wykonaniu `make` ma datę późniejszą niż tworzony plik.
- Wywołanie programu: `make plik-do-utworzenia`.
- `Plik-do-utworzenia` powinien występować po lewej stronie `:` w jakiejś regule.
- `make` bez parametrów oznacza polecenie wykonania pliku z pierwszej reguły.

Przykład

```
# Plik makefile lub Makefile
newfile: main.o p1.o p2.o /usr/lib/ll.a
     cc -o newfile main.o p1.o p2.o /usr/lib/bibl.a
main.o: main.c
     cc -c main.c
p1.o: p1.c
     cc -c p1.c
p2.o: p2.s
     as -o p2.o p2.s
clear:
     rm *.o
maybe: yes.h no.h
  cp yes.h no.h /usr/dd
```

Komentarz

- Nie jest konieczne by `nazwa-pliku` rzeczywiście oznaczała jakiś plik (`make clear`)
- Za pomocą znaku `#` piszemy komentarze
- Możliwe nietypowe wykorzystania (linia z `maybe`)
- Program `make` można wykorzystywać do tworzenia kilku wersji programu.

Inne często spotykane obiekty fikcyjne

- `clean` – usuwa utworzone pliki
- `clobber` – usuwa pliki i katalogi (odinstalowywanie)
- `install` – tworzy programy, kopiuje strony `man`-a, kopiuje program do odpowiedniego katalogu, etc.
- `all` – jak jest wiele obiektów, to tworzy je wszystkie.