

WSTĘP DO INFORMATYKI I PROGRAMOWANIA

Przemysław Kobylański

Katedra Podstaw Informatyki, Wydział Informatyki i Telekomunikacji,
Politechnika Wrocławska

6 października 2026

WYKŁAD 1.

Wprowadzenie

Plan wykładu

Wprowadzenie
Środowisko pracy
Instrukcje proste
Instrukcje warunkowe
Iteracje
Niezmienność i skończoność pętli
Złożone typy danych

Funkcje i procedury
Struktury dynamiczne – listy
Struktury dynamiczne – drzewa
Operacje wejścia/wyjścia
Rekurencja
Zasada dziel i zwyciężaj
Przegląd z nawrotami

Wprowadzenie

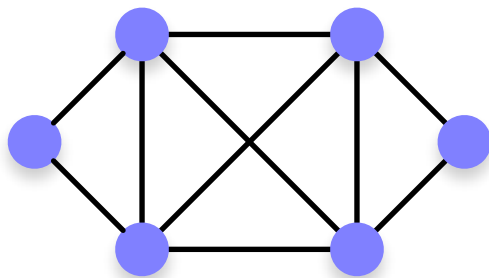
Algorytm

- ▶ „**Algorytm**, systematyczna procedura, która w skończonej liczbie kroków generuje odpowiedź na pytanie lub rozwiązanie problemu” – Britannica
- ▶ skończona liczba przypadków = zawsze istnieje algorytm
- ▶ algorytm można oceniać m.in. za pomocą czasu w jakim się wykonuje
- ▶ jeśli dla problemu jest algorytm, którego czas rośnie wielomianowo ze wzrostem rozmiaru problemu, to problem uznaje się za *łatwy do rozwiązania*
- ▶ jeśli dla problemu są tylko algorytmy, których czas wykonania rośnie wykładniczo, to uznaje się go za *trudny do rozwiązania*
- ▶ *algorytm decyzyjny* – odpowiada na pytanie TAK/NIE
- ▶ *algorytm obliczeniowy* – wylicza wynik

Wprowadzenie

Algorytm

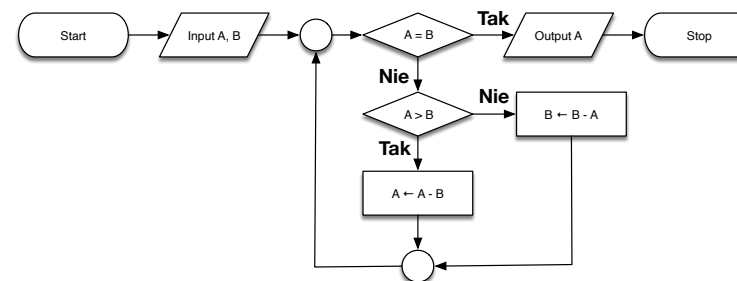
- ▶ przykładem problemu, który jest łatwy do rozwiązania jest znalezienie cyklu Eulera (przechodzi przez wszystkie krawędzie dokładnie raz)
- ▶ przykładem problemu, który jest trudny do rozwiązania jest znalezienie cyklu Hamiltona (przechodzi przez wszystkie węzły dokładnie raz)



Wprowadzenie

Algorytm

- ▶ przykładem algorytmu jest algorytm obliczający największy wspólny dzielnik dwóch całkowitych liczb dodatnich
- ▶ możemy go zapisać jednym zdaniem: *Dopóki obie liczby są różne, odejmuj od większej mniejszą.*
- ▶ albo narysować schemat blokowy:



Wprowadzenie

Algorytm

- ▶ albo w pseudokodzie:

```
while a ≠ b do
  if a > b then
    a ← a - b
  else
    b ← b - a
  end if
end while
```

- ▶ albo w języku Python:

```
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
```

Wprowadzenie

Algorytm

- ▶ albo w języku Ada:

```
while a /= b loop
  if a > b then
    a = a - b;
  else
    b = b - a;
  end if;
end loop;
```

- ▶ albo w języku C:

```
while(a != b)
  if(a > b)
    a = a - b;
  else
    b = b - a;
```


Wprowadzenie

Komputer

- ▶ Komputer jest urządzeniem, które automatycznie przetwarza dane.
- ▶ Przetwarzanie danych odbywa się zgodnie z algorytmem.
- ▶ Aby przetwarzać dane, komputer musi być wyposażony w jednostkę centralną.
- ▶ Jednostka centralna wykonuje operacje.
- ▶ Operacje i dane, na których one działają, są przechowywane w pamięci komputera.
- ▶ Więcej na kursie *Architektura komputerów* (semestr III).

Wprowadzenie

Język niskiego poziomu

- ▶ komputer zna tylko jeden język programowania – swój własny **język maszynowy**
- ▶ jednostka centralna komputera wykonuje programy w języku maszynowym tego komputera
- ▶ inny komputer może mieć inny język maszynowy, zatem programy w języku maszynowym nie są **przenośne**
- ▶ na przykład programy w języku maszynowym uruchamiane na komputerze **PowerBook**, wyposażonym w procesor **PowerPC G4**, nie uruchomią się na komputerze **MacBook** wyposażonym w procesor **Intel Core i5**
- ▶ język maszynowy składa się z podstawowych instrukcji jakie potrafi wykonać jednostka centralna

Wprowadzenie

Język niskiego poziomu

- ▶ przykładem instrukcji w języku maszynowym mikroprocesora 6502¹ jest następujący ciąg szesnastu bitów:

00001101 10101001

- ▶ zajmuje on dwa kolejne bajty pamięci: mniej znaczący (o mniejszym adresie) 0xA9 i bardziej znaczący (o kolejnym adresie) 0x0D
- ▶ bajt 0xA9 jest kodem instrukcji ładującej liczbę ośmiobitową znajdującą się w kolejnym bajcie do rejestru akumulatora
- ▶ bajt 0x0D jest liczbą ośmiobitową ładowaną do akumulatora
- ▶ w wyniku wykonania tej instrukcji w akumulatorze zostaje umieszczona liczba 13 (szesnastkowo 0x0D)

¹Popularny procesor z takich komputerów jak np. Atari, BBC Micro, Commodore albo konsoli Nintendo.

Wprowadzenie

Język niskiego poziomu

- ▶ instrukcje maszynowe w postaci ciągu bitów, nawet zapisane szesnastkowo, nie są czytelne dla człowieka
- ▶ dla języka maszynowego opracowano **język assemblera**, którym instrukcje zapisuje się używając łatwych do zapamiętania mnemoników
- ▶ w języku assemblera rozpatrywaną wcześniej instrukcję zapisuje się następująco:

LDA #13

- ▶ LDA jest skrótem od *LoaD Accumulator*, natomiast znak # oznacza, że następująca po nim liczba jest wartością umieszczaną w akumulatorze (dana)
- ▶ podobna instrukcja:

LDA 13

pobiera wartość z pamięci spod adresu 13 i umieszcza ją w akumulatorze (adres danej)

Wprowadzenie

Język wysokiego poziomu

- ▶ programowanie w języku asemblera jest żmudne gdyż dostępne są w nim jedynie elementarne operacje jakie może wykonać jednostka centralna komputera
- ▶ prosta operacja jak zwiększenie szesnastobitowej sumy o szesnastobitowy składnik (podstawienie $\text{suma} = \text{suma} + \text{składnik}$) wymaga siedmiu instrukcji maszynowych:

```
LDA suma
CLC                ; wyzerowanie bitu przeniesienia
ADC składnik       ; dodanie mniej znaczącego bajtu
STA suma
LDA suma+1
ADC składnik+1     ; dodanie bardziej znaczącego bajtu
STA suma+1
```

© 2024-26 Przemysław Kobylański

Wprowadzenie

Interpreter

- ▶ komputer nie może bezpośrednio wykonywać programów w języku wysokiego poziomu
- ▶ jedną z możliwości jest opracowanie **interpretera** języka wysokiego poziomu
- ▶ źródła programu przechowywane są w pamięci komputera
- ▶ interpreter czyta instrukcję programu jedną po drugiej, analizuje ją, następnie wykonuje
- ▶ interpretowane programy wykonują się powoli, gdyż jeśli jakaś instrukcja jest wykonywana wielokrotnie, to wielokrotnie musi być analizowana

© 2024-26 Przemysław Kobyłański

Wprowadzenie

Język wysokiego poziomu

- ▶ w roku 1954 opracowano pierwszy język wysokiego poziomu jakim był **FORTTRAN** (od ang. FORMula TRANslator)
- ▶ wybrane kolejne języki programowania:
 - 1956 **LISP** (Functional Programming)
 - 1958 Algol
 - 1959 COBOL
 - 1962 APL
 - 1962 Simula (Object Oriented Programming)
 - 1962 SNOBOL
 - 1964 **PL/I**
 - 1964 **BASIC**
 - 1970 **Pascal**
 - 1972 **C**
 - 1972 **Prolog** (Programming in Logic)
 - 1983 **Ada**
 - 1991 Python
- ▶ istnieją tysiące języków programowania

© 2024-26 Przemysław Kobylański

Wprowadzenie

Interpreter

- ▶ przykład programu w języku BASIC na komputerze Apple II:

```

1 LIST
2   FOR I = 1 TO 10
3   PRINT I
4   NEXT I
5
6 RUN
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
10
```

- ▶ instrukcja PRINT I wykonywana jest dziesięć razy dla kolejnych wartości

$$l = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

© 2024-26 Przemysław Kobylański

Wprowadzenie

Kompilator

- ▶ inną możliwością wykonania programu w języku wysokiego poziomu jest przetłumaczenie go na instrukcje maszynowe
- ▶ **kompilator** czyta tekst źródłowy i tłumaczy go na język maszynowy
- ▶ **konsolidator** dołącza do przekładu programu przekłady wywoływanych funkcji bibliotecznych

```
$ clang -c main.c          ; kompilacja
$ clang -o main main.o -lm ; konsolidacja
```
- ▶ main.c plik źródłowy, main.o przekład pliku źródłowego, m biblioteka funkcji matematycznych, main wykonywalny program wynikowy
- ▶ uruchomienie programu:

```
$ ./main
```

Wprowadzenie

Proste typy danych

- ▶ liczba całkowita bez znaku
 - ▶ ośmiobitowy bajt

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$$\sum_{i=0}^7 2^i \cdot b_i \in \{0, \dots, 255\}$$

- ▶ liczba trzydziestodwubitowa

$$\sum_{i=0}^{31} 2^i \cdot b_i \in \{0, \dots, 4\,294\,967\,295\}$$

Wprowadzenie

Proste typy danych

- ▶ liczba całkowita ze znakiem
 - ▶ ośmiobitowy bajt

znak	2^6	2^5	2^4	2^3	2^2	2^1	2^0
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$$\sum_{i=0}^6 2^i \cdot b_i - 2^7 \cdot b_7 \in \{-128, \dots, 127\}$$

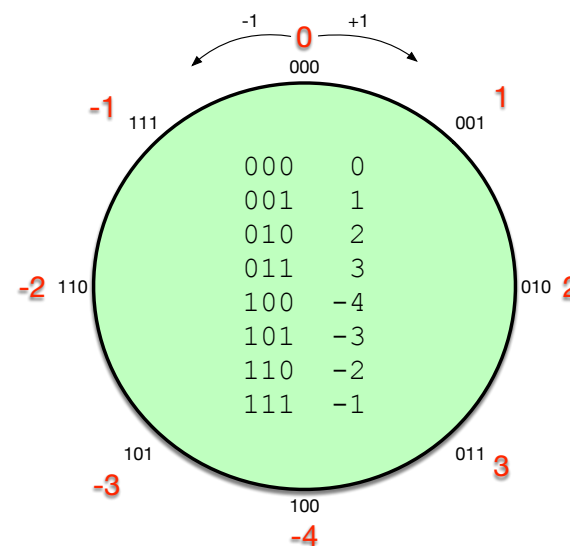
- ▶ liczba trzydziestodwubitowa

$$\sum_{i=0}^{30} 2^i \cdot b_i - 2^{31} \cdot b_{31} \in \{-2\,147\,483\,648, \dots, 2\,147\,483\,647\}$$

- ▶ jest więcej liczb ujemnych niż dodatnich

Wprowadzenie

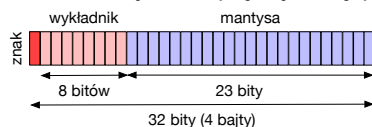
Proste typy danych



Wprowadzenie

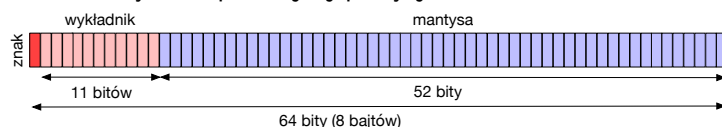
Proste typy danych

- ▶ liczba rzeczywista pojedynczej precyzji



$$(-3.40 \cdot 10^{38}, -1.18 \cdot 10^{-38}) \cup \{0\} \cup (1.18 \cdot 10^{-38}, 3.40 \cdot 10^{38})$$

- ▶ liczba rzeczywista podwójnej precyzji



$$(-1.80 \cdot 10^{308}, -2.23 \cdot 10^{-308}) \cup \{0\} \cup (2.23 \cdot 10^{-308}, 1.80 \cdot 10^{308})$$

© 2024-26 Przemysław Kobylański

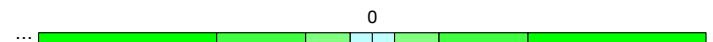
Wprowadzenie

Proste typy danych

- ▶ im dalej od zera tym coraz większe dziury w zbiorze liczb rzeczywistych

$$2^{53} + 1 = 2^{53}$$

- ▶ oddalając się od zera podwaja się szerokość przedziału o tej samej wartości wykładnika



- ▶ we wszystkich przedziałach liczb o tym samym wykładniku jest dokładnie tyle samo równomiernie rozłożonych liczb rzeczywistych:

$$2^{23} = 8\,388\,608 \text{ pojedynczej precyzji i}$$

$$2^{52} = 4\,503\,599\,627\,370\,496 \text{ podwójnej precyzji}$$

© 2024-26 Przemysław Kobylański

Wprowadzenie

Ciekawostka



Q: Który język programowania jest najlepszy?

A: Z teoretycznego punktu widzenia, pytanie pozbawione sensu.

- ▶ komputer potrafi policzyć tyle co **maszyna Turinga**²
- ▶ mówimy, że język jest **Turing-complete** jeśli można w nim wyrazić to, co potrafi policzyć maszyna Turinga
- ▶ język BASIC ograniczony do trzech instrukcji:
LET X = 0, LET X = X + 1, IF X = Y THEN GOTO
gdzie X i Y są dowolnymi zmiennymi o wartościach całkowitych (bez ograniczenia na zakres), jest Turing-complete
- ▶ instrukcja mov procesora Intel z wieloma trybami adresacji jest Turing complete
- ▶ zilustrujemy to programem w języku BASIC obliczającym silnię

²Kurs wybieralny: Teoretyczne podstawy informatyki

© 2024-26 Przemysław Kobylański

Wprowadzenie

Ciekawostka

```
1000 INPUT N
1010 LET U=0
1020 LET U=U+1
1030 LET R=0
1040 IF R=N THEN GOTO 1230
1050 LET R=R+1
1060 IF R=N THEN GOTO 1230
1070 LET U=0
1080 IF U=U THEN GOTO 1110
1090 LET U=U+1
1100 IF U=U THEN GOTO 1080
1110 LET S=0
1120 LET S=S+1
1130 LET T=0
1140 IF T=U THEN GOTO 1180
1150 LET U=U+1
1160 LET T=T+1
1170 IF T=T THEN GOTO 1140
1180 IF S=R THEN GOTO 1210
1190 LET S=S+1
1200 IF S=S THEN GOTO 1130
1210 IF R=N THEN GOTO 1230
1220 IF R=R THEN GOTO 1050
1230 PRINT U
READY.
RUN
? 10
3628800
READY.
```

© 2024-26 Przemysław Kobylański

WYKŁAD 2.

Środowisko pracy

- ▶ Język programowania C
 - ▶ kompilator **clang** albo **gcc** (zalecany clang)
 - ▶ polecenie **make**
 - ▶ analiza statyczna kodu clang **-analyze**
 - ▶ analiza stylu programowania **cpplint**

Środowisko pracy

Języki programowania (C)

```
$ nano main.c
```

```
// main.c
#include <stdio.h>
#include <math.h>
int main(void)
{
    double x;
    scanf("%lf", &x);
    printf("%f\n", sqrt(x));
    return 0;
}
```

wyjście z edytora: Ctrl-X

```
$ clang -c main.c
$ clang -o main main.o -lm
$ ./main
2
1.414214
```

Środowisko pracy

Języki programowania (C)

```
$ cpplint main.c
main.c:0: No copyright message found. You should have a line: "Copyright [year] [name]"
main.c:5: { should almost always be at the end of the previous line [whitespace]
main.c:10: Could not find a newline character at the end of the file. [whitespace]
Done processing main.c
Total errors found: 3
```

```
// Copyright 2024 Przemysław Kobylański
#include <stdio.h>
#include <math.h>
int main(void) {
    double x;
    scanf("%lf", &x);
    printf("%f\n", sqrt(x));
    return 0;
}
```

```
$ cpplint main.c
Done processing main.c
```

Środowisko pracy

Języki programowania (Python)

- ▶ Język programowania Python
 - ▶ interpreter **python3**

Środowisko pracy

Języki programowania (Python)

```
$ nano pierwiastek.py
```

```
import math
x = float(input())
print(math.sqrt(x))
```

```
$ python3 pierwiastek.py
2
1.4142135623730951
```

Środowisko pracy

Języki programowania (Ada)

- ▶ Język programowania Ada
 - ▶ zarządca pakietów **Alire**
 - ▶ utworzenie pakietu `alr init`
 - ▶ kompilacja `alr build`
 - ▶ uruchomienie `alr run`

Środowisko pracy

Języki programowania (Ada)

```
$ alr init --bin pierwiastek
$ cd pierwiastek
$ nano src/pierwiastek.adb
```

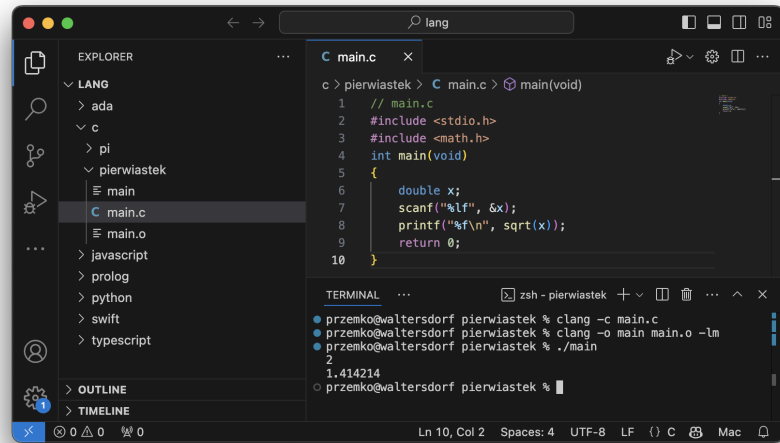
```
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;
procedure Pierwiastek is
    X : Float;
begin
    Get (X);
    Put (Sqrt (X));
end Pierwiastek;
```

```
$ alr build
$ alr run
2
1.41421E+00
```

Środowisko pracy

Zintegrowane środowisko programowania

Visual Studio Code (dostępne na Windows, Linux i osX)



© 2024-26 Przemysław Kobylański

Środowisko pracy

Zintegrowane środowisko programowania

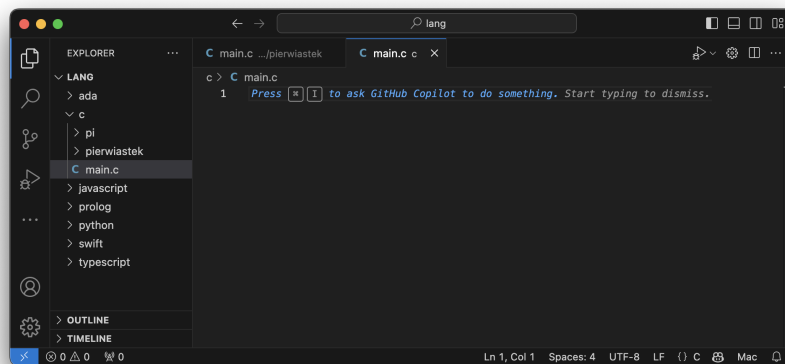
- ▶ zainstalować następujące wtyczki do obsługi języków programowania:
 - ▶ C/C++ (Microsoft)
 - ▶ Python (Microsoft)
 - ▶ Ada & SPARK (AdaCore)
- ▶ zainstalować wtyczkę wsparcia AI:
 - ▶ GitHub Copilot (GitHub)

© 2024-26 Przemysław Kobylański

Środowisko pracy

Korzystanie z Copilota

- ▶ użycie promptu:

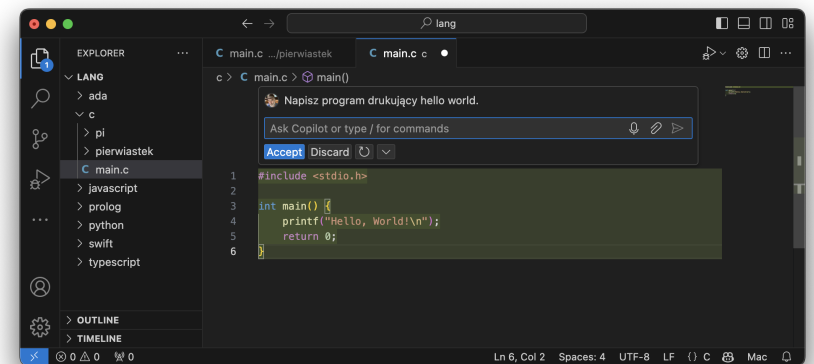


© 2024-26 Przemysław Kobylański

Środowisko pracy

Korzystanie z Copilota

- ▶ użycie promptu cd.:

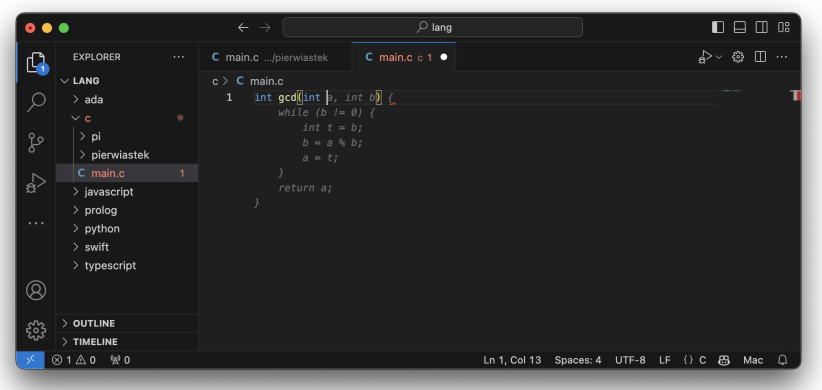


© 2024-26 Przemysław Kobylański

Środowisko pracy

Korzystanie z Copilota

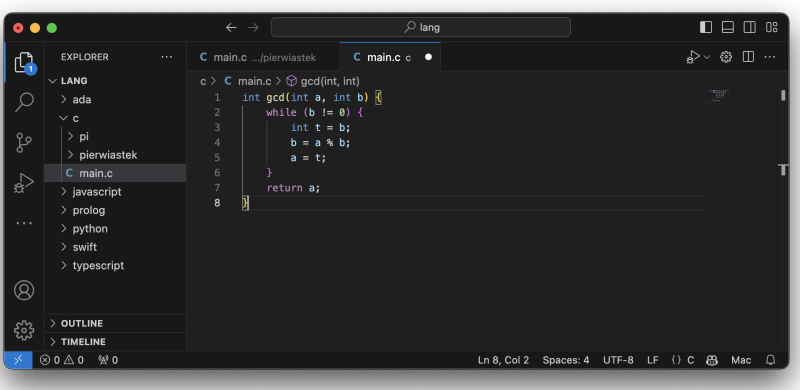
► na podstawie nagłówka funkcji:



Środowisko pracy

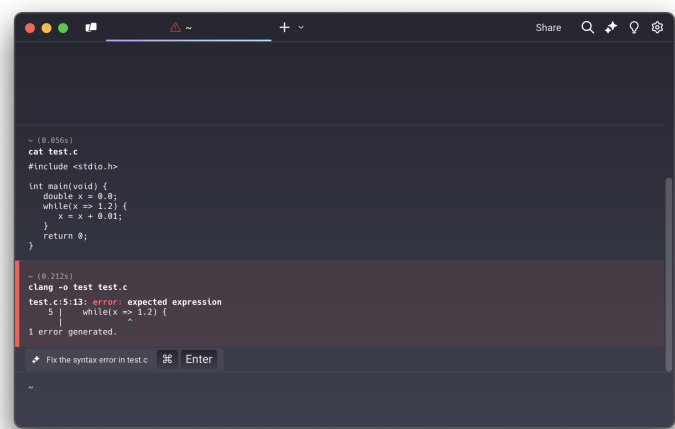
Korzystanie z Copilota

► na podstawie nagłówka funkcji cd.:



Środowisko pracy

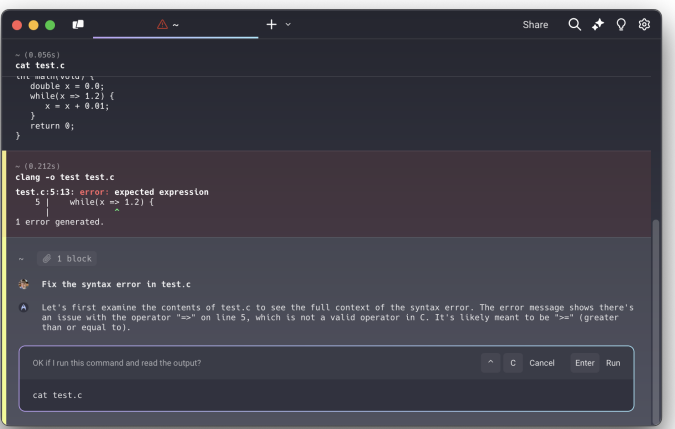
Inteligentny terminal



Praca na terminalu WARP

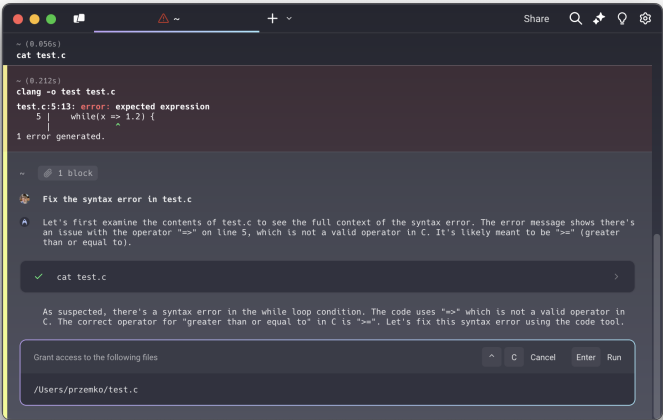
Środowisko pracy

Inteligentny terminal



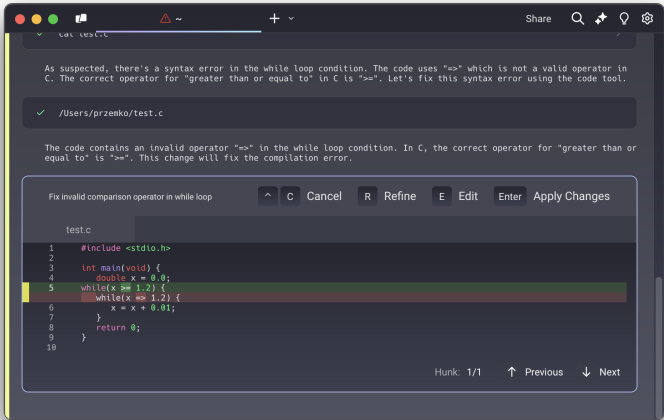
Po uruchomieniu Inteligentnego Asystenta

Środowisko pracy
Inteligentny terminal



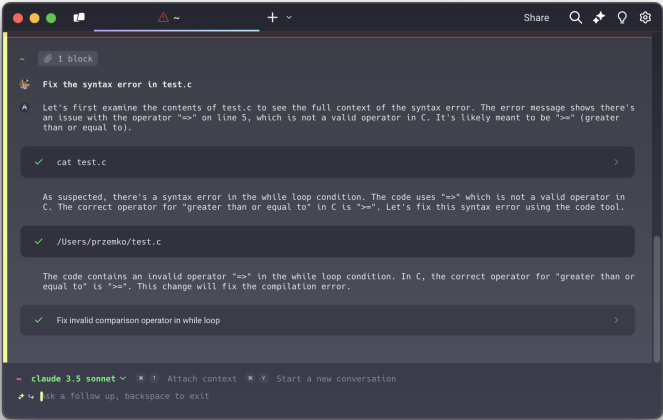
Wynik analizy pliku test.c

Środowisko pracy
Inteligentny terminal



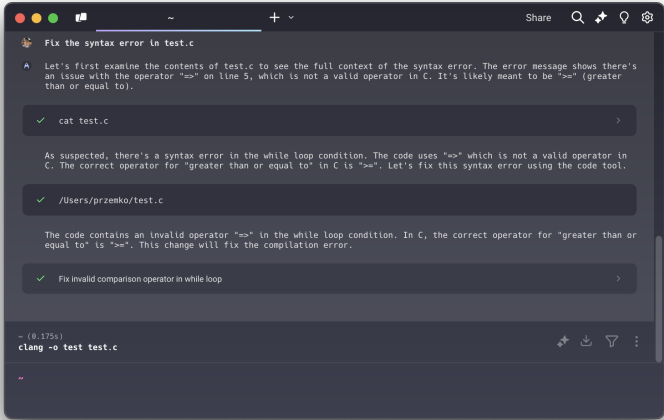
Propozycje zmian w pliku test.c

Środowisko pracy
Inteligentny terminal



Koniec pracy z Inteligentnym Asystentem

Środowisko pracy
Inteligentny terminal



Ponowna kompilacja pliku test.c



- ▶ opis instalacji narzędzi:
<https://cs.pwr.edu.pl/zawada/sp/>
- ▶ pokazać pracę w środowisku **Visual Studio Code** na kilku przykładach

WYKŁAD 3.

Instrukcje proste

Instrukcje proste

Typy danych

Algorytmy + Struktury danych = Programy



Niklaus Wirth (1934-2024)

- ▶ propagator programowania strukturalnego
- ▶ autor między innymi języków Pascal i Modula

Instrukcje proste

Typy danych

- ▶ program to algorytm operujący na strukturach danych
- ▶ program przetwarza dane zapisane za pomocą różnych typów danych
- ▶ typy danych dzieli się na proste i złożone
- ▶ złożone typy danych można rozłożyć na składowe będące prostymi typami danych
- ▶ proste typy danych są w pewien sposób atomowe tj. nierozkładalne
- ▶ na początek poznamy proste typy danych

Instrukcje proste

Typy danych

- ▶ typ danych opisuje zbiór dopuszczalnych w nim wartości oraz dostępne operacje i relacje na tych wartościach
- ▶ każdy język programowania posiada zestaw prostych typów danych
- ▶ w niektórych językach nie jest konieczne deklarowanie typu danych jakie przechowuje zmienna (np. Python)
- ▶ niektóre języki dopuszczają niezgodność typów danych na których wykonywana jest operacja (np. język C)
- ▶ w przypadku niezgodności typu danych dokonywana jest niejawna konwersja danych do tego samego typu
- ▶ są języki, w których kontrola typów jest silna i nie dopuszcza żadnych niejawnych konwersji (np. język Ada)

Instrukcje proste

Niejawna konwersja

▶ język C

```
double x = 0;           // 0 ==> 0.0, x == 0.0
int n    = 2.0;         // 2.0 ==> 2, n == 2
double y = 1.0 / n;     // 2 ==> 2.0, y == 0.5
double z = 1 / n;       // 0 ==> 0.0, z == 0.0
```

▶ język Ada

```
X : Float := 0;          -- błąd
N : Integer := 2.0;      -- błąd
Y : Float := 1.0 / N;    -- błąd
Z : Float := 1 / N;      -- błąd
```

Instrukcje proste

Jawna konwersja

▶ język C (rzutowanie)

```
double x = (double) 0;    // x == 0.0
int n    = (int) 2.0;     // n == 2
double y = 1.0 / (double) n; // y == 0.5
double z = (double) (1 / n); // z == 0.0
```

▶ język Ada

```
X : Float := Float (0);    -- X = 0.0
N : Integer := Integer (2.0); -- N = 2
Y : Float := 1.0 / Float (N); -- Y = 0.5
Z : Float := Float (1 / N); -- Z = 0
```

Instrukcje proste

Typ logiczny (C)

▶ język C (od C99)

```
#include <stdbool.h>
bool zmienna1 = true; // prawda
bool zmienna2 = false; // fałsz
bool koniunkcja = zmienna1 && zmienna2;
bool alternatywa = zmienna1 || zmienna2;
bool negacja = ! zmienna1;
```

p	q	p && q	p q	! p	! q	p == q	p != q
false	false	false	false	true	true	true	false
false	true	false	true	true	false	false	true
true	false	false	true	false	true	false	true
true	true	true	true	false	false	true	false

Instrukcje proste

Typ logiczny (Ada)

► język Ada

```
Zmienna1      : Boolean := True;  -- prawda
Zmienna2      : Boolean := False; -- fałsz
Koniunkcja    : Boolean := Zmienna1 and Zmienna2;
Alternatywa   : Boolean := Zmienna1 or  Zmienna2;
Alt_Wyklucz   : Boolean := Zmienna1 xor Zmienna2;
Negacja       : Boolean := not  Zmienna1;
```

P	Q	P xor Q	P = Q	P /= Q
False	False	False	True	False
False	True	True	False	True
True	False	True	False	True
True	True	False	True	False

Instrukcje proste

Typ logiczny (Python)

► język Python

```
zmienna1      = True  # prawda
zmienna2      = False # fałsz
koniunkcja    = zmienna1 and zmienna2
alternatywa   = zmienna1 or  zmienna2
negacja       = not  zmienna1
```

Instrukcje proste

Typ logiczny (Ada)

► szczególnymi wyrażeniami logicznymi są kwantyfikatory:

- kwantyfikator ogólny
|| (for all Zmienna in Zakres => Warunek)
- kwantyfikator szczegółowy
|| (for some Zmienna in Zakres => Warunek)

- powyższe wyrażenia muszą być ujęte w nawiasy
- przykład:

```
Is_Prime : Boolean;
...
Is_Prime := (for all I in 2 .. N - 1 => N rem I /= 0);
```

- operator **rem** wylicza resztę z dzielenia

Instrukcje proste

Typ znakowy

- typ znakowy służy do przechowywania pojedynczego znaku
- w języku Ada typem tym jest Character
- język C nie ma prawdziwego typu do przechowywania znaków
- w języku C znak przechowuje się w postaci jednobajtowej liczby będącej kodem danego znaku
- w języku C typ char przechowuje liczbę całkowitą ze znakiem³ z zakresu od −128 do 127
- w języku C literał wyrażający znak jest ujęty w apostrofy a jego wartością jest kod tego znaku

³Standard C dopuszcza, że typ char to liczby bez znaku z zakresu od 0 do 255.

Instrukcje proste

Typ znakowy (C)

- ▶ standard C nic nie mówi o wartościach kodów a jedynie podaje pewne warunki jakie muszą one spełniać:
 - ▶ `'0' + 1 == '1'`, `'1' + 1 == '2'`, `'2' + 1 == '3'`,
`'3' + 1 == '4'`, `'4' + 1 == '5'`, `'5' + 1 == '6'`,
`'6' + 1 == '7'`, `'7' + 1 == '8'`, `'8' + 1 == '9'`
 - ▶ `'A' < 'B'`, `'B' < 'C'`, ..., `'Y' < 'Z'`
 - ▶ `'a' < 'b'`, `'b' < 'c'`, ..., `'y' < 'z'`
- ▶ pierwszy warunek umożliwia wygodne obliczanie wartości liczby zapisanej jako ciąg znaków (jeśli `ch` przechowuje znak będący cyfrą, to `ch - '0'` jest wartością tej cyfry)
- ▶ dwa kolejne warunki gwarantują poprawne porządkowanie leksykograficzne ciągów znaków

Instrukcje proste

Typ znakowy (Ada)

- ▶ w języku Ada typ `Character` nie jest typem liczbowym (jego wartości nie są kodami znaków ale znakami)
- ▶ typ znakowy jest typem wyliczeniowym, w którym wymieniono po kolei wszystkie możliwe znaki
- ▶ aby uzyskać znak o danym kodzie należy skorzystać z wyrażenia `Character'Val` (kod)
- ▶ aby uzyskać kod danego znaku należy skorzystać z wyrażenia `Character'Pos` (znak)
- ▶ kody (pozycje) znaków w typie `Character` odpowiadają kodowaniu Latin 1 (formalnie ISO/IEC 8859-1)

Instrukcje proste

Typ znakowy (Python)

- ▶ w języku Python nie ma typu znakowego
- ▶ za znak w języku Python przyjmuje się łańcuch znaków długości 1
- ▶ w językach C i Ada znak i łańcuch znaków złożony z jednego znaku to dwie różne rzeczy (`'A'` znak, `"A"` łańcuch)

Instrukcje proste

Typ całkowity (C)

- ▶ w języku C dostępne są następujące typy całkowite: `char`, `signed char`, `unsigned char`, `short`, `signed short`, `unsigned short`, `int`, `signed int`, `unsigned int`, `long int`, `signed long int`, `unsigned long int`, `long long int`, `signed long long int`, `unsigned long long int`
- ▶ standard C wymaga aby:

$$\text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long int}) \leq \text{sizeof}(\text{long long int})$$
$$\text{sizeof}(\text{char}) \geq 1, \text{sizeof}(\text{short}) \geq 2, \text{sizeof}(\text{int}) \geq 2,$$
$$\text{sizeof}(\text{long int}) \geq 4, \text{sizeof}(\text{long long int}) \geq 8,$$

gdzie `sizeof()` jest liczbą bajtów pamięci jaką zajmuje wartość danego typu

Instrukcje proste

Typ całkowity (C)

► w języku C na współczesnych komputerach najczęściej:

typ	sizeof	min	max
signed char	1	-128	127
unsigned char	1	0	255
signed short int	2	-32768	32767
unsigned short int	2	0	65535
signed int	4	-2147483648	214783647
unsigned int	4	0	4294967295
signed long int	8	-9223372036854775808	9223372036854775807
unsigned long int	8	0	18446744073709551615
signed long long int	8	-9223372036854775808	9223372036854775807
unsigned long long int	8	0	18446744073709551615

Instrukcje proste

Typ całkowity (C)

► operatory dwuargumentowe:

operator	operacja	przykład
+	suma	
-	różnica	
*	iloczyn	
/	iloraz całkowity	7 / 2 == 3
%	reszta z dzielenia	7 % 3 == 1
	bitowa alternatywa	2 1 == 3 (10 01 == 11)
&	bitowa koniunkcja	5 & 3 == 1 (101 & 011 == 001)
^	bitowa alternatywa wykluczająca	12 ^ 9 == 5 (1100 ^ 1001 == 0101)

► operatory jednoargumentowe:

operator	operacja
+a	wartość a
-a	wartość przeciwna
~a	bitowa negacja
a++	postinkrementacja
a--	postdekrementacja
++a	preinkrementacja
--a	predekrementacja

Instrukcje proste

Ciekawostka z języka C



O wartości bezwzględnej wiemy, że dla dowolnej liczby x jest ona nieujemna $|x| \geq 0$.

W standardzie języka C dostępna jest następująca funkcja:

```
int abs(int x) {
    return (x >= 0) ? x : -x;
}
```

co jest równoważne definicji:

```
int abs(int x) {
    if(x >= 0) return x; else return -x;
}
```

Obliczmy wartość $abs(INT_MIN)$:

$abs(INT_MIN) = -INT_MIN = INT_MAX+1 = INT_MIN < 0$

Instrukcje proste

Typ całkowity (Ada)

► w języku Ada dostępne są dwa typy całkowite:

- typ całkowity ze znakiem **range** Low .. High
- typ modularny **mod** Modulus

► predefiniowane są następujące typy:

- Integer o zakresie wartości od Integer'First $\leq -2^{15} + 1$ do Integer'Last $\geq 2^{15} - 1$
- subtype Natural is Integer range 0..Integer'Last
- subtype Positive is Integer range 1..Integer'Last
- Long_Integer o zakresie wartości od Long_Integer'First $\leq -2^{31} + 1$ do Long_Integer'Last $\geq 2^{31} - 1$

- jeśli podczas obliczeń wartość przekroczy zakres typu całkowitego, to zgłaszany jest wyjątek Constraint_Error
- obliczenia w typie modularnym nigdy nie przekroczą zakresu wartości 0..Modulus-1 ponieważ wykonują się one w arytmetyce modulo

Instrukcje proste

Typ całkowity (Ada)

▶ operatory dwuargumentowe dla typu całkowitego:

operator	operacja	przykład
+	suma	
-	różnica	
*	iloczyn	
/	iloraz całkowity	7 / 2 = 3
rem	reszta z dzielenia	(-7) rem 3 = -1
mod	operacja modulo	(-7) mod 3 = 2
**	potęgowanie	2 ** 3 = 8

▶ operatory jednoargumentowe dla typu całkowitego:

operator	operacja
+a	wartość a
-a	wartość przeciwna

Instrukcje proste

Typ modularny (Ada)

▶ operatory dwuargumentowe dla typu modularnego:

operator	operacja
+	suma
-	różnica
*	iloczyn
/	iloraz całkowity
rem	reszta z dzielenia
mod	operacja modulo
**	potęgowanie (wykładnik całkowity)
and	bitowa koniunkcja
or	bitowa alternatywa
xor	bitowa alternatywa wykluczająca

▶ operatory jednoargumentowe dla typu modularnego:

operator	operacja
+a	wartość a
-a	wartość przeciwna (w arytmetyce modulo)
not a	największa wartość w typie - a

Instrukcje proste

Typ całkowity (dowolnej długości)

▶ w języku Python liczby całkowite są dowolnie dużej długości

```
x = 1
for x in range(1, 101):
    x = x * i
print(x)
```

93326215443944152681699238856266700490715968264381621468
59296389521759999322991560894146397615651828625369792082
7223758251185210916864000000000000000000000000000000

```
print(2**1000)
```

10715086071862673209484250490600018105614048117055336074
43750388370351051124936122493198378815695858127594672917
55314682518714528569231404359845775746985748039345677748
24230985421074605062371141877954182153046474983581941267
39876755916554394607706291457119647768654216766042983165
2624386837205668069376

Instrukcje proste

Typ całkowity (dowolnej długości)

▶ w języku Ada liczby całkowite dowolnej długości dostępne są w pakiecie Big_Integers

```
with Ada.Numerics.Big_Numbers.Big_Integers;
use Ada.Numerics.Big_Numbers.Big_Integers;
with Ada.Text_IO; use Ada.Text_IO;

procedure Big is
begin
  Put_Line (Big_Integer'Image (2 ** 1000));
end Big;
```

▶ w języku C można skorzystać z biblioteki GMP (The GNU Multiple Precision Arithmetic Library) jednak korzystanie z niej jest mniej wygodne niż w języku Python albo Ada

Instrukcje proste

Typ rzeczywisty (C)

- ▶ w języku C dostępne są następujące typy rzeczywiste:
 - `float` pojedynczej precyzji (4 bajty = 32 bity)
 - `double` podwójnej precyzji (8 bajtów = 64 bity)
 - `long double` rozszerzonej precyzji (10 bajtów = 80 bitów, 12 bajtów = 96 bitów albo 16 bajtów = 128 bitów)
- ▶ rzeczywisty literał liczbowy zapisuje się w postaci normalnej (np. 23.17) albo wykładniczej (np. 2.317e1) i jest wartością podwójnej precyzji
- ▶ aby rzeczywisty literał liczbowy był liczbą pojedynczej precyzji należy dopisać na jego końcu sufix F (np. 23.17F)

```
float x;  
...  
float y = 1.0 + x; // dwie niejawne konwersje  
                  // (float) (1.0 + (double) x)
```

© 2024-26 Przemysław Kobylański

Instrukcje proste

Typ rzeczywisty (C)

```
// main.c  
int main(void) {  
    float x;  
    float y = 1.0 + x;  
}
```

```
$ clang -S main.c
```

```
$ more main.s
```

```
.section __TEXT,__text,regular,pure_instructions  
.build_version macos, 15, 0 sdk_version 15, 0  
.globl _main  
.p2align 2  
_main:                                ; @main  
; .cfi_startproc  
; %bb.0:  
    sub     sp, sp, #16  
    .cfi_def_cfa_offset 16  
    ldr     s0, [sp, #12]  
    fcvtd   d1, s0                    konwersja single na double  
    fmov     d0, #1.000000000  
    fadd     d0, d0, d1  
    fcvts   s0, d0                    konwersja double na single  
    str     s0, [sp, #8]  
    mov     w0, #0                    ; =0x0  
    add     sp, sp, #16  
    ret  
    .cfi_endproc  
; -- End function  
.subsections_via_symbols
```

© 2024-26 Przemysław Kobylański

Instrukcje proste

Typ rzeczywisty (C)

```
// main.c  
int main(void) {  
    float x;  
    float y = 1.0F + x;  
}
```

```
$ clang -S main.c
```

```
$ more main.s
```

```
.section __TEXT,__text,regular,pure_instructions  
.build_version macos, 15, 0 sdk_version 15, 0  
.globl _main  
.p2align 2  
_main:                                ; @main  
; .cfi_startproc  
; %bb.0:  
    sub     sp, sp, #16  
    .cfi_def_cfa_offset 16  
    ldr     s1, [sp, #12]  
    fmov     s0, #1.000000000  
    fadd     s0, s0, s1  
    str     s0, [sp, #8]  
    mov     w0, #0                    ; =0x0  
    add     sp, sp, #16  
    ret  
    .cfi_endproc  
; -- End function  
.subsections_via_symbols
```

© 2024-26 Przemysław Kobylański

Instrukcje proste

Typ rzeczywisty (Ada)

- ▶ w języku Ada istnieje predefiniowany typ zmiennopozycyjny `Float`
- ▶ możemy definiować własne typy zmiennopozycyjne w zależności od precyzji i zakresu naszych obliczeń

```
type My_Float is digits D range Lo .. Hi;
```
- ▶ typ `My_Float` gwarantuje precyzję co najmniej D dokładnych cyfr w zakresie do Lo do Hi
- ▶ możemy również zdefiniować typ stałopozycyjny:

```
type My_Fixed is delta D range Lo .. Hi;
```
- ▶ typ `My_Fixed` gwarantuje jednakowy odstęp między wartościami co najwyżej wynoszący D w całym zakresie od Lo do Hi

© 2024-26 Przemysław Kobylański

Instrukcje proste

Ciekawostka



- ▶ do czego może się przydać stałopozycyjna reprezentacja liczb?
- ▶ 25 lutego 1991 roku system Patriot, broniący amerykańskiej bazy wojskowej, nie przechwycił nadlatującej rakiety Scud. Zginęło 28 Amerykanów
- ▶ komputer sterujący systemem Patriot był oparty na projekcie z lat 70'tych i operował na 24-bitowych liczbach zmiennopozycyjnych
- ▶ licznik czasu zwiększał się o 0.1 co jedną dziesiątą sekundy
- ▶ za każdym razem błąd wynikający z niedokładności reprezentacji wartości 0.1 wynosił $2^{-20} \approx 0.000000953674316$
- ▶ z każdym dodaniem kolejnej wartości błąd kumulował się
- ▶ producent zalecał resetowanie zegara (wyłączanie systemu) co 20 godzin

Instrukcje proste

Ciekawostka

- ▶ system broniący bazę nie był wyłączany przez 100 godzin

godziny	sekundy	obliczony czas (s)	niedokładność (s)	przesunięcie (m)
0	0	0.0000	0.0000	0
1	3600	3599.9966	0.0034	7
8	28800	28799.9725	0.0275	55
20	72000	71999.9313	0.0687	137
48	172800	172799.8352	0.1648	330
72	259200	259199.7528	0.2472	494
100	360000	359999.6567	0.3433	687

- ▶ skumulowany błąd obliczeń czasu wyniósł 0.3433 sekundy
- ▶ w takim czasie raketa Scud pokonuje 687 metrów z prędkością 2km/s
- ▶ raketa Patriot wybuchła ponad 600 metrów od rakiety Scud
- ▶ komisja badająca to zdarzenie zaleciła zastąpienie w systemie Patriot arytmetyki zmiennopozycyjnej stałopozycyjną

Instrukcje proste

Typ rzeczywisty (Python)

- ▶ w języku Python liczby rzeczywiste najczęściej reprezentuje się jako liczby zmiennopozycyjne podwójnej precyzji (typ float 64 bitowy)

```
Python 3.9.6 (default, Aug 9 2024, 14:24:13)
[Clang 16.0.0 (clang-1600.0.26.3)] on darwin
Type "help", "copyright", "credits" or "license" for more :
>>> 0.1 + 0.2
0.30000000000000004
```

- ▶ dostępny jest również typ complex dla liczb zespolonych

```
>>> complex('1+2j')+complex('3+4j')
(4+6j)
>>> complex('j')*complex('j')
(-1+0j)
```

Instrukcje proste

Deklaracje zmiennych

- ▶ języki C i Ada wymagają deklarowania zmiennych

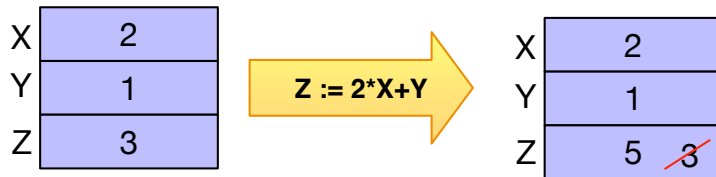
```
// język C
// Typ zmienna;
// Typ zmienna = wartość;
char ch;
char litera = 'A';
int = 13;
float x = 3.14F;
double y;
```

```
-- język Ada
-- Zmienna : Typ;
-- Zmienna : Typ := wartość;
Ch : Character := 'A';
N : Integer;
Pi : Float := 3.14;
type My_Real is digits 3 range -1.0 .. 1.0;
X : My_Real;
```

Instrukcje proste

Instrukcja podstawienia

- ▶ do zmiany wartości zmiennej służy instrukcja podstawienia
- ▶ składa się ona ze zmiennej i wyrażenia
- ▶ obliczona wartość wyrażenia jest podstawiana pod zmienną
- ▶ nowa wartość zmiennej nadpisuje starą, zatem stara wartość jest nie do odtworzenia



Instrukcje proste

Instrukcja podstawienia (C)

- ▶ w języku C „instrukcja” podstawienia ma postać
zmienna = wyrażenie;
- ▶ należy uważać, bo podczas obliczania wartości wyrażenia mogą ulec zmianie wartości zmiennych
- ▶ jeśli podczas obliczania wartości wyrażenia ulega zmianie wartość zmiennej, to mówimy, że wyrażenie ma **efekt uboczny**
- ▶ przykład efektu ubocznego w C:
`n = k++; // postinkrementacja`
Zmienna *n* przyjmuje wartość taką jaką miała zmienna *k*, natomiast po podstawieniu *n* zmienna *k* zwiększa swoją wartość o jeden.
- ▶ z powodu efektów ubocznych nie jest możliwe określenie wartości pewnych wyrażeń np.

```
int a = 10;  
printf("%d\n", ++a + --a);
```

Instrukcje proste

Instrukcja podstawienia (C)

- ▶ symbol = jest dwuargumentowym operatorem przypisania i może wystąpić wewnątrz innego podstawienia:
`a = b = c + (d = 5); // a = (b = (c + (d = 5)))`
Po wykonaniu: `d == 5`, `b == c + 5`, `a == b`.
- ▶ jeśli op jest operatorem dwuargumentowym, to w języku C możemy zapisać następujące podstawienie:
`zmienna = zmienna op wyrażenie`
w skróconej postaci:
`zmienna op= wyrażenie`

Instrukcje proste

Instrukcja podstawienia (Ada)

- ▶ w języku Ada instrukcja podstawienia ma postać
Zmienna := wyrażenie;
- ▶ w Adzie nie ma efektów ubocznych (zmienne nie mogą zmienić wartości podczas obliczania wartości wyrażenia)
- ▶ podczas obliczania wartości wyrażenia oraz podczas przypisania wartości zmiennej bardzo dokładnie kontrolowana jest zgodność typów
- ▶ mówi się, że Ada jest językiem **silnie typowanym** i nie dopuszcza żadnych niejawnych konwersji

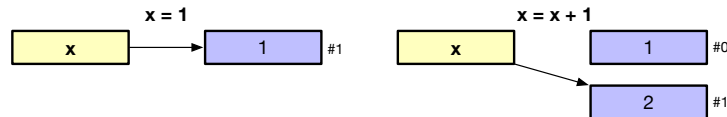
Instrukcje proste

Instrukcja podstawienia (Python)

- ▶ w języku Python nie ma zmiennych
- ▶ zamiast zmiennych są nazwy wskazujące na wartość
- ▶ proste typy, krotki i łańcuchy znaków są **niezmienne** (ang. immutable)
- ▶ w języku instrukcja podstawienia ma postać

```
|| nazwa = wyrażenie
```

- ▶ nazwa zaczyna wskazywać na wartość wyrażenia



- ▶ z każdą wartością związany jest licznik prowadzących do niej wskazań
- ▶ gdy licznik się wyzeruje wartość może być usunięta z pamięci przez **śmieciarza** (ang. garbage collector)
- ▶ możliwe są niejawne konwersje typów

© 2024-26 Przemysław Kobylański



Instrukcje proste

Instrukcja podstawienia (Python)

- ▶ symbol = nie jest operatorem jak w C ale są możliwe w Pythonie wielokrotne przypisania:

```
|| x = y = 5
```

Po wykonaniu: y == 5, x == 5

- ▶ w Pythonie możliwe są **jednoczesne** przypisania:

```
|| x, y = y, x
```

Po wykonaniu przypisania nazwy wymienią się między sobą wskazywanymi wartościami.

© 2024-26 Przemysław Kobylański

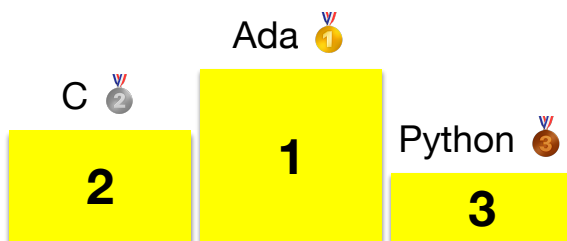


Instrukcje proste

Rodzaje typowania

- statyczne** zmienna nie może zmienić swojego typu
- dynamiczne** zmienna może zmienić swój typ
- słabe** dopuszczalna jest niejawna konwersja typu
- silne** niedopuszczalna jest niejawna konwersja typu

typowanie	słabe	silne
statyczne	C	Ada
dynamiczne	Python	—



© 2024-26 Przemysław Kobylański



Instrukcje proste

Czytanie i drukowanie wartości (C)

- ▶ aby przeczytać wartość zmiennej należy skorzystać z funkcji `scanf()`:

```
|| scanf(format, &zmienna);
```

- ▶ aby wydrukować wartość wyrażenia należy skorzystać z funkcji `printf()`:

```
|| printf(format, wyrażenie);
```

- ▶ aby skorzystać z funkcji `scanf()` i `printf()` trzeba dołączyć plik nagłówkowy `stdio.h`:

```
|| #include <stdio.h>
```

- ▶ przykład sformatowanego wydruku:

```
|| printf("po %d iteracjach suma wynosi %f\n", i, suma);
```

© 2024-26 Przemysław Kobylański



Instrukcje proste

Czytanie i drukowanie wartości (C)

► wybrane formaty:

typ	czytanie	drukowanie
char	%c	%c
int	%d	%d
unsigned int	%u	%u
float	%f	%f
double	%lf	%f
long double	%Lf	%f

► wybrane znaki specjalne:

znak	znaczenie
\n	nowa linia
\t	tabulacja
\"	cudzysłów
\\	odwrócony ukośnik

Instrukcje proste

Czytanie i drukowanie wartości (C)

```
// main.c
#include <stdio.h>
int main(void) {
    int n;
    double x;
    scanf("%d", &n);
    scanf("%lf", &x); // literka l dla double!
    printf("%f\n", x / n);
    printf("%7.3f\n", x / n);
    return 0;
}
```

```
$ ./main
2
1.5
0.750000
0.750
```

Instrukcje proste

Czytanie i drukowanie wartości (Ada)

- procedura Ada.Integer_Text_IO.Get (N) czyta wartość całkowitą i podstawia za zmienną N
- procedura Ada.Integer_Text_IO.Put (wyrażenie) drukuje wartość całkowitą wyrażenia
- aby skorzystać z procedury Get i Put należy zaimportować pakiet:
with Ada.Integer_Text_IO;
- procedura Ada.Float_Text_IO.Get (X) czyta wartość rzeczywistą i podstawia za zmienną X
- procedura Ada.Float_Text_IO.Put (wyrażenie) drukuje wartość rzeczywistą wyrażenia
- aby skorzystać z procedury Get i Put należy zaimportować pakiet:
with Ada.Float_Text_IO;

Instrukcje proste

Czytanie i drukowanie wartości (Ada)

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
procedure Main is
    N : Integer;
    X : Float;
begin
    -- przykład przeciążenia:
    Get (N); -- typ zmiennej wskazuje, którą
    Get (X); -- z dwóch procedur użyć
    Put (X / Float (N));
    Put (X / Float (N), Fore => 4, Aft => 3, Exp => 0);
end Main;
```

```
$ alr run
2
1.5
7.50000E-01 0.750
```

Instrukcje proste

Czytanie i drukowanie wartości (Python)

- ▶ do drukowania wartości wyrażenia służy procedura `print(wyrażenie)`
- ▶ za pomocą funkcji `input()` można przeczytać ciąg znaków wprowadzony na klawiaturze
- ▶ dokonując konwersji typów można przeczytać liczbę całkowitą:
`n = int (input ())`
albo rzeczywistą:
`x = float (input ())`

Instrukcje proste

Czytanie i drukowanie wartości (Python)

```
# main.py
n = int (input ())
x = float (input ())
print(x / n)
```

```
$ python3 main.py
2
1.5
0.75
```

Instrukcje proste

Stan obliczeń

- ▶ przez **stan obliczeń** obliczeń rozumiemy wartości wszystkich zmiennych w danej chwili
- ▶ w językach niedopuszczających efektów ubocznych, stan obliczeń może zmienić się jedynie w wyniku podstawienia albo przeczytania wartości zmiennej
- ▶ czasami na stan obliczeń mówi się **obraz migawkowy pamięci**
- ▶ śledzenie programu to obserwowanie jak w kolejnych krokach zmienia się jego stan obliczeń
- ▶ ciąg kolejnych zmian stanu nazywa się **śladem programu**
- ▶ niepoprawny wynik programu jest **skutkiem** błędów
- ▶ aby naprawić program należy znaleźć **przyczynę**
- ▶ dzięki **śledzeniu** działania programu można wykryć **przyczynę** niepoprawnego działania

Instrukcje proste

Śledzenie programu

- ▶ ślad programu najwygodniej zapisywać w tabelce (w kolumnach wartości zmiennych, w wierszach kolejne stany obliczeń)

```
// main.c
#include <stdio.h>
int main(void) {
    int x = 0;
    int y = 1;
    x += y++;
    x += y++;
    x += y++;
    x += y++;
    printf("%d %d\n", x, y);
}
```

x	y	komentarz
0	1	inicjalizacja
1	2	x += y++
3	3	x += y++
6	4	x += y++
10	5	x += y++

```
./main
10 5
```

WYKŁAD 4.

Instrukcje warunkowe

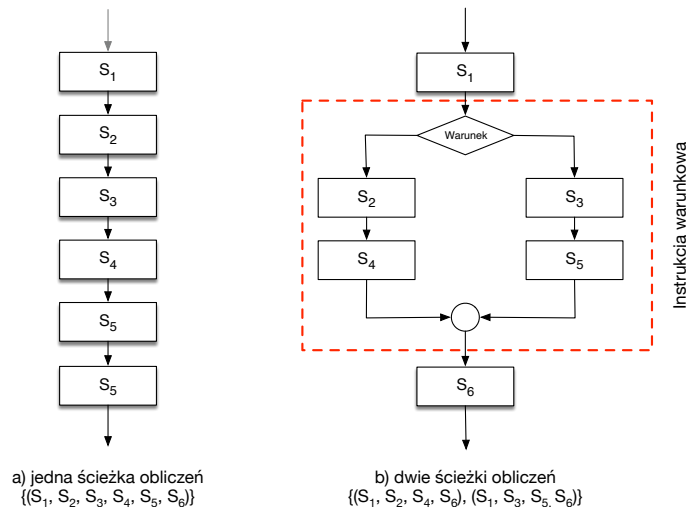
Instrukcje warunkowe

Ścieżka obliczeń

- ▶ **ścieżką obliczeń** będziemy nazywać ciąg wykonywanych podczas jego działania instrukcji
- ▶ jeśli program składa się tylko z prostych instrukcji (czytanie, podstawianie, drukowanie), to za każdym razem wykonuje dokładnie ten sam ciąg instrukcji tj. ma tylko jedną ścieżkę obliczeń
- ▶ języki programowania umożliwiają rozwidlanie ścieżek za pomocą instrukcji warunkowych
- ▶ w programowaniu strukturalnym a takim zajmujemy się na tym kursie, można sobie wyobrazić, że każda instrukcja ma jeden punkt wejścia i jeden punkt wyjścia
- ▶ ustawiając instrukcje w ciąg łączymy wyjście poprzedniej instrukcji z wejściem następnej

Instrukcje warunkowe

Ścieżka obliczeń



Instrukcje warunkowe

Ścieżka obliczeń

- ▶ program składający się z n kolejnych instrukcji warunkowych może posiadać 2^n różnych ścieżek obliczeń
- ▶ sytuacja skomplikuje się jeszcze bardziej gdy poznamy instrukcje iteracji, ponieważ program zawierający tylko jedną instrukcję iteracji może posiadać nawet nieskończenie wiele różnych ścieżek obliczeń
- ▶ testując program powinniśmy przetestować każdą możliwą ścieżkę obliczeń
- ▶ jedynym sposobem przekonania się o poprawności programu jest formalny dowód jego poprawności

Metody formalne powinny stanowić część kształcenia każdego informatyka i inżyniera oprogramowania, tak jak odpowiednia dziedzina matematyki stosowanej stanowi niezbędną część kształcenia wszystkich innych inżynierów.

– Z raportu FAA (Federal Aviation Authority) i NASA

Instrukcje warunkowe

Ścieżka obliczeń

Testowanie wskazuje na obecność nie zaś na brak błędów.

– Edsger Dijkstra

- ▶ za pomocą testów możemy jedynie przekonać się o **niepoprawności programu**
- ▶ testy nigdy nie potwierdzą **poprawności programu**
- ▶ to że program przeszedł pomyślnie wszystkie przygotowane przez nas testy nie znaczy, że jest poprawny

Instrukcje warunkowe

Instrukcja **if-else** (C)

- ▶ instrukcje **if-else** można zagnieżdżać:

```
if(w1) {  
    if(w2) { gdy w1&&w2 } else { gdy w1&&!w2 }  
}  
else {  
    if(w3) { gdy !w1&&w3 } else { gdy !w1&&!w3 }  
}
```

- ▶ jeśli ciąg instrukcji jest jednoelementowy, to można pominąć otaczające go nawiasy:

```
if(n == 1) printf("jeden\n");  
else if(n == 2) printf("dwa\n");  
    else if(n == 3) printf("trzy\n");  
        else printf("dużo\n");
```

Instrukcje warunkowe

Instrukcja **if-else** (C)

- ▶ instrukcja **if-else** zawiera warunek oraz dwa ciągi instrukcji:

```
if(warunek) {  
    // wykonuje się gdy warunek jest spełniony  
    pierwszy ciąg instrukcji  
}  
else {  
    // wykonuje się gdy warunek nie jest spełniony  
    drugi ciąg instrukcji  
}
```

- ▶ jeśli drugi ciąg instrukcji jest pusty, to możemy go pominąć wraz ze słowem kluczowym **else**:

```
if(warunek) {  
    // wykonuje się gdy warunek jest spełniony  
    ciąg instrukcji  
}
```

Instrukcje warunkowe

W potrzasku języka C



- ▶ rozpatrzmy następujący poprawny składniowo fragment programu sterującego ruchem kolejowym:

```
if(semator_podniesiony);  
    jazda_do_przodu;
```
- ▶ przez pomyłkę postawiono średnik za nawiasem zamykającym warunek
- ▶ w przypadku semafora opuszczonego pociąg pojedzie gdyż zbędny średnik oddziela instrukcję **if** od instrukcji uruchamiającej pociąg, a która wykonywana jest zawsze bez względu na spełnienie lub niespełnienie warunku
- ▶ należy tak projektować języki programowania aby takie i podobne błędy powodowały niepoprawność składniową programu (**program ma się nie skompilować a nie skompilować i działać niepoprawnie**)

Instrukcje warunkowe

Instrukcja **if-then-else** (Ada)

- instrukcja **if-then-else** wygląda w Adzie następująco:

```
if Warunek then
  -- wykonuje się gdy warunek jest spełniony
  pierwszy ciąg instrukcji
else
  -- wykonuje się gdy warunek nie jest spełniony
  drugi ciąg instrukcji
end if;
```

- jeśli drugi ciąg instrukcji jest pusty, to możemy go pominąć wraz ze słowem kluczowym **else**:

```
if Warunek then
  -- wykonuje się gdy warunek jest spełniony
  ciąg instrukcji
end if;
```

Instrukcje warunkowe

Instrukcja **if-then-else** (Ada)

- instrukcje **if-then-else** mogą być zagnieżdżone:

```
if W1 then
  if W2 then
    -- gdy W1 and W2
  else
    -- gdy W1 and not W2
  end if;
else
  if W3 then
    -- gdy not W1 and W3
  else
    -- gdy not W1 and not W3
  end if;
end if;
```

Instrukcje warunkowe

Instrukcja **if-then-else** (Ada)

- dzięki słowu kluczowemu **elsif** można sprawdzać szereg warunków bez zagnieżdżania instrukcji **if-then-else**:

```
if N = 1 then
  Put_Line ("jeden");
elsif N = 2 then
  Put_Line ("dwa");
elsif N = 3 then
  Put_Line ("trzy");
else
  Put_Line ("dużo");
end if;
```

Instrukcje warunkowe

Instrukcja **if-else** (Python)

- instrukcja **if-else** ma w języku Python postać następującą:

```
if Warunek:
  # wykonuje się gdy warunek jest spełniony
  pierwszy ciąg instrukcji
else:
  # wykonuje się gdy warunek nie jest spełniony
  drugi ciąg instrukcji
```

- ciągi instrukcji muszą być wcięte za pomocą znaku tabulacji względem kolumny, w której rozpoczyna się słowo kluczowe **if**
- słowo kluczowe **else** musi być w tej samej kolumnie co słowo kluczowe **if**
- jeśli drugi ciąg instrukcji jest pusty, to można go pominąć wraz ze słowem kluczowym **else**:

```
if Warunek:
  # wykonuje się gdy warunek jest spełniony
  ciąg instrukcji
```


Instrukcje warunkowe

Instrukcja **if-else** (Python)

- ▶ dzięki słowu kluczowemu **elif** można sprawdzać szereg warunków bez zagnieżdżania instrukcji **if-else**:

```
if n == 1:
    print("jeden")
elif n == 2:
    print("dwa")
elif n == 3:
    print("trzy")
else:
    print("dużo")
```

Instrukcje warunkowe

W potrzasku języka Python



- ▶ rozpatrzmy następujący poprawny składniowo fragment programu sterującego ruchem kolejowym:

```
if semafor_podniesiony:
    daj_sygnał_dźwiękowy
jazda_do_przodu
```
- ▶ przez pomyłkę pominięto znak tabulacji przed instrukcją `jazda_do_przodu`
- ▶ w przypadku semafora opuszczonego pociąg pojedzie i to bez sygnału dźwiękowego
- ▶ z rozpatrywanych języków programowania C, Ada i Python tylko Ada została starannie zaprojektowana i jest wolna od takich niespodzianek
- ▶ język Ada został zaprojektowany specjalnie do krytycznych zastosowań, w których od poprawności programu zależy zdrowie a nawet życie ludzkie

Instrukcje warunkowe

Instrukcja **switch-case** (C)

- ▶ jeśli należy wybrać jedną ze ścieżek obliczeń w zależności od wartości pewnego wyrażenia, to najwygodniej jest użyć instrukcji **switch-case**:

```
switch(wyrażenie) {
    case wartość1:
        // gdy wyrażenie ma wartość1
        pierwszy ciąg instrukcji
        break;
    case wartość2:
        // gdy wyrażenie ma wartość2
        drugi ciąg instrukcji
        break;
    ...
    default:
        // gdy wyrażenie ma inną wartość niż
        wymienione
        ostatni ciąg instrukcji
}
```

Instrukcje warunkowe

Instrukcja **switch-case** (C)

- ▶ przykład z liczbami:

```
switch(n) {
    case 1:
        printf("jeden\n");
        break;
    case 2:
        printf("dwa\n");
        break;
    case 3:
        printf("trzy\n");
        break;
    default:
        printf("dużo\n");
}
```

Instrukcje warunkowe

Instrukcja `switch-case` (C)

- ▶ czasami wybiera się tę samą ścieżkę w przypadku różnych wartości:

```
switch(wyrażenie) {
    case wartość1:
        // gdy wyrażenie ma wartość1
        pierwszy ciąg instrukcji
        break;
    case wartość2:
    case wartość3:
        // gdy wyrażenie ma wartość2 lub wartość3
        drugi ciąg instrukcji
        break;
    ...
    default:
        // gdy wyrażenie ma inną wartość niż
        wymienione
        ostatni ciąg instrukcji
}
```

© 2024-26 Przemysław Kobylański



Instrukcje warunkowe

Instrukcja `switch-case` (C)

- ▶ jeśli pominie się którąś z instrukcji `break`, to będzie wykonywany kolejny ciąg instrukcji:

```
switch(wyrażenie) {
    case wartość1:
        // gdy wyrażenie ma wartość1
        pierwszy ciąg instrukcji
    case wartość2:
        // gdy wyrażenie ma wartość1 lub wartość2
        drugi ciąg instrukcji
        break;
    ...
    default:
        // gdy wyrażenie ma inną wartość niż
        wymienione
        ostatni ciąg instrukcji
}
```

- ▶ gdy wyrażenie ma pierwszą wartość, to wykona się zarówno pierwszy jak i drugi ciąg instrukcji

© 2024-26 Przemysław Kobylański



Instrukcje warunkowe

W potrzasku języka C



- ▶ instrukcja `switch-case` w języku C, to tylko pewna forma skoku w ustalone miejsce programu:

```
#include <stdio.h>
int main(void) {
    switch(1) {
        for(int i=0; i < 10; i++)
            case 1: printf("%d\n", i);
    }
    return 0;
}
```

- ▶ trudno przewidzieć co wydrukuje program bo następuje skok do wnętrza pętli z pominięciem inicjowania zmiennej sterującej `i` (na moim komputerze pętla drukowała kolejne liczby począwszy od `-312999856`)

© 2024-26 Przemysław Kobylański



Instrukcje warunkowe

Instrukcja `case-when` (Ada)

- ▶ instrukcja `case-when` ma w języku Ada następującą postać:

```
case wyrażenie is
    when wartość1 =>
        -- gdy wyrażenie ma wartość1
        pierwszy ciąg instrukcji
    when wartość2 =>
        -- gdy wyrażenie ma wartość2
        drugi ciąg instrukcji
    ...
    when others =>
        -- gdy wyrażenie ma inną wartość niż
        wymienione
        ostatni ciąg instrukcji
end case;
```

- ▶ jeśli pominie się przypadek `when others` to w pozostawionych przypadkach muszą być wymienione wszystkie wartości w typie wyrażenia

© 2024-26 Przemysław Kobylański



Instrukcje warunkowe

Instrukcja **case-when** (Ada)

- ▶ przykład z liczbami:

```
case N is
  when 1      => Put_Line ("jeden");
  when 2      => Put_Line ("dwa");
  when 3      => Put_Line ("trzy");
  when others => Put_Line ("dużo");
end case;
```

Instrukcje warunkowe

Instrukcja **case-when** (Ada)

- ▶ można wykonać ten sam ciąg instrukcji przy różnych wartościach:

```
case Oczka_Na_Kostce is
  when 1..3  => Put_Line ("mało");
  when 4..6  => Put_Line ("dużo");
end case;

case Oczka_Na_Kostce is
  when 1|3|5 => Put_Line ("nieparzyste");
  when 2|4|6 => Put_Line ("parzyste");
end case;
```

- ▶ przy założeniu następujących deklaracji:

```
type Wynik_Rzutu is range 1 .. 6;
Oczka_Na_Kostce : Wynik_Rzutu;
```

Instrukcje warunkowe

Instrukcja **match-case** (Python)

- ▶ język Python posiada ogólniejszą instrukcję **match-case**, której szczególne przypadki odpowiadają instrukcjom **switch** z języka C i **case** z języka Ada:

```
match wyrażenie:
  case wartość1:
    # gdy wyrażenie ma wartość1
    pierwszy ciąg instrukcji
  case wartość2:
    # gdy wyrażenie ma wartość2
    drugi ciąg instrukcji
  ...
  case _:
    # gdy wyrażenie ma inną wartość niż
    # wymienione
    ostatni ciąg instrukcji
```

- ▶ słowo kluczowe **case** wcięte jednym znakiem tabulacji a ciąg instrukcji wcięty dwoma znakami tabulacji

Instrukcje warunkowe

Instrukcja **match-case** (Python)

- ▶ również w języku Python można wybrać ten sam ciąg instrukcji dla różnych wartości wyrażenia

```
match oczkaNaKostce:
  case 1|3|5:
    print("nieparzyste")
  case 2|4|6:
    print("parzyste")

match oczkaNaKostce:
  case n if n < 4:
    print("mało")
  case n if n > 3:
    print("dużo")
```

Instrukcje warunkowe

Ciekawostka



- ▶ w roku 1986 próbowałem skompilować na komputerze **Odra 1325** program, który kodował tekst za pomocą kropek i kresek zgodnie z alfabetem Morse'a
- ▶ program napisany w języku **Pascal** zawierał następującą instrukcję **case** z dwudziestoma sześcioma przypadkami dla każdej z liter:

```
case litera of
  'A': WriteLn ('.-');
  'B': WriteLn ('-...');
  'C': WriteLn ('-.-.');
  ...
  'Z': WriteLn ('--..');
end;
```

- ▶ kompilator przerywał kompilację programu ponieważ nie był w stanie przetłumaczyć instrukcji o zbyt długim przekładzie (nie mógł przekraczać ustalonej długości bufora)

© 2024-26 Przemysław Kobylański



WYKŁAD 5.

Iteracje

© 2024-26 Przemysław Kobylański



Iteracje

Wprowadzenie

- ▶ programy złożone jedynie z instrukcji prostych i instrukcji warunkowych mają skończoną liczbę skończonych ścieżek obliczeń
- ▶ często zachodzi potrzeba aby pewien fragment programu powtórzyć wielokrotnie dopóki spełniony jest pewien warunek albo pewną liczbę razy (zadaną lub wyliczoną)
- ▶ służą do tego instrukcje iteracji (pętli)
- ▶ **przykład:** zsumować podawany do programu ciąg liczb całkowitych zakończony liczbą 0
- ▶ długość ścieżki obliczeń takiego programu nie jest ustalona i zależy od liczby wpisywanych liczb
- ▶ **wniosek:** każdy nietrywialny program zawiera przynajmniej jedną pętlę
- ▶ **uwaga:** każdy nietrywialny program spędza prawie cały swój czas działania w pętlach

© 2024-26 Przemysław Kobylański



Iteracje

Pętla **while** (C)

- ▶ podstawową pętlą jest instrukcja **while**, która w języku C ma następującą postać:

```
while(warunek) {
    ciąg instrukcji
}
```

- ▶ w wyniku jej wykonania powtarzany jest ciąg instrukcji tak długo, dopóki spełniony jest warunek sterujący
- ▶ może się zdarzyć, że program nigdy nie opuści pętli (warunek za każdym jej przebiegiem pozostanie spełniony)
- ▶ jeśli program zakończy pętlę, to zaraz po jej opuszczeniu zachodzi negacja warunku sterującego

© 2024-26 Przemysław Kobylański



Iteracje

Pętla while (C)

- ▶ instrukcja **break** opuszcza pętlę
- ▶ instrukcja **continue** przechodzi na początek pętli

```
#include <stdio.h>
#include <math.h>
#include <stdbool.h>

int main(void) {
    double x;
    while(true) {
        printf("Podaj liczbę (0 by zakończyć): ");
        scanf("%lf", &x);
        if(x == 0)
            break;
        if(x < 0) {
            printf("Liczba powinna być dodatnia!\n");
            continue;
        }
        printf("Sqrt(%f) = %f\n", x, sqrt(x));
    }
    return 0;
}
```

© 2024-26 Przemysław Kobylański

Iteracje

Pętle for (C)

- ▶ najczęściej wykonuje się pętlę powtarzając ją n razy:

```
for(int i = 0; i < n; i++) {
    ciąg instrukcji
}
```

- drukowanie tabliczki mnożenia:

```
for(int a = 1; a <= 10; a++) {
    for(int b = 1; b <= 10; b++)
        printf("%4d", a * b); // pole 4 znaków
    printf("\n");
}
```

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

© 2024-26 Przemysław Kobylański

Iteracje

Pętle for (C)

- ▶ często mamy do czynienia z pętlą przed którą wykonuje się inicjowanie zmiennych sterujących jej pracą a potem powtarza się ciąg instrukcji dopóki zmienne sterujące spełniają pewien warunek, przy czym po każdej iteracji aktualizuje się wartość zmiennych sterujących:

```
inicjowanie;  
while(warunek) {  
    ciąg instrukcji  
    aktualizacja;  
}
```

- ▶ można to zapisać krócej w postaci pętli **for**:

```
for(inicjowanie; warunek; aktualizacja) {
    ciąg instrukcji
}
```

© 2024-26 Przemysław Kobylański

Iteracje

Pętle for (C)

- ▶ przykład pętli **for** drukującej potęgi dwójki mniejsze od miliarda:

```
for(int n = 1; n < 1000000000; n *= 2)
    printf("%d\n", n);
```

- ▶ co drukują poniższe pętle?

```
for(int a = 1; a <= 5; a++) {
    for(int b = 1; b <= 5; b++)
        printf("%2d", (a / b) * (b / a));
    printf("\n");
}
```

© 2024-26 Przemysław Kobylański

Iteracje

Pętla **do-while** (C)

- ▶ czasami zdarza się, że pętla musi wykonać się co najmniej raz i warunek nią sterujący sprawdzamy jest po każdej iteracji a nie przed
- ▶ w języku C pętlą taką jest pętla **do-while**:

```
do {  
    ciąg instrukcji  
} while(warunek);
```

- ▶ jest ona równoważna następującej pętli:

```
while(true) {  
    ciąg instrukcji  
    if(!warunek)  
        break;  
}
```

Iteracje

Pętla **do-while** (C)

- ▶ przykład kontroli poprawności wprowadzanych danych:

```
int n;  
do {  
    printf("Podaj liczbę z zakresu 1..6: ");  
    scanf("%d", &n);  
} while(n < 1 || n > 6);  
printf("Wprowadzono liczbę %d\n", n);
```

Iteracje

W potrzasku języka C



- ▶ zanim o języku C, to dygresja o najstarszym języku programowania FORTRAN (używanym do dziś)
- ▶ rozpatrzmy następujący fragment programu zawierający pętlę:

```
DO 10 K=1. 3  
...  
10 CONTINUE
```

- ▶ w zamierzeniach pętla powinna wykonać instrukcje pomiędzy instrukcją **DO** a wierszem z etykietą 10 trzykrotnie dla $K = 1, 2, 3$
- ▶ pomyłkowa zamiana przecinka na kropkę w instrukcji **DO** spowodowała zupełnie inne działanie
- ▶ kompilator języka FORTRAN przed analizą instrukcji usuwa z niej wszystkie spacje

Iteracje

W potrzasku języka C

- ▶ instrukcja **DO** przyjmuje zatem postać $DO10K=1.3$
- ▶ jest to poprawna składniowo instrukcja podstawiająca za zmienną $DO10K$ wartość 1.3
- ▶ język FORTRAN nie wymaga deklaracji zmiennych (podobnie jak Python) więc niezadeklarowana zmienna $DO10K$ była dopuszczalna
- ▶ instrukcja **CONTINUE** niczego nie wykonuje (to tzw. **pusta instrukcja**) i program po prostu przebiega przez nią
- ▶ instrukcje wewnątrz „pętli” zamiast wykonać się trzykrotnie dla $K = 1, 2, 3$ wykonają się raz dla $DO10K = 1.3$
- ▶ w języku C zamiana kropki na przecinek może podobnie zmienić sens programu

Iteracje

W potrzasku języka C

- ▶ rozpatrzmy następujący fragment programu w języku C:

```
x = 0.0;
while(x <= 1,5) {
    ...
    x = x + 0.01;
}
```

- ▶ w zamierzeniach pętla miała wykonywać się dla wartości zmiennej x od 0.0 do 1.5 z krokiem 0.01
- ▶ zamiana kropki na przecinek umieściła jako warunek sterujący w pętli wyrażenie $x \leq 1, 5$
- ▶ przecinek jest w C operatorem łączącym dwa wyrażenia w parę i narzucającym kolejność ich obliczania: najpierw lewe potem prawe wyrażenie
- ▶ wartością takiej pary wyrażeń jest wartość prawego wyrażenia
- ▶ wartość 5 (liczba całkowita różna od 0) oznacza w C logiczną prawdę, zatem pętla będzie wykonywana w nieskończoność (nie nastąpi błąd przepełnienia zmiennopozycyjnego)

© 2024-26 Przemysław Kobylański



Iteracje

Pętla **loop-end loop** (Ada)

- ▶ najprostszą pętlą w języku Ada jest pętla **loop-end loop**
- ▶ powtarza ona ciąg instrukcji w nieskończoność:

```
loop
    ciąg instrukcji
end loop;
```

- ▶ z pętli można wyjść korzystając z instrukcji **exit**
- ▶ wyjście wykonuje się gdy spełniony jest warunek W :

```
exit when W;
```

© 2024-26 Przemysław Kobylański

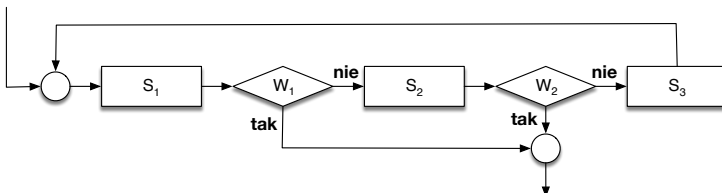


Iteracje

Pętla **loop-end loop** (Ada)

- ▶ przykład pętli z dwoma wyjściami:

```
loop
    S1
    exit when W1;
    S2
    exit when W2;
    S3
end loop;
```



© 2024-26 Przemysław Kobylański



Iteracje

Pętla **while-end loop** (Ada)

- ▶ poniższą pętlę:

```
loop
    exit when not W;
    ciąg instrukcji
end loop;
```

można zapisać w postaci pętli **while-end loop**:

```
while W loop
    ciąg instrukcji
end loop;
```

- ▶ nie ma w Adzie pętli takiej jak pętla **do-while** z języka C ale można zapisać ją w sposób równoważny:

```
loop
    ciąg instrukcji
    exit when not W;
end loop;
```

© 2024-26 Przemysław Kobylański



Iteracje

Pętla **for-end loop** (Ada)

- ▶ odpowiednikiem pętli **for** z języka C jest w Adzie poniższa pętla:

```
for Zmienna in Zakres loop
  ciąg instrukcji
end loop;
```

gdzie *Zakres* podaje się w postaci $Lo \dots Hi$

- ▶ powyższa pętla wykonuje się dla wartości Lo , $Lo+1$, $Lo+2$, ..., Hi
- ▶ zmienna sterująca pętlą jest lokalna w niej (nie jest widoczna poza pętlą)
- ▶ jeśli $Hi < Lo$, to pętla nie wykonuje się
- ▶ jeśli pętla ma się wykonać dla wartości Hi , $Hi-1$, $Hi-2$, ..., Lo , należy użyć:

```
for Zmienna in reverse Lo .. Hi loop
  ciąg instrukcji
end loop;
```

© 2024-26 Przemysław Kobylański



Iteracje

Pętla **for-end loop** (Ada)

- ▶ drukowanie tabliczki mnożenia:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Main is
begin
  for A in 1 .. 10 loop
    for B in 1 .. 10 loop
      Put (A * B, Width => 4);
    end loop;
    New_Line; -- przejście do nowej linii
  end loop;
end Main;
```

© 2024-26 Przemysław Kobylański



Iteracje

Pętla **while** (Python)

- ▶ pętla **while** ma w języku Python następującą postać:

```
while warunek:
    ciąg instrukcji
```

- ▶ ciąg instrukcji należy wciąć znakiem tabulacji
- ▶ można opuścić pętlę wykonując instrukcję **break**
- ▶ można przejść na początek pętli wykonując instrukcję **continue**

© 2024-26 Przemysław Kobylański



Iteracje

Pętla **for** (Python)

- ▶ pętla **for** ma w języku Python następującą postać:

```
for zmienna in zakres:
    ciąg instrukcji
```

- ▶ ciąg instrukcji należy wciąć znakiem tabulacji
- ▶ zakres może być następującej postaci:
 - ▶ $\text{range}(a)$ wartości $0, 1, 2, \dots, a-1$
 - ▶ $\text{range}(a, b)$ wartości $a, a+1, a+2, \dots, b-1$
 - ▶ $\text{range}(a, b, c)$ wartości $a, a+c, a+2c, \dots, a+c \lfloor \frac{b-a-1}{c} \rfloor$
- ▶ drukowanie tabliczki mnożenia:

```
for a in range(1, 11):
    for b in range(1, 11):
        print(f'{a * b:4d}', end=" ")
    print()
```

© 2024-26 Przemysław Kobylański



Iteracje

W potrzasku języka Python



- ▶ założmy, że chcemy zsumować liczby rzeczywiste mniejsze od 10.0 od 0.0 z krokiem 0.1
- ▶ rozpatrzmy następujący program:

```
x = 0.0
y = 0.0
while x < 10.0:
    y = y + x
    x = x + 0.1
print(y)
```

- ▶ w programie zapomniano nacisnąć tabulację w instrukcji `x = x + 0.1`
- ▶ zmieniło to sens programu choć pozostał on poprawny składniowo
- ▶ po uruchomieniu programu pętla **while** nie skończy pracy

© 2024-26 Przemysław Kobylański

Iteracje

Ciekawostka



- ▶ rozpatrzmy następujący program zawierający pętlę:

```
// main.c
#include <stdio.h>
int main(void) {
    int n;
    printf("Podaj liczbę n: ");
    scanf("%d", &n);
    int s = 0;
    for(int i = 1; i <= n; i++)
        s = s + i;
    printf("1 + 2 + ... + %d = %d\n", n, s);
    return 0;
}
```

© 2024-26 Przemysław Kobylański

Iteracje

Ciekawostka

- ▶ wydaje się, że do policzenia wartości $\sum_{i=1}^n i$ trzeba wykonać n dodawań
- ▶ wygenerujemy przekład na język assemblera (w przypadku mojego komputera jest to procesor M3 z rodziny ARM64)

```
$ clang -O3 -S main.c
```

- ▶ zastosowana opcja `-O3` poleca kompilatorowi zastosować najsilniejszą optymalizację kodu
- ▶ o procesorze ARM64 można poczytać na stronie <https://developer.arm.com>
- ▶ przeanalizujemy fragment uzyskanego przekładu na język assemblera
- ▶ użyte w nim operacje, to `add` – dodawanie, `lsl` – logiczne przesunięcie w prawo, `sub` – odejmowanie, `umull` – mnożenie

© 2024-26 Przemysław Kobylański

Iteracje

Ciekawostka

- ▶ w_8, w_9, w_{10} są 32-bitowymi rejestrami, x_9 jest 64-bitowym rejestrem, w_9 jest mniej znaczącą połową x_9 , początkowo $w_8 = n, w_9 = n - 1$

```
sub    w10, w8, #2
umull  x9, w9, w10
lsl    x9, x9, #1
add    w9, w9, w8, lsl #1
sub    w9, w9, #1
```

$$w_{10} = w_8 - 2 = n - 2$$

$$w_9 = w_9 \cdot w_{10} = (n - 1) \cdot (n - 2)$$

$$w_9 = \frac{w_9}{2} = \frac{(n - 1) \cdot (n - 2)}{2}$$

$$w_9 = w_9 + 2 \cdot w_8 = \frac{(n - 1) \cdot (n - 2)}{2} + 2 \cdot n$$

$$w_9 = w_9 - 1 = \frac{(n - 1) \cdot (n - 2)}{2} + 2 \cdot n - 1 = \frac{n \cdot (n + 1)}{2}$$

© 2024-26 Przemysław Kobylański

Ciekawostka

- ▶ jak widać, suma liczb od 1 do n zostaje policzona wykonując kilka operacji arytmetycznych
- ▶ jeśli kompilator optymalizując kod zauważy, że można pozbyć się pętli i wyliczyć wynik w prostszy sposób, to tak zrobi
- ▶ zoptymalizowany przekład można jeszcze trochę „ręcznie” uprościć:

```
add    w9, w9, #2
umull  x9, w9, w8
lsr    x9, x9, #1
```

$$w_9 = w_9 + 2 = n + 1$$

$$w_9 = w_9 \cdot w_8 = (n+1) \cdot n$$

$$w_9 = \frac{w_9}{2} = \frac{(n+1) \cdot n}{2}$$

© 2024-26 Przemysław Kobylański

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡

Niezmiennik i skończoność pętli

Poprawność programu

- ▶ na wykładzie tym zajmiemy się dwoma zagadnieniami:
 - ▶ czy pętla jest poprawna?
 - ▶ czy pętla zawsze kończy pracę?

```
pragma Assume (P); -- dane spełniają warunek P
S;                -- treść wykonywanego programu
pragma Assert (Q); -- wyniki spełniają warunek Q
```

Definicja (Częściowa poprawność)

Mówimy, że program S jest częściowo poprawny gdy dla danych spełniających warunek P , **jeśli** program S kończy pracę, **to** jego wyniki spełniają warunek Q .

Jeśli program nie kończy pracy, to jest częściowo poprawny.

Definicja (Pełna poprawność)

Mówimy, że program S jest w pełni poprawny gdy, **jeśli dane spełniają warunek P , to program S kończy pracę i jego wyniki spełniają warunek Q .**

Jeśli program nie kończy pracy, to nie jest w pełni poprawny.

© 2024-26 Przemysław Kobylański

st w pełni poprawny.

WYKŁAD 6.

Niezmiennik i skończoność pętli

© 2024-26 Przemysław Kobylański

◀ ◻ ▶ ◻ ◻ ▶ ◻ ≡ ▶ ◻ ≡ ▶ ≡ 🔍 ↺

Niezmiennik i skończoność pętli

Poprawność programu (SPARK podzbiór Ady)

- ▶ przykład programu spełniającego warunek końcowy:

```
1 procedure Example with SPARK_Mode is
2   X : Integer := 2;
3   Y : Integer;
4 begin
5   pragma Assume (X = 2);
6   Y := X + 2;
7   pragma Assert (Y = 4);
8 end Example;
```

```
example.adb:3:04: info: initialization of "Y" proved
```

```
example.adb:6:11: info: overflow check proved
```

```
example.adb:7:19: info: assertion proved
```

© 2024-26 Przemysław Kobylański

A set of small navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and other slide controls.

Niezmiennik i skończoność pętli

Poprawność programu (SPARK podzbiór Ady)

► przykład programu niespełniającego warunek końcowy:

```
1 | procedure Example with SPARK_Mode is
2 |   X : Integer := 2;
3 |   Y : Integer;
4 | begin
5 |   pragma Assume (X = 2);
6 |   Y := X + 2;
7 |   pragma Assert (Y = 5);
8 | end Example;
```

example.adb:3:04: info: initialization of "Y" proved
example.adb:6:11: info: overflow check proved

example.adb:7:19: medium: assertion might fail
7 | pragma Assert (Y = 5);
| ~~~~~

Niezmiennik i skończoność pętli

Ciekawostka (SPARK)



► „paradoks” z kotem Schrödingera:

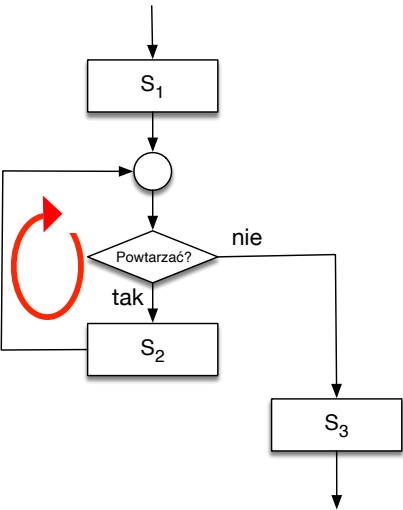
```
1 | procedure Schrodinger with SPARK_Mode, No_Return is
2 |   Cat_Is_Alive: constant Boolean := False with Ghost;
3 | begin
4 |   loop null; end loop;
5 |   pragma Assert (Cat_Is_Alive and not Cat_Is_Alive);
6 | end Schrodinger;
```

\$ alr gnatprove --report=all
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
schrodinger.adb:5:04: info: assertion proved
Summary logged in /Users/przemko/Documents/alire/schrodinger

► kot żyje i jednocześnie nie żyje!

Niezmiennik i skończoność pętli

Niezmiennik pętli



Nieskończenie wiele ścieżek obliczeń:

- (S₁, S₃)
- (S₁, S₂, S₃)
- (S₁, S₂, S₂, S₃)
- (S₁, S₂, S₂, S₂, S₃)
- (S₁, S₂, S₂, S₂, S₂, S₃)
- (S₁, S₂, S₂, S₂, S₂, S₂, S₃)
- ...

Q: Jak udowodnić poprawność nieskończenie wielu ścieżek obliczeń?
A: Indukcja!

Niezmiennik i skończoność pętli

Niezmiennik pętli

► rozpatrzmy następującą pętlę:

```
while W loop
  S
end loop;
```

gdzie *W* jest warunkiem sterującym a *S* ciągiem instrukcji

► zawsze istnieje warunek *P* taki, że:

```
pragma Assume(P and W);
S
pragma Assert(P);
```

to znaczy, jeśli spełniony jest warunek *P* oraz *W*, to po wykonaniu ciągu instrukcji *S* spełniony jest warunek *P* (takim warunkiem jest np. *True*)

Niezmiennik i skończoność pętli

Niezmiennik pętli

- ▶ można udowodnić, że taki warunek P ma również taką własność:

```
pragma Assert (P);  
while W loop  
  S  
end loop;  
pragma Assert (P and not W);
```
- ▶ warunek P o powyższej własności nazywa się niezmiennikiem pętli
- ▶ nazwa „niezmiennik” podkreśla, że warunek ten jest zachowany podczas wykonywania pętli (nie zmienia się)
- ▶ przyjmijmy, że w pętli zmieniają swoje wartości zmienne $x_1, x_2, \dots, x_n$
- ▶ będziemy to zapisywać w ten sposób:

$$P(x_1, x_2, \dots, x_n)$$

© 2024-26 Przemysław Kobylański



Niezmiennik i skończoność pętli

Niezmiennik pętli

- ▶ aby udowodnić, że $P(x_1, x_2, \dots, x_n)$ jest niezmiennikiem będziemy stosować indukcję
- ▶ w pierwszym kroku będziemy sprawdzać czy po dojściu do pętli (a przed rozpoczęciem jej wykonywania) dane spełniają warunek $P(x_1, x_2, \dots, x_n)$
- ▶ w tym celu podstawimy za zmienne $x_1, x_2, \dots, x_n$ wartości jakie mają przy dojściu do pętli i sprawdzimy czy zachodzi warunek P
- ▶ w drugim kroku będziemy udowadniać następującą implikację:

$$P(x_1, \dots, x_n) \wedge W(x_1, \dots, x_n) \wedge Q(x_1, x'_1, \dots, x_n, x'_n) \rightarrow P(x'_1, \dots, x'_n)$$

- ▶ gdzie warunek $Q(x_1, x'_1, \dots, x_n, x'_n)$ wyraża zależność nowych wartości zmiennych (te z primami) od starych wartości (te bez primów)

© 2024-26 Przemysław Kobylański



Niezmiennik i skończoność pętli

Niezmiennik pętli

- ▶ na przykład, dla poniższej pętli

```
while W loop  
  X := X + 1;  
  Y := Y + X;  
end loop;
```

warunek $Q(X, X', Y, Y')$ jest następujący:

$$X' = X + 1 \wedge Y' = Y + X'$$

- ▶ zwróć uwagę, że do starej wartości Y dodawana jest nowa wartość X' obliczona w poprzedniej instrukcji
- ▶ jeśli powiedzie się dowód indukcyjny, to udowodnimy, że po wykonaniu pętli **while** nadal zachodzi warunek P czyli jest on niezmiennikiem pętli

© 2024-26 Przemysław Kobylański



Niezmiennik i skończoność pętli

Niezmiennik pętli

- ▶ kluczową rolę odgrywa to, jaki warunek P będziemy dowodzić
- ▶ jak wcześniej wspomniano taki warunek zawsze istnieje
- ▶ warunek *True* jest bezużyteczny gdyż wszystkie wartości danych go spełniają, więc nic nie mówi on o działaniu pętli
- ▶ najlepiej byłoby za P przyjąć warunek spełniony przez jak najmniejszą liczbę danych (jak najsilniejszy)
- ▶ dlatego przez niezmiennik pętli będziemy uważać najsilniejszy warunek spośród tych, które są zachowane podczas działania pętli
- ▶ **skąd wziąć taki najsilniejszy warunek?**
- ▶ jeśli napisałeś tę pętlę, to miałeś na myśli jak ona ma działać by uzyskać pożądany efekt
- ▶ pisząc pętlę myślałeś o jej niezmienniku, nawet jeśli nie zdawałeś sobie z tego sprawy

© 2024-26 Przemysław Kobylański



Niezmiennik i skończoność pętli

Niezmiennik pętli

- ▶ jeśli to nie ty jesteś autorem pętli, to sytuacja jest bardziej skomplikowana
- ▶ najlepiej prześledzić działanie pętli analizując ślad wykonania jej pierwszych kilku/kilkunastu iteracji
- ▶ obserwując jak zmieniały się dane w kolejnych iteracjach można starać się odgadnąć postać niezmiennika
- ▶ po sformułowaniu warunku P należy jeszcze udowodnić indukcyjnie, że faktycznie jest to niezmiennik

Niezmiennik i skończoność pętli

Przykład: iloraz całkowity

- ▶ w poniższym fragmencie programu dane są wartości zmiennych $x \geq 0$ i $y > 0$:

```
int q = 0;
int r = x;
while(r >= y) {
    r = r - y;
    q = q + 1;
}
```

- ▶ udowodnimy, że powyższa pętla wylicza iloraz całkowity q i resztę z dzielenia r wartości x przez wartość y
- ▶ w tym celu pokażemy indukcyjnie, że niezmiennikiem jest $P(q, r) \equiv x = q \cdot y + r \wedge 0 \leq r$

Niezmiennik i skończoność pętli

Przykład: iloraz całkowity

- ▶ w pierwszym kroku sprawdzimy czy wartości $q = 0, r = x$ spełniają warunek niezmiennika:

$$P(0, x) \equiv x = 0 \cdot y + x \wedge 0 \leq x \equiv x = x \wedge 0 \leq x \equiv 0 \leq x$$

- ▶ w drugim kroku pokażemy prawdziwość implikacji

$$P(q, r) \wedge r \geq y \wedge Q(q, q', r, r') \rightarrow P(q', r'),$$

$$\text{gdzie } Q(q, q', r, r') \equiv r' = r - y \wedge q' = q + 1$$

Niezmiennik i skończoność pętli

Przykład: iloraz całkowity

- ▶ założenia:

$$x = q \cdot y + r, 0 \leq r, r \geq y, r' = r - y, q' = q + 1$$

- ▶ teza:

$$x = q' \cdot y + r' \wedge 0 \leq r'$$

- ▶ dowód:

$$\begin{aligned} x &= q \cdot y + r = (q' - 1) \cdot y + r = (q' - 1) \cdot y + (r' + y) \\ &= q' \cdot y + r' \\ r' &= r - y \geq 0 \end{aligned}$$

- ▶ zatem nowe wartości q' i r' spełniają warunek P
- ▶ po wyjściu z pętli zachodzi

$$P(q, r) \wedge r < y \equiv x = q \cdot y + r \wedge 0 \leq r < y$$

Niezmiennik i skończoność pętli

Przykład: potęgowanie całkowite

- ▶ w poniższym fragmencie programu dane są wartości zmiennych $x > 0$ i $n \geq 0$:

```
int y = 1, p = x, i = n;
while(i != 0) {
    if(i % 2 == 0) {
        p = p * p; i = i / 2;
    }
    else {
        y = y * p; i = i - 1;
    }
}
```

- ▶ udowodnimy, że powyższa pętla wylicza x^n
- ▶ w tym celu pokażemy indukcyjnie, że niezmiennikiem jest $P(y, p, i) \equiv y \cdot p^i = x^n$

Niezmiennik i skończoność pętli

Przykład: potęgowanie całkowite

- ▶ w pierwszym kroku sprawdzimy czy wartości $y = 1, p = x, i = n$ spełniają warunek niezmiennika:

$$P(1, x, n) \equiv 1 \cdot x^n = x^n \equiv x^n = x^n$$

- ▶ w drugim kroku pokażemy prawdziwość implikacji

$$P(y, p, i) \wedge i \neq 0 \wedge Q(y, y', p, p', i, i') \rightarrow P(y', p', i'),$$

gdzie $Q(y, y', p, p', i, i')$ opisuje jak w jednym przebiegu pętli zmieniają się wartości zmiennych y, p, i

Niezmiennik i skończoność pętli

Przykład: potęgowanie całkowite

- ▶ zmiany zmiennych zależą od parzystości wartości zmiennej i , zatem dowód będzie składał się z dwóch przypadków
- ▶ przypadek gdy i jest parzyste:
 - ▶ założenia:

$$y \cdot p^i = x^n, i \neq 0 \quad y' = y, p' = p^2, i' = \frac{i}{2}$$

- ▶ teza:

$$y' \cdot p'^{i'} = x^n$$

- ▶ dowód:

$$\begin{aligned} y' \cdot p'^{i'} &= y \cdot p'^{i'} = y \cdot (p^2)^{i'} = y \cdot (p^2)^{\frac{i}{2}} \\ &= y \cdot p^{2 \cdot \frac{i}{2}} = y \cdot p^i = x^n \end{aligned}$$

Niezmiennik i skończoność pętli

Przykład: potęgowanie całkowite

- ▶ przypadek gdy i jest nieparzyste:
 - ▶ założenia:

$$y \cdot p^i = x^n, i \neq 0 \quad y' = y \cdot p, p' = p, i' = i - 1$$

- ▶ teza:

$$y' \cdot p'^{i'} = x^n$$

- ▶ dowód:

$$\begin{aligned} y' \cdot p'^{i'} &= (y \cdot p) \cdot p'^{i'} = y \cdot p \cdot p'^{i'} = y \cdot p \cdot p^{i-1} \\ &= y \cdot p^i = x^n \end{aligned}$$

- ▶ po wyjściu z pętli zachodzi warunek $P(y, p, i) \wedge i = 0$, zatem

$$P(y, p, i) \wedge i = 0 \equiv y \cdot p^i = x^n \wedge i = 0 \rightarrow y \cdot p^0 = x^n \equiv y = x^n$$

- ▶ czy po wyjściu z pętli zmienna p może być równa 0?

Niezmienność i skończoność pętli

Skończoność pętli

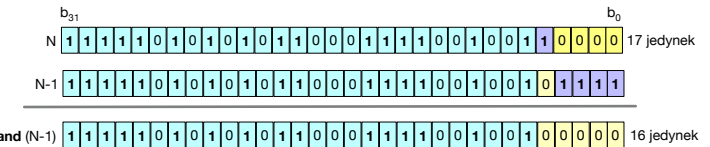
- ▶ samo udowodnienie niezmiennika nie gwarantuje, że program jest w pełni poprawny
- ▶ należy jeszcze udowodnić, że dla wszystkich poprawnych danych pętla kończy pracę
- ▶ aby pokazać, że pętla kończy pracę należy podać wyrażenie o wartościach całkowitych ograniczonych z dołu, które przy każdej iteracji pętli maleją
- ▶ porządek $<$ na liczbach naturalnych jest dobrze ufundowany gdyż nie istnieje nieskończony ciąg malejący o wartościach naturalnych
- ▶ analogicznie można dla dowodu podać wyrażenie o wartościach całkowitych ograniczonych z góry, które przy każdej iteracji pętli rosną (wówczas maleje dystans do górnego ograniczenia)

Niezmienność i skończoność pętli

Przykład w języku Ada

- ▶ czy poniższa pętla zawsze kończy pracę?

```
type Liczba is mod 2**32;  
N : Liczba;  
...  
while N /= 0 loop  
    N := N and (N - 1);  
end loop;
```



- ▶ o jeden maleje liczba jedynek w binarnej reprezentacji liczby N
- ▶ łatwo jest oszacować maksymalną liczbę iteracji
- ▶ pętla skończy się maksymalnie po 32 iteracjach dla $N = \text{Liczba}'\text{Last} = 2^{32} - 1 = 4294967295$

Niezmienność i skończoność pętli

Ciekawostka

- ▶ Hipoteza Collatza (hipoteza $3n+1$)
- ▶ w roku 1937 niemiecki matematyk **Lothar Collatz** sformułował hipotezę, że poniższy ciąg zawsze osiąga wartość 1 dla dowolnej początkowej wartości całkowitej $c_0 > 1$

$$c_{n+1} = \begin{cases} c_n/2 & \text{gdy } c_n \text{ jest parzyste,} \\ 3 \cdot c_n + 1 & \text{gdy } c_n \text{ jest nieparzyste} \end{cases}$$

- ▶ hipoteza pozostaje do dziś nierozstrzygnięta (sprawdzono liczby mniejsze od $20 \cdot 2^{58} \approx 5.764 \cdot 10^{18}$)

```
while N /= 1 loop  
    if N mod 2 = 0 then  
        N := N / 2;  
    else  
        N := 3 * N + 1;  
    end if;  
end loop;
```

Niezmienność i skończoność pętli

Przykład w języku SPARK

- ▶ zadanie zsumowania liczb od 1 do 100:

```
procedure Gauss_Summation is  
    X : Natural := 100;  
    Y : Natural := 0;  
begin  
    while X /= 0 loop  
        Y := Y + X;  
        X := X - 1;  
    end loop;  
end Gauss_Summation;
```

- ▶ czy pętla kończy pracę?
- ▶ ile wynosi wartość Y po wyjściu z pętli?

Niezmiennik i skończoność pętli

Przykład w języku SPARK

► dowód niezmiennika pętli:

1. Niech $S = \frac{100 \cdot (100+1)}{2}$.
2. Niezmiennikiem pętli jest warunek $P(X, Y) \equiv \left(Y = S - \frac{X \cdot (X+1)}{2} \right)$.
3. Dowód niezmiennika przeprowadzimy indukcyjnie.

Krok 1. Wartości początkowe zmiennych spełniają niezmiennik:

$$P(100, 0) \equiv 0 = \frac{100 \cdot (100+1)}{2} - \frac{100 \cdot (100+1)}{2} \equiv 0 = 0.$$

Krok 2. Udowodnimy prawdziwość implikacji:

$$P(X, Y) \wedge X \neq 0 \wedge Q(X, X', Y, Y') \rightarrow P(X', Y'),$$

gdzie $Q(X, X', Y, Y')$ opisuje zmianę w jednym przebiegu pętli starych wartości zmiennych X, Y na nowe X', Y' :

$$Q(X, X', Y, Y') \equiv Y' = Y + X \wedge X' = X - 1$$

© 2024-26 Przemysław Kobylański



Niezmiennik i skończoność pętli

Przykład w języku SPARK

► założenia:

$$Y = S - \frac{X \cdot (X+1)}{2}, X \neq 0, Y' = Y + X, X' = X - 1$$

► teza:

$$Y' = S - \frac{X' \cdot (X' + 1)}{2}$$

► dowód:

$$\begin{aligned} Y' &= Y + X \\ &= S - \frac{X \cdot (X+1)}{2} + X = S - \frac{(X'+1) \cdot (X'+2)}{2} + X' + 1 \\ &= S - \frac{(X'+1) \cdot (X'+2) - 2 \cdot (X'+1)}{2} = S - \frac{(X'+1) \cdot (X'+2-2)}{2} \\ &= S - \frac{(X'+1) \cdot X'}{2} \quad \square \end{aligned}$$

© 2024-26 Przemysław Kobylański



Niezmiennik i skończoność pętli

Przykład w języku SPARK

► dowód skończoności pętli:

1. Wartość zmiennej X jest ograniczona z dołu przez 0.
2. Za każdym razem wartość X maleje o jeden.
3. Zatem po skończonej liczbie iteracji osiągnię wartość $X = 0$ i wykonywanie pętli zakończy się.

► udowodniliśmy niezmiennik pętli i skończoność pętli

► z niezmiennika pętli wynika, że po wyjściu z pętli $Y = S - \frac{X \cdot (X+1)}{2}$

► po wyjściu z pętli zachodzi negacja warunku sterującego pętlą, zatem $X = 0$ i $Y = S$

© 2024-26 Przemysław Kobylański



Niezmiennik i skończoność pętli

Przykład w języku SPARK

- przeprowadzimy automatyczny dowód w języku SPARK
- dopiszemy w programie warunki niezmiennika, zmiennika i asercję
- stała S jest „duchem” na potrzeby formalnego dowodu (nie istnieje w wykonywalnym przekładzie)

```
1 procedure Gauss_Summation with SPARK_Mode is
2   X : Natural := 100;
3   S : constant Natural := X * (X + 1) / 2 with Ghost;
4   Y : Natural := 0;
5 begin
6   while X /= 0 loop
7     pragma Loop_Invariant (Y = S - X * (X + 1) / 2);
8     pragma Loop_Variant (Decreases => X);
9     Y := Y + X;
10    X := X - 1;
11  end loop;
12  pragma Assert (Y = S);
13 end Gauss_Summation;
```

© 2024-26 Przemysław Kobylański



Niezmienność i skończoność pętli

Przykład w języku SPARK

- ▶ Uruchomienie programu **GNATprove**:

```
$ alr gnatprove --report=all
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...
gauss_summation.adb:3:30: info: overflow check proved
gauss_summation.adb:3:35: info: overflow check proved
gauss_summation.adb:3:40: info: division check proved
gauss_summation.adb:3:40: info: range check proved
gauss_summation.adb:7:30: info: loop invariant initialization proved
gauss_summation.adb:7:30: info: loop invariant preservation proved
gauss_summation.adb:7:36: info: overflow check proved
gauss_summation.adb:7:40: info: overflow check proved
gauss_summation.adb:7:45: info: overflow check proved
gauss_summation.adb:7:50: info: division check proved
gauss_summation.adb:8:28: info: loop variant proved
gauss_summation.adb:9:14: info: overflow check proved
gauss_summation.adb:10:14: info: range check proved
gauss_summation.adb:12:19: info: assertion proved
Summary logged in /Users/przemko/Documents/alire/gauss_summation/obj/developmen
```

© 2024-26 Przemysław Kobylański

WYKŁAD 7.

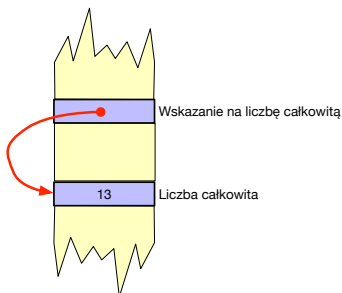
Złożone typy danych

© 2024-26 Przemysław Kobylański

Złożone typy danych

Wprowadzenie

- ▶ istnieje w językach programowania potrzeba posługiwania się odnośnikiem do wartości zamiast samą wartością
- ▶ na przykład funkcja `scanf()` w języku C czytając wartość zmiennej musi znać miejsce przechowywania tej zmiennej
- ▶ wskazanie przechowuje informację gdzie znajduje się wskazywana wartość:



© 2024-26 Przemysław Kobylański

Złożone typy danych

Wskaźniki (C)

- ▶ jeśli T jest typem danej, to T^* jest typem wskaźnika prowadzącego do danej typu T
- ▶ jeśli x jest daną typu T , to $\&x$ jest wskazaniem prowadzącym do danej x i jest typu T^*
- ▶ jeśli p jest wskazaniem typu T^* tj. wskazuje na daną typu T , to $*p$ jest daną wskazywaną przez wskaźnik p (operator $*$ nazywa się **operatorem wyłuskania** albo **operatorem dereferencji**)

```
int x = 13;
int* p = &x;
int y = *p + 10; // wartość 23
*p = 7;
int z = x + 10; // wartość 17
```

$$T^* p \equiv T *p$$

$$*(\&x) \equiv x, \&(*p) \equiv p$$

© 2024-26 Przemysław Kobylański

Złożone typy danych

Wskaźniki (C)

- ▶ czasami będziemy posługiwać się wskazaniem na wskazanie
- ▶ jeśli T jest typem danej, to T** jest typem wskazania do miejsca gdzie znajduje się wskazanie prowadzące do danej typu T
- ▶ jeśli q jest typu T**, to *q jest wskazaniem typu T* prowadzącym do miejsca gdzie znajduje się dana typu T
- ▶ jeśli q jest typu T**, to **q jest daną typu T wskazywaną przez wskazanie wskazywane przez q

```
int x = 13;
int* p = &x;
int** q = &p;
int y = **q + 10; // wartość 23
**q = 7;
int z = x + 10; // wartość 17
```

© 2024-26 Przemysław Kobylański

Złożone typy danych

Wskaźniki (C)

- ▶ niech T będzie typem, wówczas sizeof(T) jest rozmiarem danej typu T liczonym w bajtach
- ▶ funkcja malloc() służy do rezerwowania danej liczby bajtów pamięci i zwraca wskazanie typu void* na zarezerwowane miejsce (jeśli nie było już tyle wolnego miejsca, to zwracany jest pusty wskaźnik NULL)
- ▶ niech p będzie wskaźnikiem typu T*, wykonując podstawienie p = (T*)malloc(sizeof(T)) pod zmienną p zostaje podstawione wskazanie na nowo zarezerwowane miejsce rozmiaru sizeof(T) (albo NULL gdy nie było wolnego miejsca)

```
#include <stdlib.h>
...
int* p = (int*)malloc(sizeof(int));
*p = 13;
int* q = p;
*q = 7;
int x = *p + 2; // wartość 9
```

© 2024-26 Przemysław Kobylański

Złożone typy danych

Wskaźniki (C)

- ▶ w języku C możliwa jest arytmetyka na wskaźnikach
- ▶ niech p będzie typu T*, wówczas p + 1 jest wskaźnikiem wskazującym miejsce o sizeof(T) bajtów za miejscem wskazywanym przez p
- ▶ analogicznie p + n, dla całkowitego n, jest miejscem znajdującym się o n * sizeof(T) bajtów za miejscem wskazywanym przez p

```
#include <stdlib.h>
...
int* p = (int*)malloc(2 * sizeof(int));
*p = 13;
*(p + 1) = 7;
int x = *p + *(p + 1); // wartość 20
```

© 2024-26 Przemysław Kobylański

Złożone typy danych

Wskaźniki (C)

- ▶ aby zwolnić obszar pamięci wskazywany przez wskaźnik p należy wywołać funkcję free(p)
- ▶ funkcja free() nie podstawia pod p wartości NULL
- ▶ po zwolnieniu obszaru wskazywanego przez p, wskaźnik p staje się **wiszącym wskaźnikiem** (ang. dangling pointer) tj. wskaźnikiem różnym od NULL ale niewskazującym danej

```
int* p = (int*)malloc(sizeof(int));
*p = 13;
free(p);
int x = *p + 1; // błędne odwołanie do p
```

© 2024-26 Przemysław Kobylański

Złożone typy danych

Wskaźniki (Ada)

- ▶ jeśli T jest typem danej, to access T jest typem wskaźnika prowadzącego do danej typu T
- ▶ jeśli P jest typu access T, to new T tworzy nowy obiekt typu T i zwraca wskazanie na niego (albo null gdy nie było dostatecznie dużo wolnego miejsca)
- ▶ jeśli P jest wskazaniem typu access T, to P.all jest daną typu T wskazywaną przez wskaźnik P

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Main is
  type AInteger is access Integer;
  P : AInteger;
begin
  P := new Integer;
  P.all := 13;
  Put (P.all);
end Main;
```

© 2024-26 Przemysław Kobylański



Złożone typy danych

Wskaźniki (Ada)

- ▶ aby zwolnić obszar przechowujący daną typu T wskazywaną przez wskaźnik typu access T należy przygotować procedurę Delete na bazie **procedury rodzajowej** Ada.Unchecked_Deallocation
- ▶ procedura Delete nie tylko zwolni obszar ale i podstawia pod wskaźnik wartość null

```
with Ada.Unchecked_Deallocation;
procedure Main is
  type AInteger is access Integer;
  P : AInteger;
  procedure Delete is new
    Ada.Unchecked_Deallocation (Integer, AInteger);
begin
  P := new Integer;
  P.all := 13;
  Delete (P);
  pragma Assert (P = null);
end Main;
```

© 2024-26 Przemysław Kobylański



Złożone typy danych

Wskaźniki (Ada)

- ▶ dlaczego procedura rodzajowa nazywa się Ada.Unchecked_Deallocation?
- ▶ wskaźniki w Ada są bezpieczne do momentu kiedy je zwalniamy
- ▶ procedura Ada.Unchecked_Deallocation nie sprawdza czy zwalniany obszar nie jest wskazywany jeszcze przez inny wskaźnik
- ▶ zwolnienie takiego obszaru spowoduje, że ten drugi wskaźnik będzie **wiszący**, co jest niebezpieczne

© 2024-26 Przemysław Kobylański



Złożone typy danych

Wskaźniki (Ada)

- ▶ przykład wiszącego wskaźnika w Adzie:

```
with Ada.Unchecked_Deallocation;
procedure Main is
  type AInteger is access Integer;
  P, Q : AInteger;
  procedure Delete is new Ada.Unchecked_Deallocation
    (Integer, AInteger);
begin
  P := new Integer;
  P.all := 13;
  Q := P;
  Delete (P);
  Q.all := 7; -- wiszący wskaźnik!
end Main;
```

© 2024-26 Przemysław Kobylański



Złożone typy danych

Wskaźniki (Python)

- ▶ w języku Python nie ma typu wskaźnikowego
- ▶ co nie znaczy, że program operując na danych nie korzysta ze wskaźników

Złożone typy danych

Typy złożone

- ▶ typ złożony składa się ze składowych prostszych typów
- ▶ zbiór wartości typu złożonego może być:
 - ▶ zbiorem wektorów ustalonej długości złożonych z wartości tego samego typu ([tablice](#))
 - ▶ zbiorem krotek o ustalonej liczbie składowych różnych typów ([struktury](#), [rekordy](#), [krotki](#))
 - ▶ zbiorem wartości będącym sumą mnogościową różnych typów ([unie](#))

Złożone typy danych

Tablice (C)

- ▶ tablicę o n elementach tego samego typu T deklaruje się w C w następujący sposób:

```
|| T tab[n];
```

gdzie `tab` jest nazwą deklarowanej tablicy

- ▶ zostaje zarezerwowany obszar wielkości $n * \text{sizeof}(T)$ bajtów
- ▶ do wartości elementów tablicy odwołujemy się za pomocą indeksu o wartościach $0, 1, 2, \dots, n - 1$
- ▶ jeśli i jest wartością indeksu, to `tab[i]` zachowuje się dokładnie tak samo jak zmienna typu T :

```
|| tab[0] = k;  
|| tab[1] = tab[0] - 1;
```

- ▶ wartość elementu tablicy można użyć do wyliczenia indeksu:

```
|| tab[i] = tab[tab[j] + 1];
```

Złożone typy danych

Tablice (C)

- ▶ wartości elementów tablicy można zainicjować podczas deklaracji:

```
|| int tab[5] = {1, 2, 3, 4, 5};
```

lub krócej:

```
|| int tab[] = {1, 2, 3, 4, 5};
```

- ▶ zerowanie elementów tablicy w pętli:

```
|| int tab[10000];  
|| for(int i = 0; i < 10000; i++)  
||     tab[i] = 0;
```

- ▶ tablica w C może mieć liczbę elementów ustalaną podczas wykonywania programu:

```
|| int n;  
|| printf("podaj rozmiar tablicy: ");  
|| scanf("%d", &n);  
|| double tab[n];
```

Złożone typy danych

W potrzasku języka C



- ▶ w języku C nie ma prawdziwych tablic, jest tylko **arytmetyka wskaźników**
- ▶ po zarezerwowaniu obszaru pamięci na tablicę:

```
|| int tab[n];
```

identyfikator tablicy jest zwykłym wskaźnikiem na początek zarezerwowanego obszaru

$$x[i] \equiv *(x + i)$$

- ▶ na przykład instrukcja:

```
|| tab[i] = tab[tab[j] + 1];
```

jest zamieniana na:

```
|| *(tab + i) = *(tab + *(tab + j) + 1);
```

i dlatego nie wykonuje się kontroli zakresu indeksów

© 2024-26 Przemysław Kobylański

Złożone typy danych

Tablice (Ada)

- ▶ typ tablicowy w języku Ada deklaruje się następująco:

```
|| type Tablica is array (Lo .. Hi) of T;
```

gdzie Lo i Hi jest zakresem wartości indeksu a T jest typem elementów tablicy

- ▶ tablicę deklaruje się następująco:

```
|| Nazwa : Tablica;
```

- ▶ podczas działania programu kontrolowany jest zakres indeksu
- ▶ przekroczenie zakresu indeksu sygnalizowane jest wyjątkiem CONSTRAINT_ERROR
- ▶ dolny i górny zakres indeksu można sprawdzić za pomocą Nazwa'First i Nazwa'Last

© 2024-26 Przemysław Kobylański

Złożone typy danych

Tablice (Ada)

- ▶ wartości elementów tablicy można zainicjować podczas deklaracji:

```
|| type Tablica is array (1 .. 5) of Integer;  
|| Nazwa : Tablica := (1, 0, 0, 0, 5);
```

można też tak:

```
|| Nazwa : Tablica := (1 => 1, 5 => 5, others => 0);
```

- ▶ zerowanie elementów tablicy w pętli:

```
|| for I in Nazwa'Range loop -- iteracja po  
||     indeksach  
||     Nazwa(I) := 0;  
|| end loop;
```

albo prościej:

```
|| for E of Nazwa loop -- iteracja po elementach  
||     E := 0;  
|| end loop;
```

© 2024-26 Przemysław Kobylański

Złożone typy danych

Tablice (Ada)

- ▶ deklarując typ tablicowy nie trzeba podawać zakresu indeksów:

```
|| type Tablica is array (Integer range <>) of  
||     Integer;
```

- ▶ jest to bardzo przydatne przy przekazywaniu tablicy jako parametru:

```
|| function Max (A : Tablica) return Integer is  
||     M : Integer := A (A'First);  
|| begin  
||     for I in A'First + 1 .. A'Last loop  
||         if A (I) > M then  
||             M := A (I);  
||         end if;  
||     end loop;  
||     return M;  
|| end Max;
```

- ▶ zakres indeksów należy podać podczas deklarowania tablicy:

```
|| Tab1 : Tablica (1 .. 10);  
|| Tab2 : Tablica (-5 .. 5);
```

© 2024-26 Przemysław Kobylański

Złożone typy danych

Listy (Python)

- ▶ tablice w języku Python nazywa się listami
- ▶ elementy indeksuje się od 0
- ▶ elementy listy mogą być różnych typów:

```
x = [1, 3.14, 'napis', (0, 0)]
for y in x: # iteracja po elementach
    print(type(y))
for i in range(len(x)): # iteracja po indeksach
    print(x[i])
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'tuple'>
1
3.14
napis
(0, 0)
```

Złożone typy danych

Listy (Python)

- ▶ plasterek (ang. slice) listy to jej fragment:
 - ▶ $x[i]$ i-ty element
 - ▶ $x[i:j]$ elementy od i-tego do j-1-ego
 - ▶ $x[i:j:k]$ elementy od i-tego do j-1-ego z krokiem co k
 - ▶ $x[:j]$ prefiks listy długości j
 - ▶ $x[i:]$ sufiks listy od i-tego elementu
 - ▶ $x[:k]$ elementy listy z krokiem co k
 - ▶ $x[::-1]$ lista o tych samych elementach ale w odwrotnej kolejności

```
>>> x = [0, 1, 2, 3, 4, 5, 6]
>>> x[3:5]
[3, 4]
>>> x[2:5:2]
[2, 4]
>>> x[:4]
[0, 1, 2, 3]
>>> x[3:]
[3, 4, 5, 6]
>>> x[::2]
[0, 2, 4, 6]
>>> x[::-1]
[6, 5, 4, 3, 2, 1, 0]
```

Złożone typy danych

Listy (Python)

- ▶ dodawanie i usuwanie elementów:
 - ▶ $x.append(y)$ dołącz na końcu listy x wartość y
 - ▶ $x.insert(i, y)$ wstaw na i-tą pozycję wartość y
 - ▶ $x.remove(y)$ usuń pierwsze wystąpienie wartości y
 - ▶ $x.pop()$ usuń pierwszy element listy
 - ▶ $x.pop(i)$ usuń i-ty element listy

```
>>> x = []
>>> x.append(1)
>>> x.insert(0, 2)
>>> x.append(3)
>>> x
[2, 1, 3]
>>> x.remove(1)
>>> x
[2, 3]
>>> x = [2, 1, 3]
>>> x.remove(3)
>>> x
[2, 1]
>>> x.pop(1)
1
>>> x
[2]
```

Złożone typy danych

Listy (Python)

- ▶ łączenie list:

```
>>> x = [1, 2, 3]
>>> y = ['jeden', 'dwa', 'trzy']
>>> z = x + y
>>> z
[1, 2, 3, 'jeden', 'dwa', 'trzy']
>>> y.extend(x)
>>> y
['jeden', 'dwa', 'trzy', 1, 2, 3]
```

Złożone typy danych

Struktury (C)

- ▶ struktura składa się z pól przechowujących wartości
- ▶ pola mogą być różnych typów
- ▶ deklaracja typu:

```
|| struct T { T1 p1; T2 p2; ...; Tn pn;};
```

- ▶ deklaracja zmiennej:

```
|| struct T x;
```

- ▶ deklaracja i zainicjowanie zmiennej:

```
|| struct T x = {w1, w2, ..., wn};
```

- ▶ selektor pola:

```
|| x.p
```

```
|| int main(void) {  
||     struct Punkt { double x; double y; };  
||     struct Punkt p = {2.0, 3.0};  
||     p.x = p.x + 2.0;  
||     p.y = p.y - 4.0;  
|| }
```

© 2024-26 Przemysław Kobylański



Złożone typy danych

Rekordy (Ada)

- ▶ rekord składa się z pól przechowujących wartości
- ▶ pola mogą być różnych typów
- ▶ deklaracja typu:

```
|| type T is  
||     record  
||         P1 : T1; P2 : T2; ...; Pn : Tn;  
||     end record;
```

- ▶ deklaracja zmiennej:

```
|| X : T;
```

- ▶ deklaracja i zainicjowanie zmiennej:

```
|| X : T := (w1, w2, ..., wn);
```

- ▶ selektor pola:

```
|| X.P
```

© 2024-26 Przemysław Kobylański



Złożone typy danych

Rekordy (Ada)

```
|| procedure Main is  
||     type Punkt is  
||         record  
||             X : Float;  
||             Y : Float;  
||         end record;  
||     P1 : Punkt := (2.0, 3.0);  
||     P2 : Punkt := (Y => 1.0, X => -1.0);  
|| begin  
||     P1.X := P1.X + P2.X;  
||     P1.Y := P1.Y + P2.Y;  
|| end Main;
```

© 2024-26 Przemysław Kobylański



Złożone typy danych

Krotki (Python)

- ▶ krotka (ang. tuple) składa się ze składowych przechowujących wartości
- ▶ składowe mogą być różnych typów
- ▶ inicjowanie zmiennej:

```
|| x = (w1, w2, ..., wn)  
|| y = (w1,) # przecinek jest konieczny
```

- ▶ składowe indeksowane są od 0

- ▶ selektor składowej:

```
|| x[i]
```

© 2024-26 Przemysław Kobylański



Złożone typy danych

Ciekawostka



- o tym jak dziwnym językiem programowania jest C świadczy poniższy program:

```
#include <stdio.h>
int main(void) {
    int tab[] = {4, 2, 1, 0, 3};
    printf("%d\n", 2[tab]);
    return 0;
}
```

- jaka wartość zostanie wydrukowana?
- skorzystajmy z arytmetyki wskaźników:

$$2[\text{tab}] = *(2 + \text{tab}) = *(\text{tab} + 2) = \text{tab}[2] = 1$$

WYKŁAD 8.

Funkcje i procedury

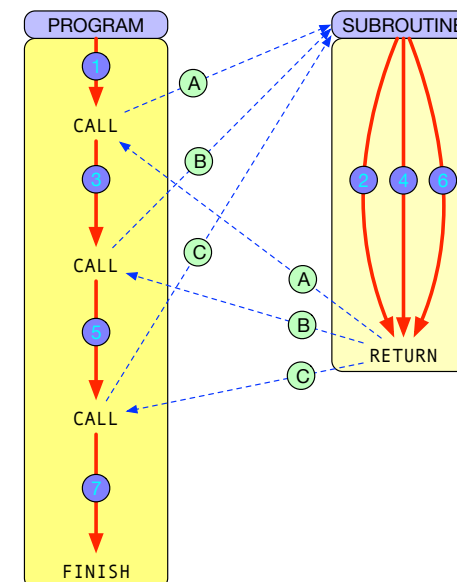
Funkcje i procedury

Podprogramy

- często ten sam ciąg instrukcji musi być wykonany w różnych miejscach programu
- zamiast powtarzać w kodzie ten sam fragment można:
 - wydzielić powtarzający się fragment
 - nazwać go i zapisać jako podprogram
 - w miejscach gdzie występowały powtarzające się fragmenty kodu umieścić wywołanie podprogramu
- działanie wydzielonego fragmentu można uzależnić od parametrów
- w miejscu wywołania podprogramu należy podać wartości parametrów dla jakich ma się wykonać wydzielony kod
- parametry zadeklarowane w podprogramie nazywa się **parametrami formalnymi** (ang. formal parameters)
- wartości podane przy wywołaniu podprogramu nazywa się **parametrami rzeczywistymi** (ang. actual parameters)

Funkcje i procedury

Podprogramy



Funkcje i procedury

Funkcje (C)

- ▶ nagłówek deklaruje funkcję, tj. opisuje jej nazwę, typ zwracanej przez nią wartości oraz nazwy i typy jej parametrów:

```
|| T nazwa(T1 p1, T2 p2, ..., Tn pn);
```

gdzie T jest typem wartości funkcji, nazwa jej nazwą, $T_1, \dots, T_n$ typami parametrów a $p_1, \dots, p_n$ nazwami parametrów

- ▶ przykład:

```
|| int max(int x, int y); // większa z dwóch wartości
```

- ▶ gdy kompilator przeczyta nagłówek funkcji wie jak poprawnie przekazać do funkcji wartości parametrów i jak odebrać od niej wynik

Funkcje i procedury

Funkcje (C)

- ▶ definicja funkcji ma w języku C następującą postać:

```
|| T nazwa(T1 p1, T2 p2, ..., Tn pn) {  
    ciąg instrukcji  
}
```

przy czym każda ścieżka wykonania ciągu instrukcji powinna kończyć się instrukcją **return** W, która przerywa działanie funkcji i zwraca wartość wyrażenia W typu T

- ▶ przykład:

```
|| int max(int x, int y) {  
    if(x > y)  
        return x;  
    else  
        return y;  
}
```

każda z dwóch ścieżek kończy się instrukcją **return** a to czy zostanie zwrócona wartość x czy y zależy od relacji między tymi wartościami

Funkcje i procedury

Funkcje (C)

- ▶ szczególnymi funkcjami w C są funkcje o typie wartości **void**
- ▶ funkcje takie nie zwracają żadnej wartości i pełnią rolę procedur z innych języków programowania
- ▶ główną różnicą między funkcją a procedurą jest kontekst jej wywołania:
 - ▶ funkcje wywołuje się w kontekście wyrażen (tj. w miejscach gdzie może wystąpić wyrażenie)
 - ▶ procedury wywołuje się w kontekście instrukcji (tj. w miejscach gdzie może wystąpić instrukcja)
- ▶ **funkcja** coś wylicza a **procedura** coś wykonuje

Funkcje i procedury

Przekazywanie parametrów (C)

- ▶ w języku C dostępne jest tylko przekazanie parametru **przez wartość**
- ▶ jeśli należy odebrać z podprogramu wartość, to przekazaną do podprogramu wartością parametru jest adres gdzie podprogram ma odłożyć oddawaną wartość
- ▶ przykład:

```
|| scanf("%d", &n);
```

funkcja `scanf` wie jak zinterpretować wczytany ciąg znaków (pierwszym parametrem jest format) oraz wie gdzie odłożyć przeczytaną wartość (drugim parametrem jest wskazanie na miejsce pamięci)

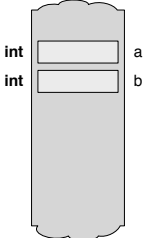
Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera



1. w pamięci znajdują się zmienne a i b typu int

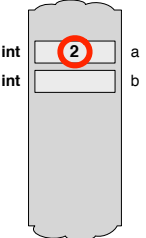
Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2

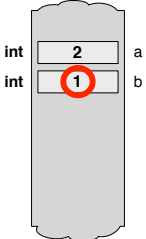
Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1

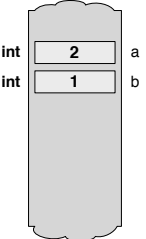
Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera



1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int 2 a
int 1 b
int x

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int 2 a
int 1 b
int x
int* y

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int 2 a
int 1 b
int 2 x
int* y

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int 2 a
int 1 b
int 2 x
int* 2 y

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętany w zmiennej y o 1

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętany w zmiennej y o 1
12. kończy się wykonanie funkcji p

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int 2 a
int 2 b

int 3 x

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętany w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int 2 a
int 2 b

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętany w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

pamięć komputera

int 2 a
int 2 b

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętany w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x
15. sterowanie powraca do miejsca za wywołaniem funkcji p

Funkcje i procedury

Przekazywanie parametrów (C)

```
void p(int x, int *y)
{
    x = x+1;
    *y = *y+1;
}

int a, b;
a = 2;
b = 1;
p(a, &b);
```

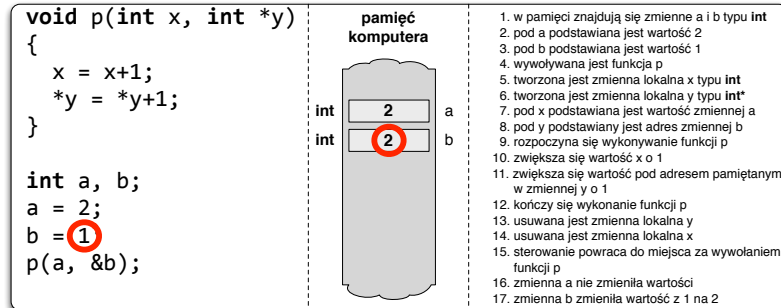
pamięć komputera

int 2 a
int 2 b

1. w pamięci znajdują się zmienne a i b typu int
2. pod a podstawiana jest wartość 2
3. pod b podstawiana jest wartość 1
4. wywoływana jest funkcja p
5. tworzona jest zmienna lokalna x typu int
6. tworzona jest zmienna lokalna y typu int*
7. pod x podstawiana jest wartość zmiennej a
8. pod y podstawiany jest adres zmiennej b
9. rozpoczyna się wykonywanie funkcji p
10. zwiększa się wartość x o 1
11. zwiększa się wartość pod adresem pamiętany w zmiennej y o 1
12. kończy się wykonanie funkcji p
13. usuwana jest zmienna lokalna y
14. usuwana jest zmienna lokalna x
15. sterowanie powraca do miejsca za wywołaniem funkcji p
16. zmienna a nie zmieniła wartości

Funkcje i procedury

Przekazywanie parametrów (C)



Funkcje i procedury

Przekazywanie parametrów (C)

- przekazywanie tablicy T t[n] jako parametr
- można to zrobić na kilka sposobów:

```
// tablica to tylko wskazanie:
f(int n, T* p) {
    ...
    p[i] = 0; // co jest równoważne *(p+i)=0
    ...
}

// ale wskazanie nie tylko na jedną wartość:
g(int n, T p[]) {
    ...
    p[i] = 0; // co jest równoważne *(p+i)=0
    ...
}

// to nadal tylko przekazanie wskaźnika:
h(int n, T p[n]) {
    ...
    p[i] = 0; // co jest równoważne *(p+i)=0
    ...
}
```

Funkcje i procedury

Przekazywanie parametrów (C)

- przykład:

```
#include <stdio.h>

int max(int n, int x[n]) {
    // sizeof(x) = sizeof(int*) = 8
    int m = x[0];
    for(int i = 1; i < n; i++)
        if(x[i] > m)
            m = x[i];
    return m;
}

int main(void) {
    int t[] = {4, 2, 6, 3, 1};
    // sizeof(t) = 5 * sizeof(int) = 20
    printf("max=%d\n", max(sizeof(t)/sizeof(int), t));
    return 0;
}
```

Funkcje i procedury

Procedury (Ada)

- nagłówek deklaruje procedurę:

```
|| procedure Nazwa (P1 : M1 T1; ...; Pn : Mn Tn);
```

gdzie Nazwa jest jej nazwą, $P_1, \dots, P_n$ są nazwami jej parametrów, $M_1, \dots, M_n$ są trybami przekazywania parametrów a $T_1, \dots, T_n$ są typami parametrów

- przykład:

```
procedure Min_Max (
    X : in Integer;
    Y : in Integer;
    Min : out Integer; -- mniejsza z wartości
    Max : out Integer); -- większa z wartości
```

Funkcje i procedury

Procedury (Ada)

- definicja procedury ma w Adzie następującą postać:

```
procedure Nazwa (P1 : M1 T1; ...; Pn : Mn Tn) is
  deklaracje lokalne
begin
  ciąg instrukcji
end Nazwa;
```

- przykład:

```
procedure Min_Max (
  X : in Integer;
  Y : in Integer;
  Min : out Integer; -- mniejsza z wartości
  Max : out Integer) -- większa z wartości
is
begin
  if X < Y then
    Min := X; Max := Y;
  else
    Min := Y; Max := X;
  end if;
end Min_Max;
```

Funkcje i procedury

Przekazywanie parametrów (Ada)

- w języku Ada dostępne są następujące tryby przekazywania parametrów:
 - in** do podprogramu dostarczana jest wartość
 - out** z podprogramu odbierana jest wartość
 - in out** wartość dostarczona do podprogramu jest zmieniana
- przykład:

```
procedure Zliczaj_Przekraczające_Próg (
  Wartość : in Integer;
  Próg : in Integer;
  Przekroczone : out Boolean;
  Licznik : in out Integer) is
begin
  if Wartość > Próg then
    Przekroczone := True;
    Licznik := Licznik + 1;
  else
    Przekroczone := False;
  end if;
end Zliczaj_Przekraczające_Próg;
```

Funkcje i procedury

Przekazywanie parametrów (Ada)

- tryby przekazywania parametrów podlegają pewnym ograniczeniom:

tryb	parametr rzeczywisty	
	stała lub wyrażenie	zmienna
in	TAK ⁴	TAK ⁵⁶
out	NIE	TAK ⁷
in out	NIE	TAK ⁸

⁴Parametr formalny nie może zmienić swojej wartości w podprogramie.
⁵Zmienna musi być zainicjowana przed wywołaniem podprogramu.
⁶Parametr formalny nie może zmienić swojej wartości w podprogramie.
⁷Parametr formalny musi być podstawiony podczas wykonywania podprogramu
⁸Zmienna musi być zainicjowana przed wywołaniem podprogramu.

Funkcje i procedury

Funkcje (Ada)

- nagłówek deklaruje funkcję:
- definicja funkcji ma następującą postać:
- każda ścieżka obliczeń w ciągu instrukcji musi zakończyć się instrukcją **return** W, która przerywa działanie funkcji i zwraca wartość wyrażenia W typu T

```
function Nazwa (P1 : in T1; ...; Pn : in Tn)
  return T;

function Nazwa (P1 : in T1; ...; Pn : in Tn)
is
  deklaracje lokalne
begin
  ciąg instrukcji
end Nazwa;
```

Funkcje i procedury

Funkcje (Ada)

► przykład:

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Main is
  function Max (X : in Integer; Y : in Integer)
    return Integer is
  begin
    if X > Y then
      return X;
    else
      return Y;
    end if;
  end Max;
  A, B : Integer;
begin
  Get (A);
  Get (B);
  Put (Max (A, B));
end Main;
```

Funkcje i procedury

Przeciążanie operatorów (Ada)

- można przedefiniować operator definiując jego działanie dla innych typów

► przykład:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is

  function "+" (X : in String; Y : in String)
    return String is
  begin
    return X & Y; -- konkatenacja
  end "+";

begin
  Put_Line ("Hello" + ", " + "World" + "!");
end Main;
```

Funkcje i procedury

Funkcje (Python)

- definicja funkcji ma następującą postać:

```
def nazwa(p1, ..., pn):
    ciąg instrukcji
```

gdzie nazwa jest nazwą funkcji a $p_1, \dots, p_n$ są parametrami funkcji

- każda ścieżka wykonania ciągu instrukcji musi zakończyć się instrukcją **return** W, która przerywa działanie funkcji i zwraca wartość wyrażenia W

► przykład:

```
def max(x, y):
    if x > y:
        return x
    else:
        return y

print(max(7, 13)) # 13
```

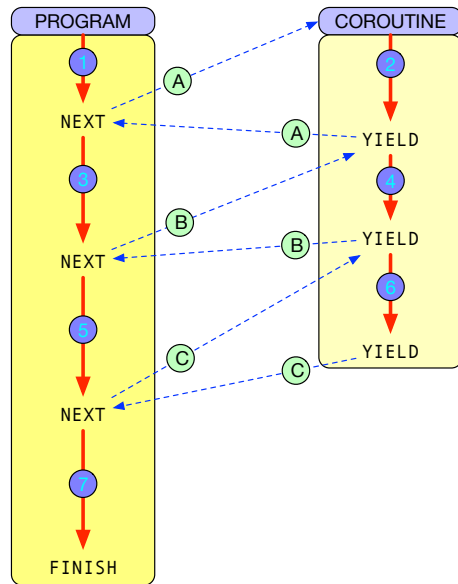
Funkcje i procedury

Współprogramy (Python)

- szczególnym rodzajem podprogramów są **współprogramy** (korutyny)
- działają one podobnie do podprogramu ale mogą przerwać swoje działanie do momentu gdy zostanie ono wznowione
- wznowienie działania odbywa się od miejsca gdzie zostało ono przerwane
- jeśli program i współprogram są na zmianę przerywane i wznowiane, to wykonują się **współbieżnie** (ale nie równolegle)
- można sobie wyobrazić, że fragmenty kodu programu i współprogramu są przeplecione

Funkcje i procedury

Współprogramy (Python)



© 2024-26 Przemysław Kobylański

Funkcje i procedury

Współprogramy (Python)

- ▶ korutyna jako generator dostarczający kolejno trzy napisy:

```
def coroutine():  
    yield 'one'  
    yield 'two'  
    yield 'three'
```

- ▶ generowanie kolejnych wartości:

```
x = coroutine()  
print(next(x)) # one  
print(next(x)) # two  
print(next(x)) # three
```

- ▶ iteracja po kolejnych wartościach:

```
for x in coroutine():  
    print(x, end = ' ') # one two three
```

- ▶ do współprogramów w Pythonie powrócimy przy okazji przeglądu z nawrotami

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Koncepcja struktury dynamicznej



- ▶ serwisy społecznościowe (Facebook, BlueSky, Twitter/X itp.) są przykładami dynamicznie zmieniających się sieci:
 1. użytkownicy są dodawani – **rośnie liczba węzłów**
 2. użytkownicy są usuwani – **maleje liczba węzłów**
 3. znajomi są dodawani – **rośnie liczba połączeń**
 4. znajomi są usuwani – **maleje liczba połączeń**
- ▶ sieć w najprostszy sposób można reprezentować za pomocą **tablicy** ale taka reprezentacja oprócz nielicznych zalet ma wiele wad
- ▶ zdecydowanie lepiej do ich reprezentowania użyć **struktur dynamicznych** budowanych przy użyciu **wskaźników**

© 2024-26 Przemysław Kobylański

© 2024-26 Przemysław Kobylański

WYKŁAD 9.

Struktury dynamiczne – listy

Struktury dynamiczne – listy

Koncepcja struktury dynamicznej

- ▶ sieć/graf $G = (V, A)$ jest uporządkowaną parą dwóch zbiorów: zbioru węzłów $V = \{1, 2, \dots, n\}$, zbioru łuków $A \subseteq V^2$
- ▶ łuk prowadzący od węzła $i \in V$ do węzła $j \in V$ reprezentuje się parą uporządkowaną $(i, j) \in A$
- ▶ jedną z reprezentacji grafu G jest macierz incydencji $M = (m_{ij})_{n \times n}$ o wartościach zero-jedynkowych
- ▶ łuki grafu zapisuje się w następujący sposób:

$$\forall i \in V \forall j \in V \quad m_{ij} = 1 \leftrightarrow (i, j) \in A$$

- ▶ dodanie i usunięcie łuku jest proste – wystarczy umieścić w odpowiednim elemencie tablicy wartość 1 albo 0
- ▶ **jak dodać i usunąć węzeł?**
- ▶ w językach takich jak C czy Ada tablica jest statyczna tj. nie zmienia rozmiaru
- ▶ jedyna możliwość to: zarezerwowanie w pamięci miejsca na większą tablicę i przepisanie rozszerzanej tablicy w nowe miejsce

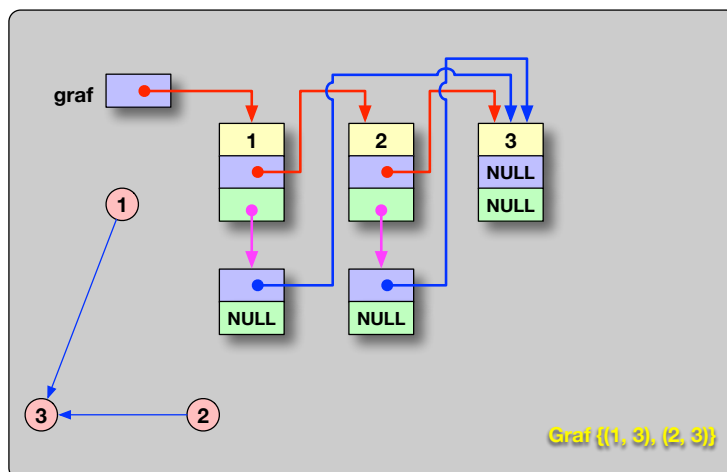
Struktury dynamiczne – listy

Koncepcja struktury dynamicznej

- ▶ do reprezentowania sieci użyjemy struktury dynamicznej zbudowanej przy użyciu wskaźników
- ▶ **węzeł** będzie reprezentowany strukturą
 - ▶ wśród pól tej struktury jest informacja o numerze węzła, wskazanie na następny węzeł w grafie i wskazanie na pierwszy łuk wychodzący z tego węzła
- ▶ **łuk** jest reprezentowany strukturą
 - ▶ wśród pól tej struktury jest wskazanie na węzeł do którego prowadzi łuk i wskazanie na następny wychodzący łuk
- ▶ powyższą strukturę dynamiczną omówimy na przykładzie prostego grafu

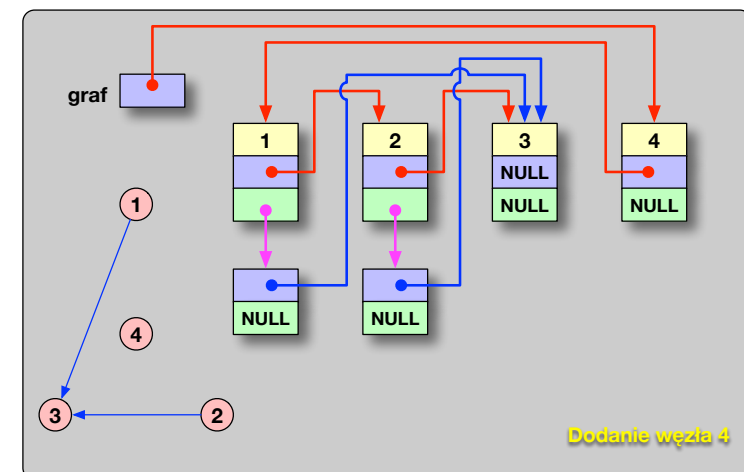
Struktury dynamiczne – listy

Koncepcja struktury dynamicznej



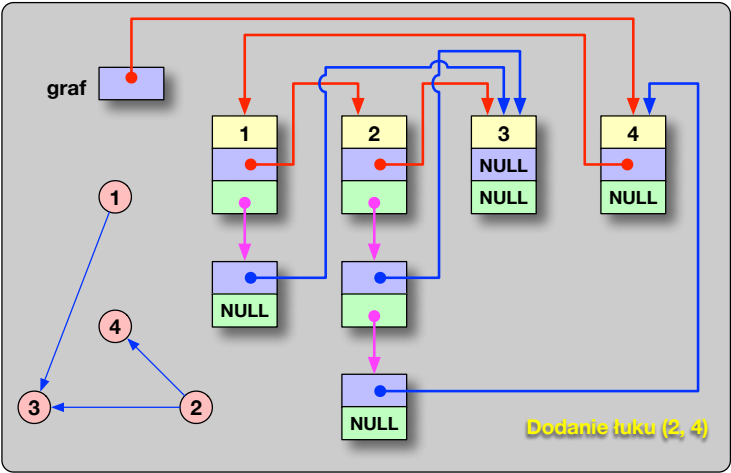
Struktury dynamiczne – listy

Koncepcja struktury dynamicznej



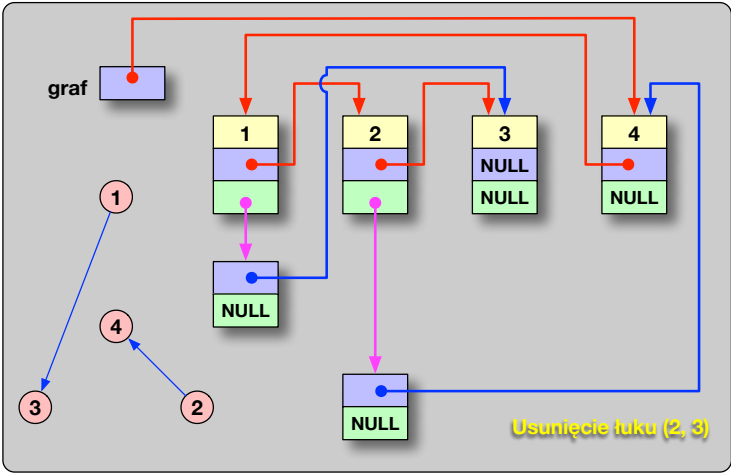
Struktury dynamiczne – listy

Koncepcja struktury dynamicznej



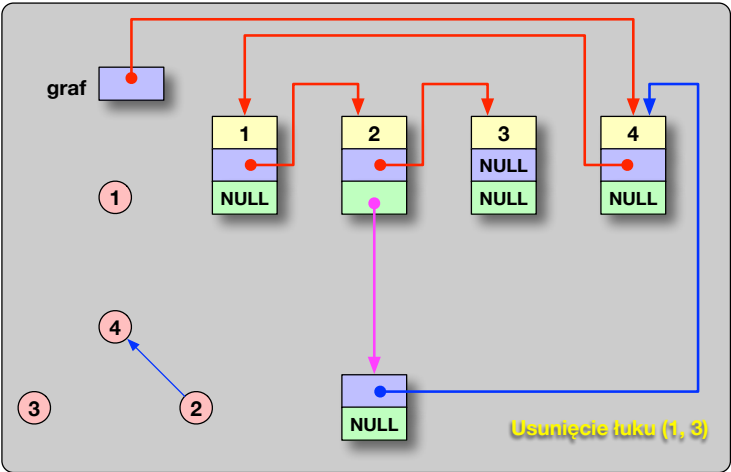
Struktury dynamiczne – listy

Koncepcja struktury dynamicznej



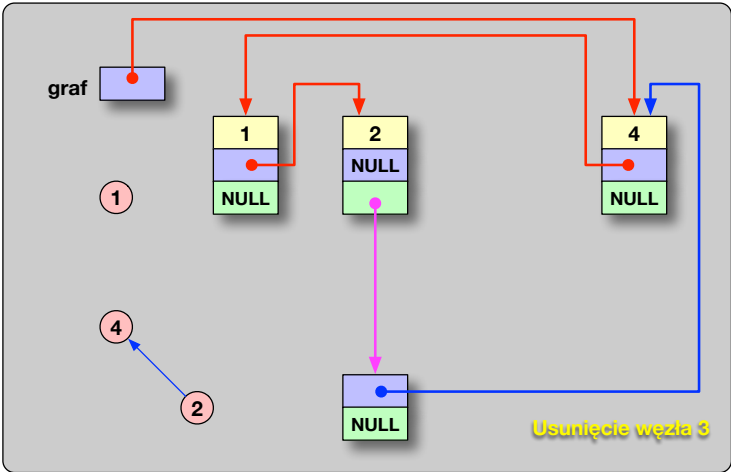
Struktury dynamiczne – listy

Koncepcja struktury dynamicznej



Struktury dynamiczne – listy

Koncepcja struktury dynamicznej



Struktury dynamiczne – listy

Listy jednokierunkowe

- ▶ **lista jednokierunkowa** składa się z **węzłów**
- ▶ węzeł reprezentowany jest strukturą **Node**
- ▶ struktura **Node** posiada dwa pola:
 - id** przechowuje identyfikator węzła
 - next** przechowuje wskazanie na następny węzeł na liście
- ▶ struktura **Node** może posiadać jeszcze inne pola przechowujące dodatkowe informacje o węźle
- ▶ pole **id** jest typu **całkowitego**
- ▶ pole **next** jest typu **wskazanie na strukturę Node**
- ▶ aby dostać się do węzłów musi być wskaźnik **first** typu **wskazanie na strukturę Node** wskazujący pierwszy węzeł na liście
- ▶ jeśli **first** jest pustym wskaźnikiem, to lista jest pusta
- ▶ jeśli pole **next** jest pustym wskaźnikiem, to węzeł jest ostatnim na liście

Struktury dynamiczne – listy

Listy jednokierunkowe

- ▶ deklaracja listy jednokierunkowej w języku C:

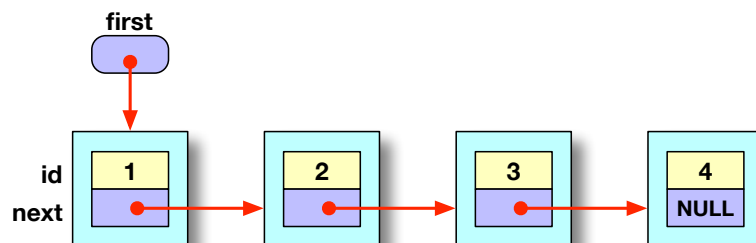
```
struct Node {           // struktura węzła
    int id;              // identyfikator węzła
    struct Node* next;   // wskazanie na następny węzeł
};

typedef struct Node* List;

List first = NULL;      // wskazanie na pustą listę
```

Struktury dynamiczne – listy

Listy jednokierunkowe



Struktury dynamiczne – listy

Listy jednokierunkowe

- ▶ drukowanie listy w C:

```
void drukuj(List ptr) {
    while(ptr != NULL) {
        printf("%d ", ptr->id);
        ptr = ptr->next;
    }
    printf("\n");
}
```

- ▶ wstawianie na początek listy w C:

```
void na_początek(List* ptr, int n) {
    List temp = malloc(sizeof(struct Node));
    temp->id = n;
    temp->next = *ptr;
    *ptr = temp;
}
```

Struktury dynamiczne – listy

Listy jednokierunkowe

► wstawianie na koniec listy w C:

```
void na_koniec(List* ptr, int n) {
    List temp = malloc(sizeof(struct Node));
    temp->id = n;
    temp->next = NULL;
    List curr = *ptr;
    if(curr == NULL)
        *ptr = temp;
    else {
        while(curr->next != NULL)
            curr = curr->next;
        curr->next = temp;
    }
}
```

Struktury dynamiczne – listy

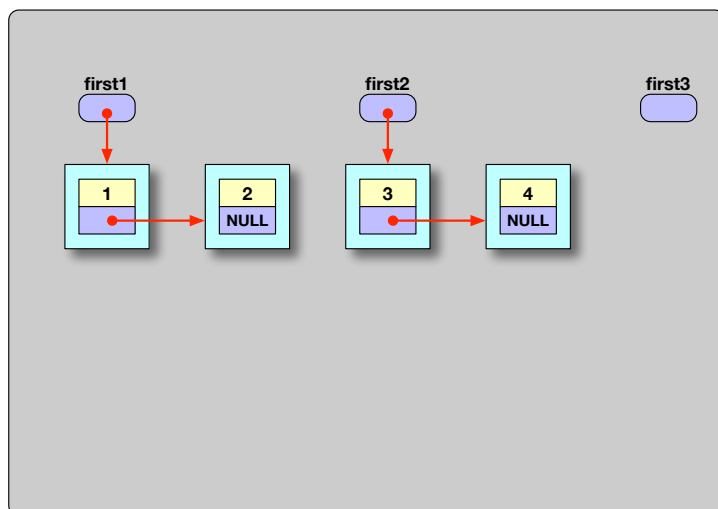
Listy jednokierunkowe

► połączenie dwóch list w C:

```
// odpowiednik predykatu append/3 z języka Prolog
//
void połącz(List ptr1, List ptr2, List* ptr3) {
    while(ptr1 != NULL) {
        *ptr3 = malloc(sizeof(struct Node));
        (*ptr3)->id = ptr1->id;
        ptr1 = ptr1->next;
        ptr3 = &((*ptr3)->next);
    }
    *ptr3 = ptr2;
}
...
połącz(first1, first2, &first3); // wywołanie
```

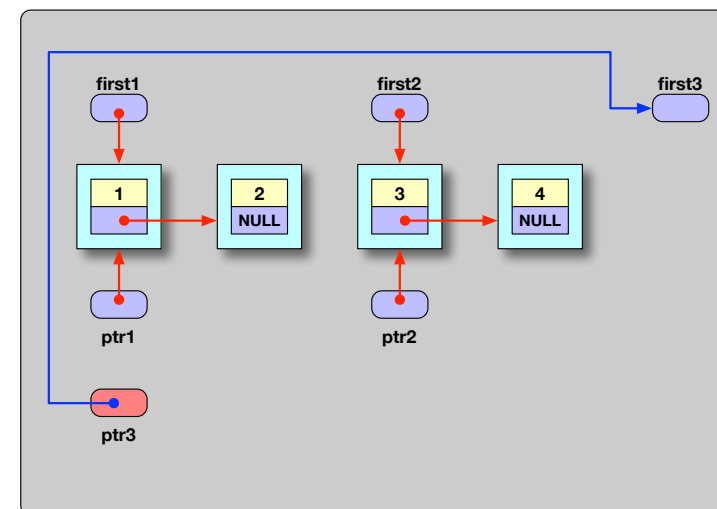
Struktury dynamiczne – listy

Listy jednokierunkowe



Struktury dynamiczne – listy

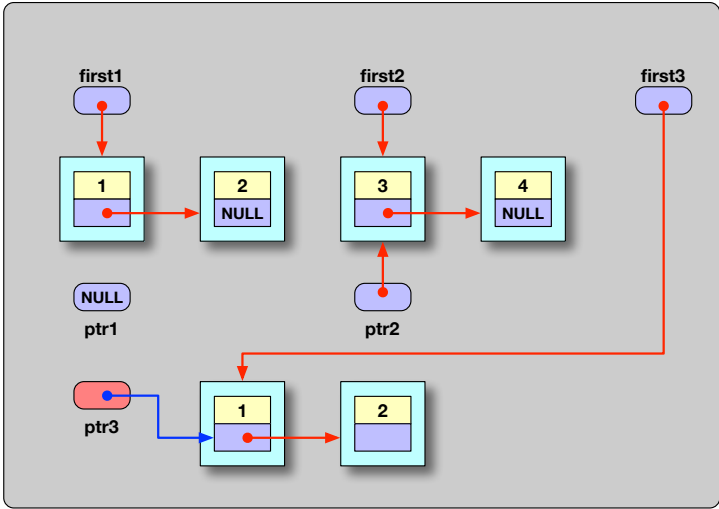
Listy jednokierunkowe



© 2024-26 Przemysław Kobylański

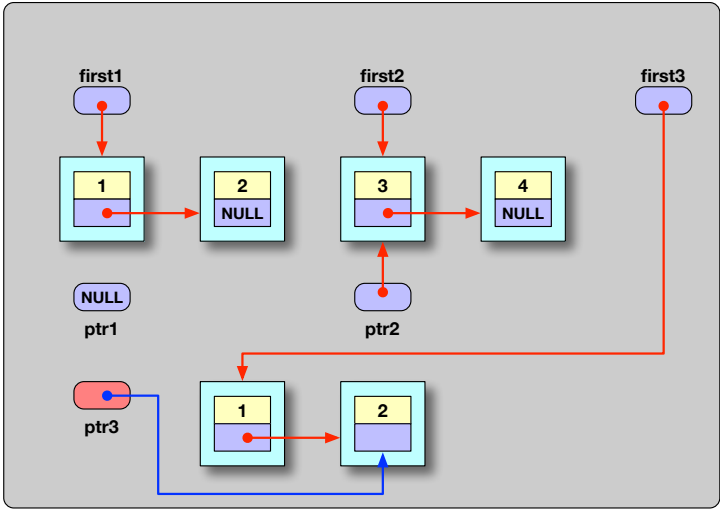
Struktury dynamiczne – listy

Listy jednokierunkowe



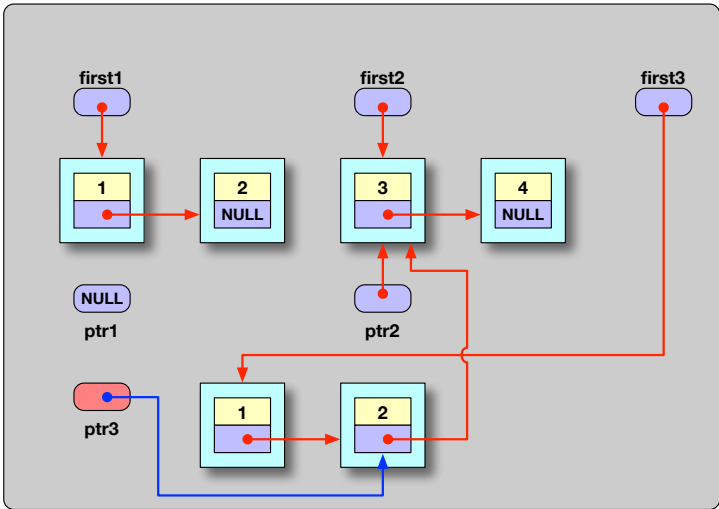
Struktury dynamiczne – listy

Listy jednokierunkowe



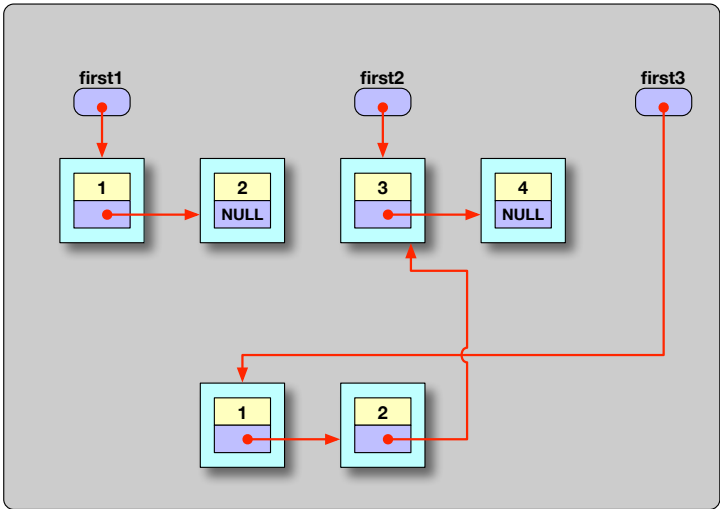
Struktury dynamiczne – listy

Listy jednokierunkowe



Struktury dynamiczne – listy

Listy jednokierunkowe



Struktury dynamiczne – listy

Listy jednokierunkowe

Ćwiczenie

Prześledź w analogiczny sposób rysując⁹ na kartce działanie funkcji `połącz()` dla wywołania:

```
List first = NULL;
na_koniec(&first, 1);
na_koniec(&first, 2);
połącz(first, first, &first);
```

⁹Przyda się ołówek i gumka do wymazywania zmieniających się wskaźników.

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Listy jednokierunkowe

► usunięcie całej listy w C:

```
void usuń_listę(List* ptr) {
    List temp;
    while(*ptr != NULL) {
        List temp = (*ptr)->next;
        free(*ptr);
        *ptr = temp;
    }
}
```

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Listy jednokierunkowe

► deklaracja listy jednokierunkowej w języku Ada:

```
type Node;           -- zapowiedź typu

type List is access Node;

type Node is
    record
        Id      : Integer; -- identyfikator węzła
        Next    : List;    -- wskazanie na następny węzeł
    end record;

First : List;         -- wskazanie na pustą listę
```

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Listy jednokierunkowe

► drukowanie listy w Adzie:

```
procedure Drukuj (Ptr : List) is
    Curr : List := Ptr;
begin
    while Curr /= null loop
        Put (Curr.all.Id);
        Curr := Curr.all.Next;
    end loop;
    New_Line;
end Drukuj;
```

► wstawianie na początek listy w Adzie:

```
procedure Na_Początek (Ptr : in out List; N : Integer)
is
    Temp : List := new Node;
begin
    Temp.all.Id := N;
    Temp.all.Next := Ptr;
    Ptr := Temp;
end Na_Początek;
```

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Listy jednokierunkowe

► wstawianie na koniec listy w Adzie:

```
procedure Na_Koniec (Ptr : in out List; N : Integer)
is
begin
    Temp : List := new Node;
    Curr : List := Ptr;
begin
    Temp.all.Id := N;
    if Curr = null then
        Ptr := Temp;
    else
        while Curr.all.Next /= null loop
            Curr := Curr.all.Next;
        end loop;
        Curr.all.Next := Temp;
    end if;
end Na_Koniec;
```

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Listy jednokierunkowe

► połączenie dwóch list w Adzie:

```
procedure Połącz (Ptr1,Ptr2: List; Ptr3: out List) is
    Curr : List := Ptr1;
    Prev : List := null;
    Temp : List;
begin
    Ptr3 := Ptr2;
    while Curr /= null loop
        Temp := new Node;
        if Prev = null then
            Ptr3 := Temp;
        else
            Prev.all.Next := Temp;
        end if;
        Temp.all.Id := Curr.all.Id;
        Temp.all.Next := Ptr2;
        Curr := Curr.all.Next;
        Prev := Temp;
    end loop;
end Połącz;
```

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Listy jednokierunkowe

Ćwiczenie

Prześledź rysując¹⁰ na kartce działanie procedury Połącz dla wywołania:

```
First1 : List;
First2 : List;
First3 : List;
...
Na_Koniec(First1, 1);
Na_Koniec(First1, 2);
Na_Koniec(First2, 3);
Na_Koniec(First2, 4);
Polacz(First1, First2, First3);
```

¹⁰Przyda się ołówek i gumka do wymazywania zmieniających się wskaźników.

Struktury dynamiczne – listy

Listy jednokierunkowe

► usunięcie całej listy w Adzie:

```
with Ada.Unchecked_Deallocation;
...
procedure Delete is new
    Ada.Unchecked_Deallocation (Node, List);

procedure Usuń_Listę (Ptr : in out List) is
    Temp : List;
begin
    while Ptr /= null loop
        Temp := Ptr.all.Next;
        Delete (Ptr);
        Ptr := Temp;
    end loop;
end Usuń_Listę;
```

© 2024-26 Przemysław Kobylański

Struktury dynamiczne – listy

Listy dwukierunkowe

- ▶ **lista dwukierunkowa** składa się z **węzłów**
- ▶ węzeł reprezentowany jest strukturą **Node**
- ▶ struktura **Node** posiada trzy pola:
 - id** przechowuje identyfikator węzła
 - next** przechowuje wskazanie na następny węzeł na liście
 - prev** przechowuje wskazanie na poprzedni węzeł na liście
- ▶ pole **id** jest typu **całkowitego**
- ▶ pole **next** jest typu **wskazanie na strukturę Node**
- ▶ pole **prev** jest typu **wskazanie na strukturę Node**
- ▶ wskaźnik **first** typu **wskazanie na strukturę Node** wskazuje pierwszy węzeł na liście

Struktury dynamiczne – listy

Listy dwukierunkowe

- ▶ deklaracja listy dwukierunkowej w języku C:

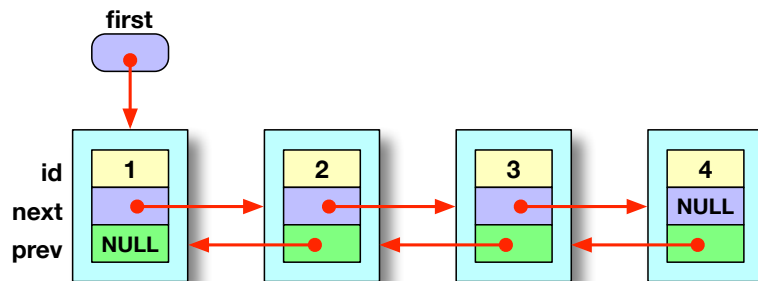
```
struct Node {           // struktura węzła
    int id;              // identyfikator węzła
    struct Node* next;   // wskazanie na następny węzeł
    struct Node* prev;   // wskazanie na poprzedni węzeł
};

typedef struct Node* List;

List first = NULL;      // wskazanie na pustą listę
```

Struktury dynamiczne – listy

Listy dwukierunkowe



Struktury dynamiczne – listy

Listy dwukierunkowe

- ▶ deklaracja listy dwukierunkowej w języku Ada:

```
type Node;              -- zapowiedź typu

type List is access Node;

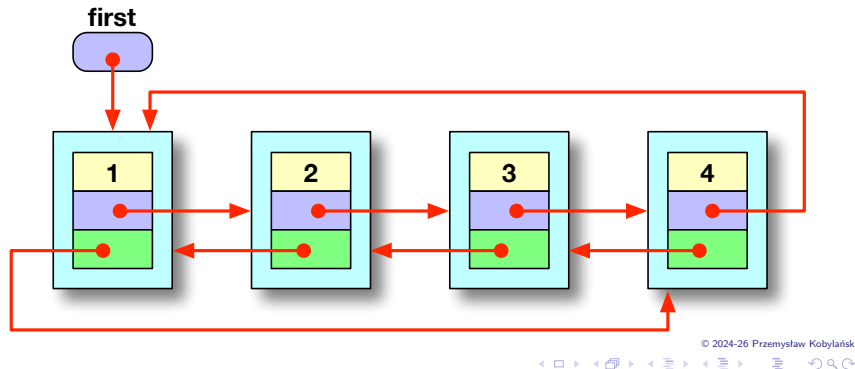
type Node is
record
    Id      : Integer;    -- identyfikator węzła
    Next    : List;       -- wskazanie na następny węzeł
    Prev    : List;       -- wskazanie na poprzedni węzeł
end record;

First : List;            -- wskazanie na pustą listę
```

Struktury dynamiczne – listy

Listy cykliczne

- ▶ **listą cykliczną** nazywamy listę dwukierunkową, w której za ostatnim elementem jest jej pierwszy element
- ▶ na liście cyklicznej wszystkie elementy są równorzędne bo lista nie ma początku ani końca



WYKŁAD 10.

Struktury dynamiczne – drzewa

Struktury dynamiczne – drzewa

Rozważania o efektywności

- ▶ założmy, że mamy kolekcję n obiektów
 - ▶ obiekty rozróżniamy po ich identyfikatorach
- $$\{id_i : i = 1, 2, \dots, n\}$$
- ▶ możemy przyjąć, że identyfikatorami są liczby całkowite
 - ▶ jak przechowywać kolekcję aby szybko wyszukiwać zadany identyfikator?
 - ▶ najprostsze pomysły:
 1. **tablica**
 2. **uporządkowana tablica**
 3. **lista jednokierunkowa**
 4. **uporządkowana lista jednokierunkowa**

Struktury dynamiczne – drzewa

Rozważania o efektywności

▶ kolekcja w tablicy

reprezentacja

- ▶ elementami nieuporządkowanej tablicy są liczby będące identyfikatorami obiektów

zalety

- ▶ brak

wady

- ▶ wyszukiwanie w czasie $O(n)$
- ▶ usunięcie identyfikatora nie zmniejsza rozmiaru tablicy
- ▶ wstawienie identyfikatora wymaga przepisania tablicy w nowe miejsce w czasie $O(n)$

Struktury dynamiczne – drzewa

Rozważania o efektywności

► kolekcja w uporządkowanej tablicy

reprezentacja

- elementy tablicy, będące liczbowymi identyfikatorami, są uporządkowane rosnąco

zalety

- szybkie wyszukiwanie binarne w czasie $O(\log n)$

wady

- usunięcie identyfikatora nie zmniejsza rozmiaru tablicy i może popsuć uporządkowanie
- naprawienie uporządkowania wymaga przesunięcia elementów w czasie $O(n)$
- wstawienie identyfikatora wymaga przepisania tablicy w nowe miejsce w czasie $O(n)$

Struktury dynamiczne – drzewa

Rozważania o efektywności

► kolekcja na liście jednokierunkowej

reprezentacja

- węzły listy przechowują identyfikatory obiektów

zalety

- wstawianie identyfikatora w stałym czasie $O(1)$
- usunięcie identyfikatora w stałym czasie $O(1)$ po wcześniejszym jego odszukaniu

wady

- wyszukiwanie identyfikatora w czasie $O(n)$

Struktury dynamiczne – drzewa

Rozważania o efektywności

► kolekcja na uporządkowanej liście jednokierunkowej

reprezentacja

- węzły listy przechowują identyfikatory obiektów i są uporządkowane rosnąco

zalety

- wstawianie identyfikatora w stałym czasie $O(1)$ po wcześniejszym wyszukaniu miejsca
- usunięcie identyfikatora w stałym czasie $O(1)$ po wcześniejszym jego odszukaniu

wady

- wyszukiwanie identyfikatora w czasie $O(n)$
- wyszukanie miejsca w czasie $O(n)$

Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

- uporządkowanie listy jednokierunkowej nie poprawiło efektywności operacji w stosunku do operacji na liście nieuporządkowanej
- **drzewo binarne** umożliwia szybkie wyszukiwanie, wstawianie i usuwanie identyfikatorów
- struktura drzewa binarnego pozwala pomijać duże fragmenty drzewa podczas jego przeszukiwania
- uporządkowane drzewo binarne składa się z **węzłów**
- każdy węzeł przechowuje identyfikator oraz wskaźniki na **lewe** i **prawe poddrzewo**
- każde z poddrzew jest uporządkowanym drzewem binarnym
- identyfikatory są uporządkowane w ten sposób, że w lewym poddrzewie wskazanym z węzła o identyfikatorze id znajdują się węzły o identyfikatorach mniejszych od id a w prawym większe od id

Struktury dynamiczne – drzewa

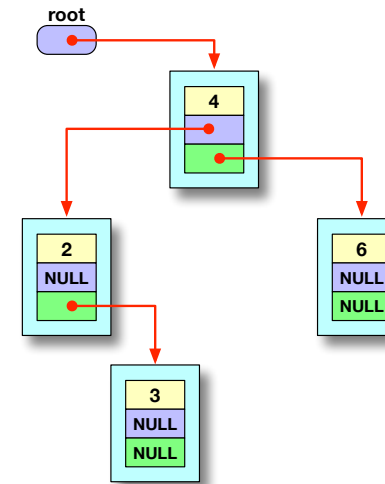
Uporządkowane drzewa binarne

► deklaracja drzewa binarnego w C:

```
struct Node {  
    int id;           // identyfikator  
    struct Node* left; // wskazanie na lewe poddrzewo  
    struct Node* right; // wskazanie na prawe poddrzewo  
};  
  
typedef struct Node* Tree;  
  
Tree root = NULL; // puste drzewo
```

Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

► drukowanie drzewa binarnego w C:

```
void drukuj(Tree ptr) {  
    if(ptr != NULL) {  
        // gdy drzewo niepuste  
        drukuj(ptr->left); // wydrukuj mniejsze id.  
        printf("%d ", ptr->id); // wydrukuj id.  
        drukuj(ptr->right); // wydrukuj większe id.  
    }  
}  
...  
drukuj(root); // wywołanie
```

Struktury dynamiczne – drzewa

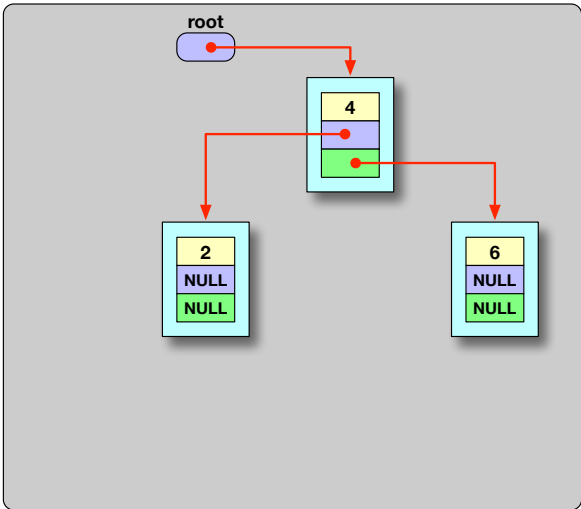
Uporządkowane drzewa binarne

► wstawianie do drzewa uporządkowanego w C:

```
void wstaw(Tree* ptr, int n) {  
    while(*ptr != NULL) {  
        if(n < (*ptr)->id)  
            ptr = &((*ptr)->left); // idź w lewo  
        else if(n > (*ptr)->id)  
            ptr = &((*ptr)->right); // idź w prawo  
        else  
            return; // n już jest w drzewie  
    }  
    *ptr = malloc(sizeof(struct Node));  
    (*ptr)->id = n; // wstaw n  
    (*ptr)->left = (*ptr)->right = NULL;  
}  
...  
wstaw(&root, n); // wywołanie
```

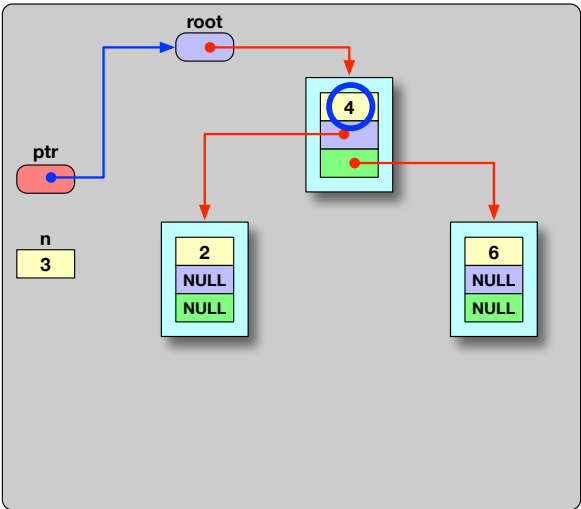
Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



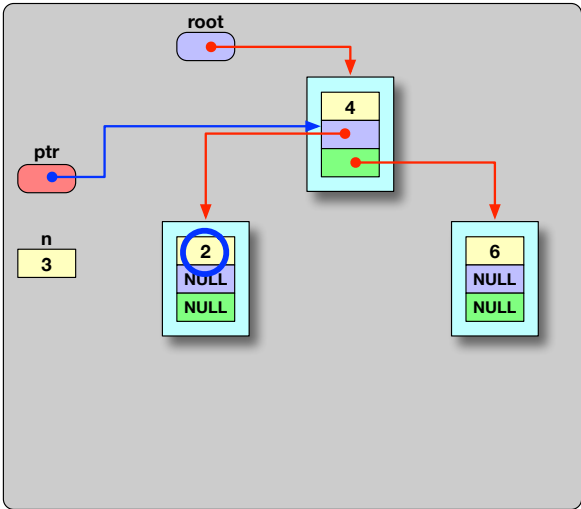
Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



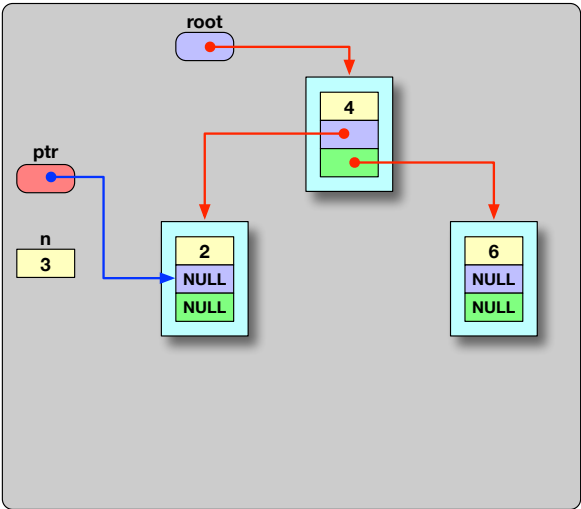
Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



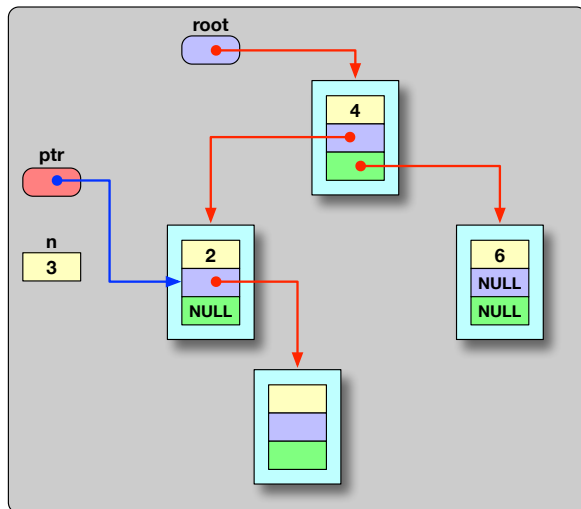
Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



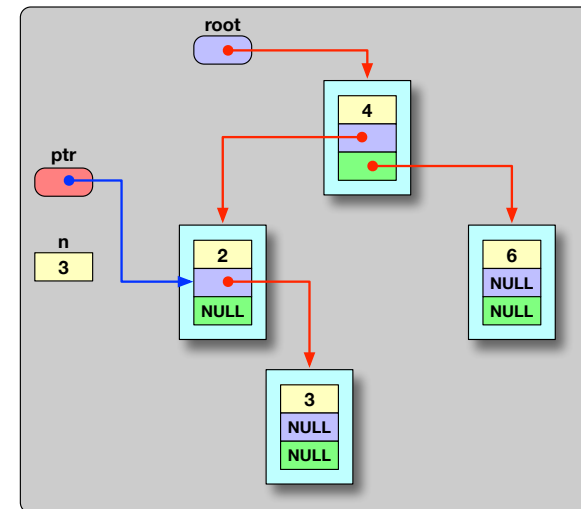
Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



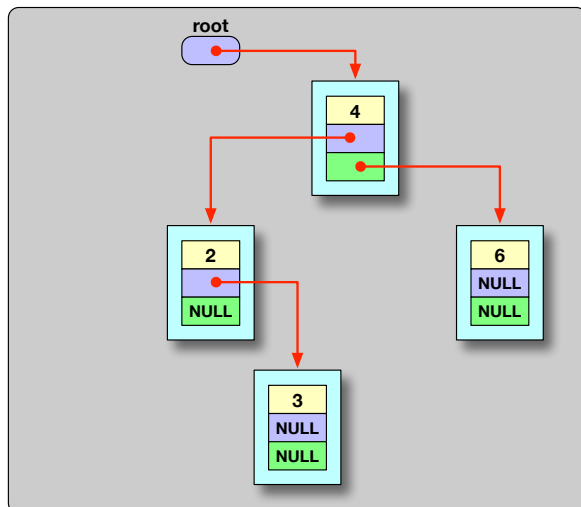
Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne



Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

► rekurencyjne wstawianie do drzewa uporządkowanego w C:

```
void rek_wstaw(Tree* ptr, int n) {
    if(*ptr == NULL) {
        *ptr = malloc(sizeof(struct Node));
        (*ptr)->id = n; // wstaw n
        (*ptr)->left = (*ptr)->right = NULL;
    }
    else if(n < (*ptr)->id)
        rek_wstaw(&((*ptr)->left), n); // idź w lewo
    else if(n > (*ptr)->id)
        rek_wstaw(&((*ptr)->right), n); // idź w prawo
}
...
rek_wstaw(&root, n); // wywołanie
```

Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

Ćwiczenie

Napisz w języku C funkcję **usuń**, która **zachowując uporządkowanie** usuwa daną wartość z drzewa.

- ▶ podczas usuwania węzła drzewo może rozpaść się na dwa kawałki i trzeba je na nowo połączyć zachowując uporządkowanie

Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

- ▶ rekurencyjne usunięcie całego drzewa w C:

```
void rek_usuń_drzewo (Tree ptr) {  
    if(ptr != NULL) {  
        rek_usuń_drzewo(ptr->left);  
        rek_usuń_drzewo(ptr->right);  
        free(ptr);  
    }  
}
```

Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

- ▶ deklaracja drzewa binarnego w Adzie:

```
type Node;  
type Tree is access Node;  
  
type Node is  
    record  
        Id      : Integer;  
        Left    : Tree;  
        Right   : Tree;  
    end record;  
  
Root : Tree;
```

Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

- ▶ drukowanie drzewa binarnego w Adzie:

```
procedure Drukuj (Ptr : Tree) is  
begin  
    if Ptr /= null then  
        Drukuj (Ptr.all.Left);  
        Put (Ptr.all.Id);  
        Drukuj (Ptr.all.Right);  
    end if;  
end Drukuj;  
...  
Drukuj (Root); -- wywołanie
```


Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

- wstawianie do drzewa uporządkowanego w Adzie:

```
procedure Wstaw (Ptr : in out Tree; N : Integer) is
  Curr      : Tree := Ptr;
  Prev, Temp : Tree; Turn_Left : Boolean;
begin
  while Curr /= null loop
    Prev := Curr;
    if N < Curr.all.Id then
      Curr := Curr.all.Left; Turn_Left := True;
    else
      Curr := Curr.all.Right; Turn_Left := False;
    end if;
  end loop;
  Temp := new Node; Temp.all.Id := N;
  if Prev /= null then
    if Turn_Left then Prev.all.Left := Temp;
    else Prev.all.Right := Temp;
  end if;
  else Ptr := Temp;
  end if;
end Wstaw;
```

© 2024-26 Przemysław Kobylański



Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

- rekurencyjne wstawianie do drzewa uporządkowanego w Adzie:

```
procedure Rek_Wstaw (Ptr : in out Tree; N : Integer)
  is
  begin
    if Ptr = null then
      Ptr := new Node;
      Ptr.all.Id := N; -- wstaw N
    elsif N < Ptr.all.Id then
      Rek_Wstaw (Ptr.all.Left, N); -- idź w lewo
    elsif N > Ptr.all.Id then
      Rek_Wstaw (Ptr.all.Right, N); -- idź w prawo
    end if;
  end Rek_Wstaw;
  ...
  Rek_Wstaw (Root, N); -- wywołanie
```

© 2024-26 Przemysław Kobylański



Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

Ćwiczenie

Napisz w języku Ada procedurę **Usuń**, która **zachowując uporządkowania** usuwa daną wartość z drzewa.

- podczas usuwania węzła drzewo może rozpaść się na dwa kawałki i trzeba je na nowo połączyć zachowując uporządkowanie

© 2024-26 Przemysław Kobylański



Struktury dynamiczne – drzewa

Uporządkowane drzewa binarne

- rekurencyjne usunięcie całego drzewa w Adzie:

```
with Ada.Unchecked_Deallocation;
...
procedure Delete is
  new Ada.Unchecked_Deallocation (Node, Tree);

procedure Rek_Usuń_Drzewo (Ptr : in out Tree) is
begin
  if Ptr /= null then
    Rek_Usuń_Drzewo (Ptr.all.Left);
    Rek_Usuń_Drzewo (Ptr.all.Right);
    Delete (Ptr);
  end if;
end Rek_Usuń_Drzewo;
```

© 2024-26 Przemysław Kobylański



Struktury dynamiczne – drzewa

Trochę nomenklatury

Definicja (Korzeń)

Korzeniem drzewa binarnego jest węzeł tż. wszystkie pozostałe węzły drzewa leżą w jego dwóch poddrzewach.

Definicja (Liść)

Liściem drzewa binarnego nazywamy węzeł, którego oba poddrzewa są pustę.

Definicja (Węzeł wewnętrzny)

Węzłem wewnętrznym drzewa binarnego jest węzeł niebędący liściem.

Definicja (Gałąź)

Gałąź drzewa nazywamy ciąg jego węzłów leżących na ścieżce od korzenia do liścia.

Struktury dynamiczne – drzewa

Trochę nomenklatury

Definicja (Długość gałęzi)

Długością gałęzi jest liczba węzłów na niej leżących.

Definicja (Wysokość)

Wysokością drzewa jest długość jego najdłuższej gałęzi.

Twierdzenie

Drzewo o n węzłach ma wysokość h leżącą w zakresie:

$$\lceil \log_2(n+1) \rceil \leq h \leq n.$$

Twierdzenie

Drzewo o wysokości h ma liczbę węzłów n leżącą w zakresie:

$$h \leq n \leq 2^h - 1.$$

Struktury dynamiczne – drzewa

Trochę nomenklatury

Definicja (Drzewo regularne)

Regularnym drzewem binarnym nazywamy takie, w którym każdy węzeł wewnętrzny ma dwa niepuste poddrzewa.

Twierdzenie

Drzewo regularne o $m \geq 1$ liściach ma $n = 2 \cdot m - 1$ węzłów, w tym $m - 1$ węzłów wewnętrznych.

Wniosek

Każde drzewo regularne ma nieparzystą liczbę węzłów.

Ćwiczenie

Udowodnij trzy twierdzenia o drzewach binarnych.

WYKŁAD 11.

Operacje wejście/wyjścia

Operacje wejścia/wyjścia

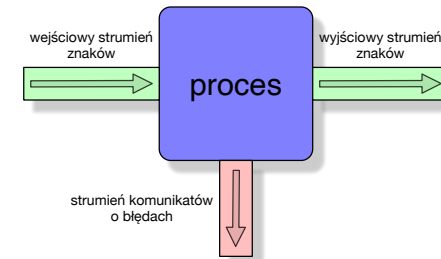
Proces i strumienie znaków

- ▶ na **system operacyjny** składa się oprogramowanie ułatwiające aplikacjom korzystanie z zasobów maszyny (procesor(y), pamięć, urządzenia wejścia/wyjścia)
- ▶ system operacyjny jest **wielozadaniowy** czyli mogą być w nim uruchomione jednocześnie różne programy
- ▶ działający program nazywa się w systemie **procesem**
- ▶ proces posiada trzy strumienie znaków: wejściowy, wyjściowy i strumień błędów
- ▶ **powłoka systemu** drukuje prompt \$ i czyta wpisane po nim **polecenie**
- ▶ powłoka uruchamia proces wykonujący wywołany w poleceniu program
- ▶ jeśli program wymaga **przekierowania plików** do/ze strumieni, to powłoka wykona odpowiednie przekierowania

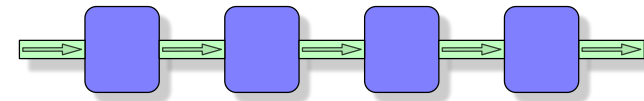
© 2024-26 Przemysław Kobylański

Operacje wejścia/wyjścia

Proces i strumienie znaków



a)



b)

a) Proces i jego strumienie b) Procesy połączone strumieniami

© 2024-26 Przemysław Kobylański

Operacje wejścia/wyjścia

Przekierowanie strumienia

- ▶ za pomocą znaków < i > można przekierować plik do strumienia i strumień do pliku
- ▶ czytanie z klawiatury i drukowanie na ekran:
\$./main
- ▶ czytanie z pliku dane.txt i drukowanie na ekran:
\$./main < dane.txt
- ▶ czytanie z klawiatury i drukowanie do pliku wyniki.txt:
\$./main > wyniki.txt
- ▶ czytanie z pliku dane.txt i drukowanie do pliku wyniki.txt:
\$./main < dane.txt > wyniki.txt

© 2024-26 Przemysław Kobylański

Operacje wejścia/wyjścia

Przetwarzanie potokowe

- ▶ założmy, że mamy trzy następujące programy:

generowanie wypisuje wygenerowane dane na standardowe wyjście

przetwarzanie czyta dane ze standardowego wejścia, przetwarza je i wypisuje wyniki na standardowe wyjście

prezentowanie czyta wyniki ze standardowego wejścia i prezentuje na standardowym wyjściu

- ▶ bez przetwarzania potokowego można uruchomić te programy następująco:
\$./generowanie > dane
\$./przetwarzanie < dane > wyniki
\$./prezentowanie < wyniki
- ▶ zanim zacznie działać przetwarzanie musi zakończyć się generowanie i dane muszą być przechowane na dysku
- ▶ zanim zacznie się prezentowanie musi zakończyć się przetwarzanie i wyniki muszą być przechowane na dysku
- ▶ na dysku pozostają pliki z danymi i wynikami

© 2024-26 Przemysław Kobylański

Operacje wejścia/wyjścia

Przetwarzanie potokowe

- ▶ korzystając z przetwarzania potokowego można przyspieszyć działanie programów:
`$ ./generowanie | ./przetwarzanie | ./prezentowanie`
- ▶ trzy programy zostają uruchomione jednocześnie
- ▶ przetwarzanie rozpoczyna się natychmiast gdy pojawią się pierwsze dane
- ▶ prezentacja rozpocznie się natychmiast gdy pojawią się pierwsze wyniki
- ▶ na dysku nie są tworzone pliki z danymi i wynikami
- ▶ dane między programami przekazywane są szybko poprzez pamięć operacyjną

Operacje wejścia/wyjścia

Operacje na standardowym wejściu i wyjściu

- ▶ dołączenie pliku nagłówkowego
`#include <stdio.h>`
- ▶ czytanie ze standardowego wejścia stdin
 - ▶ funkcja `getchar()`
`int getchar(void) // zwraca kod przeczytanego znaku`
 - ▶ funkcja `scanf()`
`int scanf(format, &zmienna) // zwraca liczbę przeczytanych poprawnie wartości`

Operacje wejścia/wyjścia

Operacje na standardowym wejściu i wyjściu

- ▶ pisanie na standardowe wyjście stdout
 - ▶ funkcja `putchar()`
`int putchar(kod) // wypisanie znaku o podanym kodzie`
 - ▶ funkcja `printf()`
`int printf(format, wyrażenie, ...)`

Operacje wejścia/wyjścia

Operacje na standardowym wejściu i wyjściu

- ▶ książkowy¹¹ przykład:

```
#include <stdio.h>
int main(void) {
    int ch;
    while((ch = getchar()) != EOF) putchar(ch);
    return 0;
}
```
- ▶ warunek sterujący pętlą wykonuje trzy rzeczy:
 1. czyta znak
 2. kod znaku podstawia pod zmienną `ch`
 3. porównuje kod ze stałą `EOF` oznaczającą koniec pliku

¹¹Brian W. Kernighan, Dennis M. Ritchie. **Język ANSI C. Programowanie.** Helion, 2010.

Operacje wejścia/wyjścia

Operacje na plikach

- ▶ dołączenie pliku nagłówkowego

```
#include <stdio.h>
```

w którym zdefiniowano między innymi typ FILE, reprezentujący deskryptor pliku, oraz następujące wskazania typu FILE*:

`stdin` standardowe wejście
`stdout` standardowe wyjście
`stderr` strumień komunikatów o błędach

- ▶ otwieranie pliku

```
FILE* plik;  
fopen(ścieżka, "tryb");
```

gdzie tryb jest jednym z:

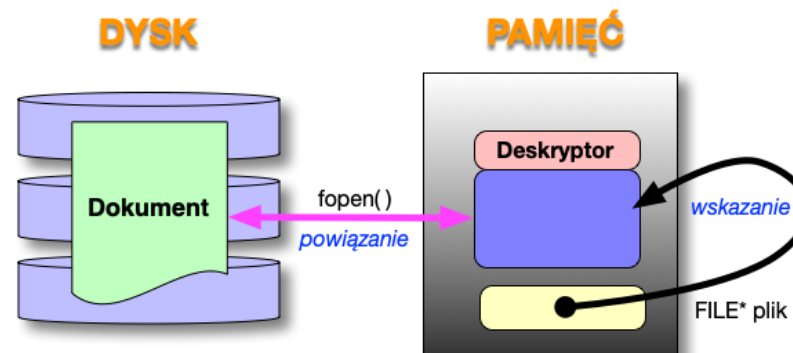
- `r` czytanie (plik musi istnieć)
- `w` pisanie (plik nie musi istnieć)
- `a` dopisywanie na końcu (plik nie musi istnieć)

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Operacje na plikach



Powiązanie deskryptora pliku z dokumentem

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Operacje na plikach

- ▶ czytanie z pliku

- ▶ funkcja `fgetc()`

```
int fgetc(plik) // wczytanie znaku z pliku
```

`fgetc(stdin) ≡ getchar()`

- ▶ funkcja `fscanf()`

```
int fscanf(plik, format, &zmienna) // zwraca  
liczbę przeczytanych poprawnie wartości
```

`fscanf(stdin, format, &zmienna) ≡ scanf(format, &zmienna)`

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Operacje na plikach

- ▶ pisanie do pliku

- ▶ funkcja `fputc()`

```
int fputc(plik, kod) // wypisanie znaku o  
podanym kodzie
```

`fputc(stdout, znak) ≡ putchar(znak)`

- ▶ funkcja `fprintf()`

```
int fprintf(plik, format, wyrażenie, ...)
```

`fprintf(stdout, format, &zmienna) ≡ printf(format, &zmienna)`

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Operacje na plikach

- ▶ opróżnienie bufora dla pliku wyjściowego

- ▶ funkcja fflush()

```
|| int fflush(plik) // zapisanie zmian w pliku  
|| wyjściowym
```

- ▶ zamykanie pliku

- ▶ funkcja fclose()

```
|| int fclose(plik) // zamknięcie pliku
```

jeśli plik był otwarty w trybie **w** albo **a**, to przed jego zamknięciem zostaną zapisane zmiany

```
|| FILE* plik;  
|| fopen(plik, "r");  
|| // przetwarzanie danych  
|| ...  
|| fclose(plik);
```

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Przekazywanie argumentów

- ▶ drukowanie przekazanych do programu argumentów

```
|| // argumenty.c  
||  
|| // Program drukuje argumenty uruchomienia programu.  
  
|| #include <stdio.h>  
  
|| int main(int argc, char *argv[]) {  
||     printf("Liczba argumentów (argc) = %d\n", argc);  
||     printf("Kolejne argumenty o indeksach od 0 do %d:\n",  
||         argc - 1);  
||     for(int i = 0; i < argc; i++)  
||         printf("\targv[%d] = \"%s\"\n", i, argv[i]);  
||     return 0;  
|| }
```

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Przekazywanie argumentów

- ▶ przykładowe uruchomienia:

```
$ ./argumenty
```

```
Liczba argumentów (argc) = 1
```

```
Kolejne argumenty o indeksach od 0 do 0:
```

```
    argv[0] = "./argumenty"
```

```
$ ./argumenty a1 "a2 a3" a4 \
```

```
a5 a6
```

```
Liczba argumentów (argc) = 6
```

```
Kolejne argumenty o indeksach od 0 do 5:
```

```
    argv[0] = "./argumenty"
```

```
    argv[1] = "a1"
```

```
    argv[2] = "a2 a3"
```

```
    argv[3] = "a4"
```

```
    argv[4] = "a5"
```

```
    argv[5] = "a6"
```

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Przykład programu

- ▶ sumowanie liczb ze strumienia

```
|| // sumator.c  
||  
|| #include <stdio.h>  
||  
|| int main(int argc, char *argv[]) {  
||     FILE *fp;  
||     if(argc < 2)  
||         fp = stdin;  
||     else {  
||         fp = fopen(argv[1], "r");  
||         if(fp == NULL) {  
||             printf("Plik %s nie istnieje!\n", argv[1]);  
||             return 1;  
||         }  
||     }  
|| }
```

© 2024-26 Przemysław Kobylański



Operacje wejścia/wyjścia

Przykład programu

```
// ciąg dalszy pliku sumator.c

double suma = 0.0;

for(;;) {
    double wartosc;
    int wynik = fscanf(fp, "%lf", &wartosc);
    if(wynik < 1)
        break;
    suma = suma + wartosc;
}

fclose(fp);

printf("suma = %f\n", suma);
return 0;
}
```

© 2024-26 Przemysław Kobylański

Operacje wejścia/wyjścia

Przykład programu

- ▶ przykłady uruchomienia:

założmy, że w pliku dane.txt znajdują się trzy liczby 10, 20 i 30

```
$ ./sumator
1
2
3
<Ctrl-D>
suma = 6.000000

$ ./sumator zla_nazwa.txt
Plik zla_nazwa.txt nie istnieje!

$ ./sumator dane.txt
suma = 60.000000
```

© 2024-26 Przemysław Kobylański

Operacje wejścia/wyjścia

Ciekawostka



- ▶ w książce pod tytułem **10 PRINT CHR\$(205.5+RND(1))**: **GOTO 10** opisano na ponad 320 stronach jednolinijkowy program w języku BASIC
- ▶ odpowiada mu następujący program w języku C:

```
#include <stdio.h>
#include <stdlib.h> // funkcje srand() i rand()
#include <time.h>    // funkcja time()
int main(void) {
    srand(time(NULL));
    while(1) {
        printf(rand() & 1 ? "\u2571" : "\u2572");
        fflush(stdout);
    }
}
```

- książkę znajdziesz tutaj: <https://10print.org>

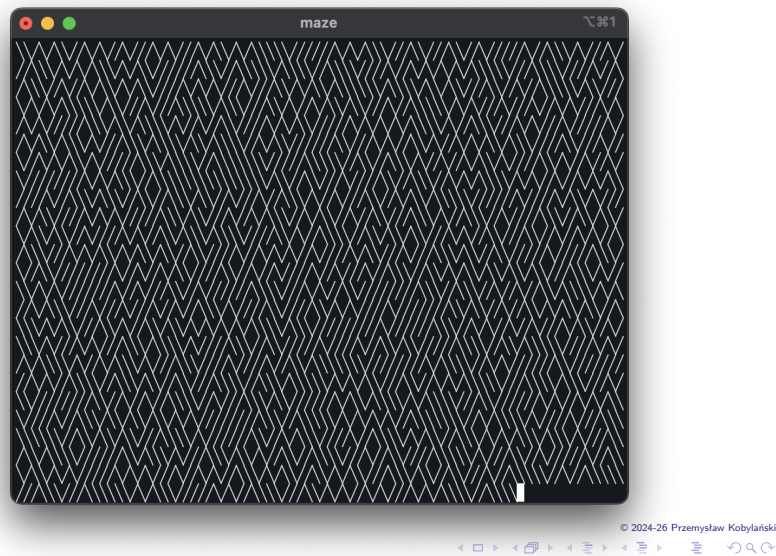
© 2024-26 Przemysław Kobylański

Operacje wejścia/wyjścia

Ciekawostka

- ▶ program drukuje nieskończony labirynt
- ▶ ściany labiryntu składają się z ciągu dwóch znaków / i \
- ▶ znaki wybierane są losowo
- ▶ funkcja `time(NULL)` zwraca liczbę sekund od początku ery systemu UNIX (od godziny 0:00 w dniu 1 stycznia 1970 roku)
- ▶ funkcja `srand(time(NULL))` ustala ziarno generatora liczb pseudolosowych (bez tej funkcji za każdym razem generowany byłby ten sam ciąg liczb)
- ▶ funkcja `rand()` zwraca pseudolosową liczbę całkowitą
- ▶ wartość wyrażenia `rand()\&1` jest albo 0 albo 1 (najmniej znaczący bit) z jednakowym prawdopodobieństwem 1/2
- ▶ wartość wyrażenia `rand()\&1?"\u2571":"\u2572"` jest albo "/" albo "\" z jednakowym prawdopodobieństwem 1/2

© 2024-26 Przemysław Kobylański



WYKŁAD 12.

Rekurencja

Rekurencja

Rekurencja w matematyce

- ▶ rekurencja to definicja odwołująca się sama do siebie
- ▶ definicja taka składa się z:
 1. prostych przypadków, które nie korzystają z rekurencji
 2. zbioru reguł, które redukują złożone przypadki do przypadków prostych
- ▶ przykład definicji ciągu Fibonacciego:

$$f_n = \begin{cases} 0 & \text{gdy } n = 0 \text{ (prosty przypadek)} \\ 1 & \text{gdy } n = 1 \text{ (prosty przypadek)} \\ f_{n-1} + f_{n-2} & \text{gdy } n > 1 \text{ (redukcja)} \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12	...
f_n	0	1	1	2	3	5	8	13	21	34	55	89	144	...

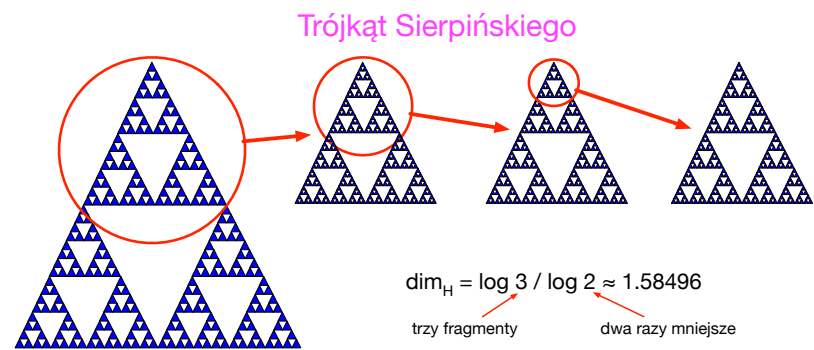
Rekurencja

Rekurencja w matematyce

- ▶ z rekurencją wiążą się ciekawe obiekty geometryczne jakimi są fraktale
- ▶ ich nazwa pochodzi od łac. fractus (złamany, ułamkowy) i odnosi się do niecałkowitego wymiaru Hausdorffa
- ▶ nieformalnie mówiąc, jeśli obiekt ma wymiar z przedziału $(1, 2)$, to jest „grubszy” niż odcinek i „cieńszy” niż kwadrat
- ▶ charakteryzują się one własnością **samopodobieństwa** polegającą na tym, że ich fragmenty są podobne do całości

Rekurencja

Rekurencja w matematyce



© 2024-26 Przemysław Kobylański



Rekurencja

Rekurencja w przyrodzie



Kryształ dendryczny (źródło: Wikipedia)

© 2024-26 Przemysław Kobylański



Rekurencja

Rekurencja w przyrodzie



Płatek śniegu (źródło: The New York Times)

© 2024-26 Przemysław Kobylański



Rekurencja

Rekurencja w przyrodzie



Kalafior Romanesco $\dim_H \approx 2.8$ (źródło: Wikipedia)

© 2024-26 Przemysław Kobylański



Rekurencja

Obrazy rekurencyjne

Definicja (Efekt Droste)

- Obraz ukazujący efekt Droste zawiera mniejszą wersję samego siebie. Ta mniejsza wersja zawiera jeszcze mniejszą wersję w tym samym miejscu i tak dalej.
- Termin został wymyślony przez **Nico Scheepmakera** pod koniec lat 70-tych XX wieku. Nazwa pochodzi od **Droste**, holenderskiej marki gorącej czekolady, na której opakowaniu zamieszczony był wizerunek pielęgniarki trzymającej tacę z filiżanką i pudełkiem kakao tej właśnie marki.

Źródło: https://pl.wikipedia.org/wiki/Efekt_Droste.

Rekurencja

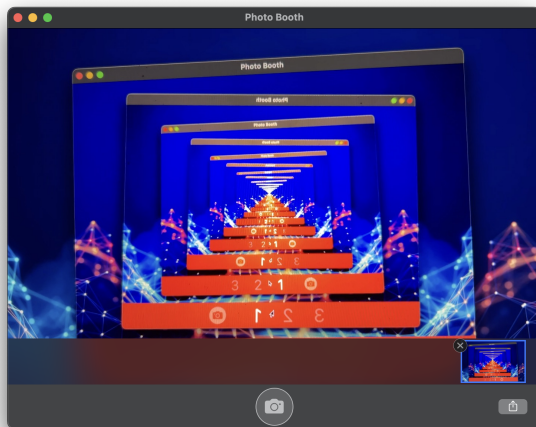
Obrazy rekurencyjne



Opakowanie kakao Droste (źródło: Wikipedia)

Rekurencja

Obrazy rekurencyjne



Efekt umieszczenia kamery przed monitorem (źródło: własne)

Rekurencja

Przykłady funkcji rekurencyjnych

- klasycznym przykładem funkcji definiowanej rekurencyjnie jest silnia:

$$n! = 1 \cdot 2 \cdot \dots \cdot n$$

- jej postać rekurencyjna:

$$n! = \begin{cases} 1 & \text{gdy } n = 0 \\ n \cdot (n - 1)! & \text{gdy } n > 1 \end{cases}$$

```
int silnia(int n) {  
    if(n == 0)  
        return 1;  
    else  
        return n * silnia(n - 1);  
}
```

Rekurencja

Przykłady funkcji rekurencyjnych

- ▶ funkcja obliczająca największy wspólny dzielnik:

$$NWD(a, b) = \begin{cases} a & \text{gdy } b = 0 \\ NWD(b, a \bmod b) & \text{gdy } b > 0 \end{cases}$$

```
def nwd(a, b):  
    if b == 0:  
        return a  
    else:  
        return nwd(b, a % b)
```

© 2024-26 Przemysław Kobylański

Rekurencja

Przykłady funkcji rekurencyjnych

- ▶ w teorii obliczeń korzysta się z funkcji Ackermana zdefiniowanej rekurencyjnie:

$$A(m, n) = \begin{cases} n + 1 & \text{gdy } m = 0 \\ A(m - 1, 1) & \text{gdy } m > 0 \wedge n = 0 \\ A(m - 1, A(m, n - 1)) & \text{gdy } m > 0 \wedge n > 0 \end{cases}$$

- ▶ powyższa funkcja bardzo szybko rośnie:

$$A(4, 2) = 2^{65536} - 3$$

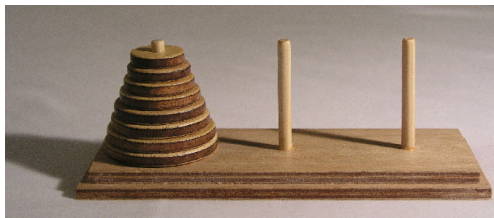
```
function Ackerman (M, N : Natural) return Positive is  
begin  
    if M = 0 then  
        return N + 1;  
    elsif N = 0 then  
        return Ackerman (M - 1, 1);  
    else  
        return Ackerman (M - 1, Ackerman (M, N - 1));  
    end if;  
end Ackerman;
```

© 2024-26 Przemysław Kobylański

Rekurencja

Rozwiązywanie problemów

- ▶ rekurencję można użyć do rozwiązywania problemów
- ▶ złożony problem można często zredukować do prostszego
- ▶ przedstawimy to na przykładzie łamigłówki **Wieża Hanoi** :
 - ▶ w deseczce wbite są trzy patyki *lewy*, *środkowy* i *prawy*
 - ▶ na patyk A nanizanych jest n krążków o coraz większych średnicach
 - ▶ należy przenieść wszystkie krążki z patyka *lewego* na patyk *prawy* pomagając sobie patykiem *środkowym*
 - ▶ za każdym razem przenosi się jeden krążek i nigdy nie kładzie się większy krążek na mniejszym



(źródło: Wikipedia)

© 2024-26 Przemysław Kobylański

Rekurencja

Rozwiązywanie problemów

- ▶ program rozwiązujący łamigłówkę Wieża Hanoi:

```
with Ada.Text_IO; use Ada.Text_IO;  
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;  
procedure Main is  
  
    procedure Hanoi (N : Natural; A, B, C : String) is  
    begin  
        if N > 0 then  
            Hanoi (N - 1, A, C, B);  
            Put_Line ("FROM " & A & " TO " & C);  
            Hanoi (N - 1, B, A, C);  
        end if;  
    end Hanoi;  
  
    N : Integer;  
begin  
    Get (N);  
    Hanoi (N, "lewy", "srodkowy", "prawy");  
end Main;
```

© 2024-26 Przemysław Kobylański

Rekurencja

Rozwiązanie problemów

- ▶ przykład uruchomienia dla $n = 3$:

```
$ alr run
3
FROM lewy TO prawy
FROM lewy TO srodkowy
FROM prawy TO srodkowy
FROM lewy TO prawy
FROM srodkowy TO lewy
FROM srodkowy TO prawy
FROM lewy TO prawy
```

Rekurencja

Rozwiązanie problemów

- ▶ policzmy ile ruchów wykonywanych jest przy przenoszeniu n krążków
- ▶ niech $R(n)$ oznacza liczbę wykonywanych ruchów:

$$R(n) = \begin{cases} 0 & \text{gdy } n = 0 \\ 2 \cdot R(n-1) + 1 & \text{gdy } n > 0 \end{cases}$$

$$\begin{aligned} R(n) &= 2 \cdot R(n-1) + 1 = 2 \cdot (2 \cdot R(n-2) + 1) + 1 \\ &= 4 \cdot R(n-2) + 2 + 1 = 4 \cdot (2 \cdot R(n-3) + 1) + 2 + 1 \\ &= 8 \cdot R(n-3) + 4 + 2 + 1 = \dots = 2^k \cdot R(n-k) + \sum_{i=0}^{k-1} 2^i \\ &= \sum_{i=0}^{n-1} 2^i \quad (\text{dla } k = n) \\ &= 2^n - 1 \end{aligned}$$

Rekurencja

Rozwiązanie problemów

Twierdzenie

Niech n będzie liczbą krążków, wówczas liczba ruchów wykonywanych wynosi $R(n) = 2^n - 1$.

Dowód.

Indukcja po n :

krok 1. $R(0) = 2^0 - 1 = 0$.

krok 2. Niech $n > 0$. Wówczas:

$$\begin{aligned} R(n) &= 2 \cdot R(n-1) + 1 \quad (\text{z def.}) \\ &= 2 \cdot (2^{n-1} - 1) + 1 \quad (\text{z zał. ind.}) \\ &= 2 \cdot 2^{n-1} - 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

Rekurencja

Problemy z rekurencją

- ▶ często rekurencyjny zapis funkcji jest wygodny i czytelny
- ▶ wykonywanie funkcji rekurencyjnej może jednak nastroczać problemów:
 - ▶ lokalne zmienne i parametry funkcji przechowywane są na stosie a ten rośnie z każdym jej wywołaniem
 - ▶ prosta definicja rekurencyjna może wymagać wielu kroków obliczeń
- ▶ problemy te omówimy na przykładzie obliczania ciągu Fibonacciego

```
int f(int n) {
    if(n == 0) return 0;
    else if(n == 1) return 1;
    else.      return f(n - 1) + f(n - 2);
}
```

Rekurencja

Problemy z rekurencją

- ▶ prześledźmy proces obliczenia wartości f_5 :

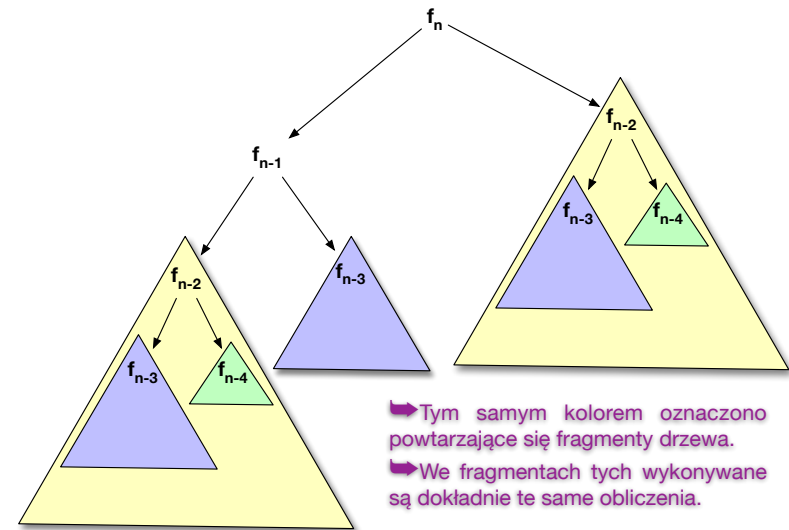
$$\begin{aligned}
 f_5 &= f_4 + f_3 \\
 &= f_3 + f_2 + f_3 \\
 &= f_2 + f_1 + f_2 + f_3 \\
 &= f_1 + f_0 + f_1 + f_2 + f_3 \\
 &= 1 + f_0 + f_1 + f_2 + f_3 \\
 &= 1 + 0 + f_1 + f_2 + f_3 \\
 &= 1 + 0 + 1 + f_2 + f_3 \\
 &= 1 + 0 + 1 + f_1 + f_0 + f_3 \\
 &= 1 + 0 + 1 + 1 + f_0 + f_3 \\
 &= 1 + 0 + 1 + 1 + 0 + f_3 \\
 &= 1 + 0 + 1 + 1 + 0 + f_2 + f_1 \\
 &= 1 + 0 + 1 + 1 + 0 + f_1 + f_0 + f_1 \\
 &= 1 + 0 + 1 + 1 + 0 + 1 + f_0 + f_1 \\
 &= 1 + 0 + 1 + 1 + 0 + 1 + 0 + f_1 \\
 &= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\
 &= 5
 \end{aligned}$$

© 2024-26 Przemysław Kobylański

Rekurencja

Problemy z rekurencją

- ▶ drzewo obliczeń wartości f_n :



iw Kobylański

Rekurencja

Problemy z rekurencją

- ▶ policzymy ile operacji dodawania wykonuje się podczas obliczania wartości f_n
- ▶ niech $S(n)$ będzie liczbą operacji dodawania:

$$S(n) = \begin{cases} 0 & \text{gdy } n = 0 \vee n = 1 \\ 1 + S(n-1) + S(n-2) & \text{gdy } n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10	11	...
f_n	0	1	1	2	3	5	8	13	21	34	55	89	...
$S(n)$	0	0	1	2	4	7	12	20	33	54	88	143	...

Hipoteza

$$S(n) = \sum_{i=0}^{n-1} f_i = f_{n+1} - 1.$$

© 2024-26 Przemysław Kobylański

Rekurencja

Problemy z rekurencją

Twierdzenie

Niech $S(n)$ będzie liczbą operacji dodawania podczas obliczania wartości f_n . Wówczas:

$$S(n) = f_{n+1} - 1.$$

Dowód.

Indukcja po n :

krok 1. $S(0) = f_1 - 1 = 1 - 0 = 0$, $S(1) = f_2 - 1 = 1 - 1 = 0$.

krok 2. Niech $n > 1$. Wówczas:

$$\begin{aligned}
 S(n) &= 1 + S(n-1) + S(n-2) \text{ (z def.)} \\
 &= 1 + (f_n - 1) + (f_{n-1} - 1) \text{ (z zał. ind.)} \\
 &= f_n + f_{n-1} - 1 = f_{n+1} - 1 \text{ (z def.)}
 \end{aligned}$$

© 2024-26 Przemysław Kobylański

Rekurencja

Problemy z rekurencją

- ▶ jak szybko rośnie wartość $S(n)$?
- ▶ na drugim semestrze, podczas kursu **Matematyka dyskretna**, poznacie funkcje tworzące i nauczycie się wyprowadzać za ich pomocą jawny wzór na n -ty wyraz ciągu Fibonacciego:

$$f_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

$$\frac{1-\sqrt{5}}{2} \approx -0.618034$$

$$\lim_{n \rightarrow \infty} \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n = 0$$

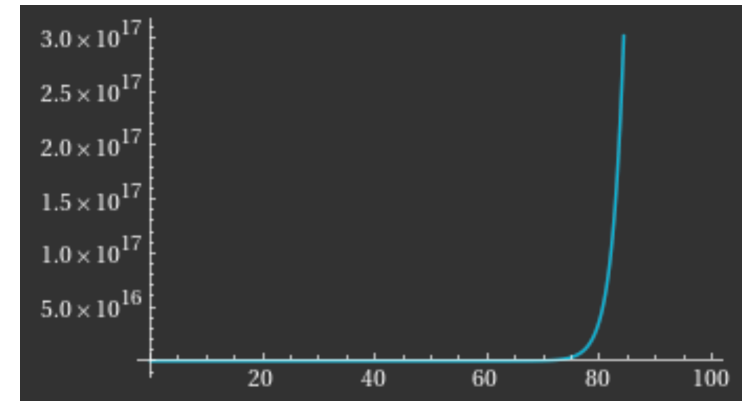
$$f_n \sim \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n$$

$$S(n) \sim \frac{1+\sqrt{5}}{2\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n$$

© 2024-26 Przemysław Kobylański

Rekurencja

Problemy z rekurencją



Wykres $S(n)$ (źródło: WolframAlpha.com)

- ▶ jak policzyć wartość f_n wykonując mniejszą liczbę operacji?
- ▶ zastosować iterację zamiast rekurencji

© 2024-26 Przemysław Kobylański

Rekurencja

Problemy z rekurencją

- ▶ iteracyjna funkcja w języku Python:

```
def f(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1
        for i in range(n - 1):
            a, b = b, a + b
        return b

print(f(1000)) # około 4.35e208
```

4346655768693745643568852767504062580256466051737178040248
1729089536555417949051890403879840079255169295922593080322
6347752096896232398733224711616429964409065331879382989696
49928516003704476137795166849228875

© 2024-26 Przemysław Kobylański

Rekurencja

Postać ogonowa rekurencji

- ▶ rozpatrzmy funkcję obliczającą $n!$

```
int silnia(int n) {
    if(n == 0)
        return 1;
    else
        return n * silnia(n - 1);
}
```

- ▶ po powrocie z rekurencyjnego wywołania $\text{silnia}(n-1)$ wynik jest przemnażany przez wartość zmiennej n
- ▶ zmienna n musi być zachowana na stosie by po powrocie z rekurencyjnego wywołania dostępna była jej wartość przed wywołaniem
- ▶ funkcja rekurencyjna, która ma więcej niż jedno wywołanie rekurencyjne w swojej treści albo ma jedno wywołanie ale po powrocie z niego wykonywane są jeszcze obliczenia **nie jest w postaci rekurencji ogonowej**

© 2024-26 Przemysław Kobylański

Rekurencja

Postać ogonowa rekurencji

- ▶ prześledźmy proces obliczenia wartości *silnia*(5):

$$\begin{aligned} \text{silnia}(5) &= 5 * \text{silnia}(4) \\ &= 5 * 4 * \text{silnia}(3) \\ &= 5 * 4 * 3 * \text{silnia}(2) \\ &= 5 * 4 * 3 * 2 * \text{silnia}(1) \\ &= 5 * 4 * 3 * 2 * 1 * \text{silnia}(0) \\ &= 5 * 4 * 3 * 2 * 1 * 1 \\ &= 5 * 4 * 3 * 2 * 1 \\ &= 5 * 4 * 3 * 2 \\ &= 5 * 4 * 6 \\ &= 5 * 24 \\ &= 120 \end{aligned}$$

Rekurencja

Postać ogonowa rekurencji

- ▶ możemy postarać się przekształcić funkcję do **postaci rekurencji ogonowej**, w której po jedynym wywołaniu rekurencyjnym nie są wykonywane już żadne obliczenia:

```
int silnia_ogonowa(int n, int akumulator) {
    if(n == 0)
        return akumulator;
    else
        return silnia_ogonowa(n - 1, n * akumulator);
}

int silnia(int n) {
    return silnia_ogonowa(n, 1);
}
```

Rekurencja

Postać ogonowa rekurencji

- ▶ prześledźmy proces obliczenia wartości *silnia*(5) z użyciem rekursji ogonowej:

$$\begin{aligned} \text{silnia}(5) &= \text{silnia_ogonowa}(5, 1) \\ &= \text{silnia_ogonowa}(4, 5) \\ &= \text{silnia_ogonowa}(3, 20) \\ &= \text{silnia_ogonowa}(2, 60) \\ &= \text{silnia_ogonowa}(1, 120) \\ &= \text{silnia_ogonowa}(0, 120) \\ &= 120 \end{aligned}$$

- ▶ jak widać, proces obliczania bardzo się uprościł
- ▶ zwracane wartości nie są już przez nic przemnażane
- ▶ wszystkie obliczenia (odejmowanie i mnożenie) odbywają się podczas obliczania wartości parametrów rzeczywistych dla kolejnego wywołania rekurencyjnego

Rekurencja

Postać ogonowa rekurencji

- ▶ funkcję w postaci ogonowej rekurencji bardzo łatwo przekształcić na iterację:

```
int silnia(int n) {
    int akumulator = 1;
    while(n != 0) {
        akumulator = akumulator * n;
        n = n - 1;
    }
    return akumulator;
}
```

- ▶ skorzystano tu z równoważności następujących kodów:

```
int F1() {
    if(W)
        S1
    else {
        S2
        F1();
    }
}

int F2() {
    while(!W)
        S2
}
```


Rekurencja

Ciekawostka



- ▶ gdy funkcja zawiera dwa wywołania rekurencyjne nie można jej przekształcić do postaci rekurencji ogonowej
- ▶ można jednak postarać się zastąpić jedno wywołanie rekurencyjne iteracją, w której znajduje się drugie wywołanie rekurencyjne
- ▶ pokażemy to na przykładzie funkcji obliczającej ciąg Fibonacciego
- ▶ kompilator maksymalnie optymalizując przekład (opcja -O3) pozbywa się jednego z dwóch wywołań rekurencyjnych
- ▶ korzysta przy tym z następującej własności ciągu Fibonacciego:

$$f_n = 1 + \sum_{i=0}^{n-2} f_i = \sum_{i=1}^{n \div 2} f_{n-2i+1} + (n \bmod 2)$$

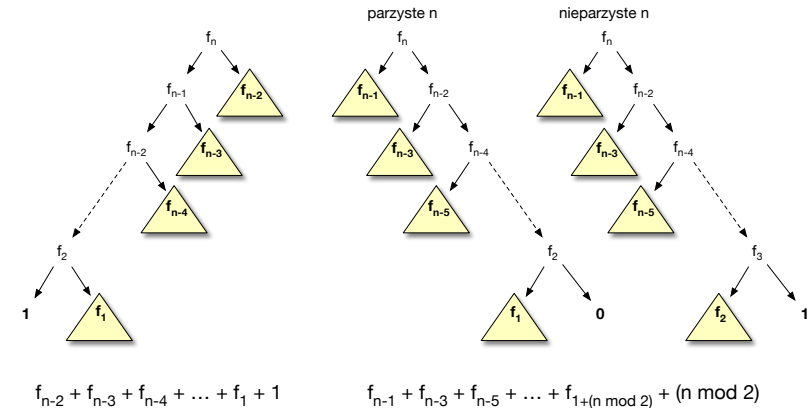
© 2024-26 Przemysław Kobylański



Rekurencja

Ciekawostka

- ▶ co kryje się za tą własnością?



© 2024-26 Przemysław Kobylański



Rekurencja

Ciekawostka

Twierdzenie

Niech $n > 0$. Wówczas: $f_0 = 0, f_n = 1 + \sum_{i=0}^{n-2} f_i$.

Dowód.

Indukcja po n :

krok 1. $f_0 = 0, f_1 = 1 + 0$.

krok 2. Niech $n > 1$. Wówczas:

$$\begin{aligned} f_n &= f_{n-1} + f_{n-2} \text{ (z def.)} \\ &= 1 + \sum_{i=0}^{n-3} f_i + f_{n-2} \text{ (z zał. ind.)} \\ &= 1 + \sum_{i=0}^{n-2} f_i \end{aligned}$$

© 2024-26 Przemysław Kobylański



Rekurencja

Ciekawostka

Twierdzenie

Niech $n \geq 0$. Wówczas: $f_n = \sum_{i=1}^{n \div 2} f_{n-2i+1} + (n \bmod 2)$.

Dowód.

Indukcja po n :

krok 1. $f_0 = 0 + 0, f_1 = 0 + 1$.

krok 2. Niech $n > 1$. Wówczas:

$$\begin{aligned} f_n &= f_{n-1} + f_{n-2} \text{ (z def.)} \\ &= f_{n-1} + \sum_{i=1}^{(n-2) \div 2} f_{n-2-2i+1} + (n-2 \bmod 2) \text{ (z zał. ind.)} \\ &= f_{n-1} + \sum_{i=2}^{n \div 2} f_{n-2i+1} + (n \bmod 2) = \sum_{i=1}^{n \div 2} f_{n-2i+1} + (n \bmod 2) \end{aligned}$$

© 2024-26 Przemysław Kobylański



Rekurencja

Ciekawostka

- ▶ kompilując poniższą funkcję z opcjami -O3 -S:

```
int f(int n) {
    if(n == 0) return 0;
    else if(n == 1) return 1;
    else return f(n - 1) + f(n - 2);
}
```

- otrzymuje się przekład równoważny następującej funkcji:

```
int f(int n) {
    if(n < 2) return n;
    int a = 0;
    for(; n > 1; n-=2) a += f(n - 1);
    return a + n;
}
```

- ▶ funkcja wylicza sumę:

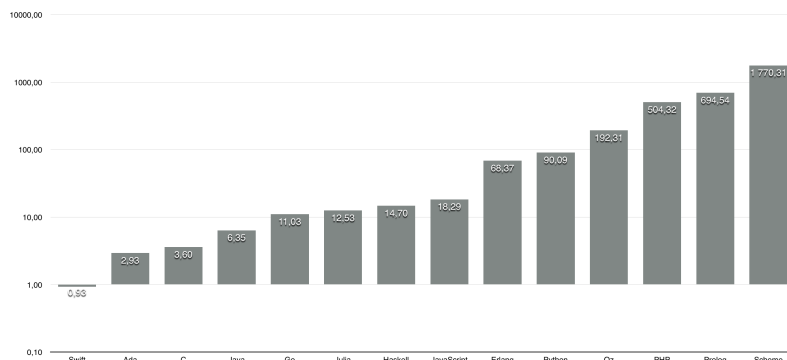
$$f_{n-1} + f_{n-3} + f_{n-5} + \cdots + f_{1+(n \bmod 2)} + (n \bmod 2) = f_n$$

© 2024-26 Przemysław Kobylański

Rekurencja

Ciekawostka

- ▶ na stronie <https://github.com/przemko/fib45> można znaleźć porównanie efektywności obliczania ciągu Fibonacciego w wybranych czternastu językach programowania



Czas obliczania f_{45} w sekundach na Intel Core i5 2.6GHz (źródło: własne)

Rekurencja

Ciekawostka

- ▶ przykład kompilatorem Apple CLANG 16.0.0 na procesor M3:

[illegible]

© 2024-26 Przemysław Kobylański

WYKŁAD 13.

Zasada dziel i zwyciężaj

Zasada dziel i zwyciężaj

Notacja asymptotyczna

- ▶ na kursie **Analiza matematyczna I** poznaliście notację asymptotyczną
- ▶ tutaj tylko przypomnimy podstawowe informacje o niej

Definicja (Notacja $O(\cdot)$)

Niech $f, g : \mathbb{N} \mapsto \mathbb{R}^+$. Mówimy, że $f(n) = O(g(n))$ jeśli

$$(\exists c > 0)(\exists n_0 > 0)(\forall n \geq n_0) f(n) < c \cdot g(n).$$

Gdy $f(n) = O(g(n))$, to mówimy, że $g(n)$ jest **asymptotycznym ograniczeniem górnym dla $f(n)$** .

Definicja (Notacja $\Theta(\cdot)$)

Niech $f, g : \mathbb{N} \mapsto \mathbb{R}^+$. Jeśli $f(n) = O(g(n))$ i $g(n) = O(f(n))$, to mówimy, że $f(n) = \Theta(g(n))$.

Zasada dziel i zwyciężaj

Notacja asymptotyczna

Twierdzenie

Niech $f, g : \mathbb{N} \mapsto \mathbb{R}^+$. Załóżmy, że istnieje granica $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ i jest skończona. Wówczas

1. Jeśli $0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$, to $f(n) = O(g(n))$.
2. Jeśli $0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$, to $f(n) = \Theta(g(n))$.

Dowód.

Niech $c = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$. Załóżmy, że $0 \leq c < \infty$. Wówczas dla dostatecznie dużych n , $0 \leq \frac{f(n)}{g(n)} < c + 1$. Zatem dla dostatecznie dużych n , $f(n) < (c + 1) \cdot g(n)$, więc $f(n) = O(g(n))$.

Jeśli ponadto $c > 0$, to $0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{1}{c} < \infty$, więc z poprzedniego punktu wynika, że $g(n) = O(f(n))$, czyli $f(n) = \Theta(g(n))$. □

Zasada dziel i zwyciężaj

Notacja asymptotyczna

Definicja (Notacja $o(\cdot)$)

Niech $f, g : \mathbb{N} \mapsto \mathbb{R}^+$. Mówimy, że $f(n) = o(g(n))$ jeśli

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

- ▶ czasami zamiast pisać $f(n) = o(g(n))$ będziemy stosować zapis $f(n) \ll g(n)$
- ▶ łatwo sprawdzić, że

$$\dots \ll \sqrt[4]{n} \ll \sqrt[3]{n} \ll \sqrt{n} \ll n \ll n^2 \ll n^3 \ll n^4 \ll \dots$$

Zasada dziel i zwyciężaj

Notacja asymptotyczna

- ▶ pokażemy, że funkcja $\log_2 n$ rośnie wolniej od pierwiastka dowolnego stopnia
- ▶ skorzystamy z następującego twierdzenia:

Twierdzenie (de l'Hospitala)

Załóżmy, że f i g są funkcjami różniczkowalnymi tż. $\lim_{x \rightarrow \infty} f(x) = \infty$ oraz $\lim_{x \rightarrow \infty} g(x) = \infty$. Wówczas

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}.$$

Zasada dziel i zwyciężaj

Notacja asymptotyczna

Twierdzenie

$$(\forall_{\alpha > 0}) \log_2 n = o(n^\alpha).$$

Dowód.

Niech $\alpha > 0$. Funkcje $f(x) = \log_2 x$ oraz $g(x) = x^\alpha$ spełniają założenia twierdzenia de l'Hospitala. Zatem

$$\lim_{x \rightarrow \infty} \frac{\log_2 x}{x^\alpha} = \lim_{x \rightarrow \infty} \frac{\frac{1}{x \cdot \ln 2}}{\alpha \cdot x^{\alpha-1}} = \lim_{x \rightarrow \infty} \frac{1}{\alpha \cdot \ln 2 \cdot x^\alpha} = 0.$$

□

Zasada dziel i zwyciężaj

Schemat metody

- ▶ zasada **dziel i zwyciężaj** (ang. divide and conquer) jest bardzo efektywną metodą projektowania algorytmów
- ▶ opiera się ona na rekurencji
- ▶ dzieli ona rozwiązywany problem na pewną liczbę prostszych podproblemów
- ▶ następnie rozwiązuje, zgodnie z tą zasadą, każdy z podproblemów
- ▶ na koniec przeprowadzana jest synteza rozwiązania problemu z rozwiązań podproblemów

Zasada dziel i zwyciężaj

Schemat metody

▶ zasadę dziel i zwyciężaj zapiszemy w postaci pseudokodu:

```
function RozwiążZłożonyProblem ( $P$ ):  
if  $P$  jest prostym problemem then  
    return RozwiązanieProstegoProblemu( $P$ )  
else  
    podziel  $P$  na prostsze podproblemy  $P_1, P_2, \dots, P_a$  ▷ analiza  
    for  $i \leftarrow 1 \dots a$  do  
         $R_i \leftarrow$  RozwiążZłożonyProblem( $P_i$ ) ▷ rekurencja  
    end for  
    return Synteza( $R_1, R_2, \dots, R_a$ ) ▷ synteza  
end if
```

Zasada dziel i zwyciężaj

Złożoność obliczeniowa

- ▶ niech n będzie rozmiarem problemu P
- ▶ założmy, że problem P jest dzielony na a podproblemów b razy mniejszego rozmiaru
- ▶ niech funkcja $T(n)$ wyraża liczbę operacji jakie wykonywane są podczas rozwiązywania problemu P zgodnie z zasadą dziel i zwyciężaj:

$$T(n) = \begin{cases} \Theta(1) & \text{dla } n = 1 \\ a \cdot T(\frac{n}{b}) + O(n) & \text{dla } n > 1 \end{cases}$$

tn. rozwiązanie prostego problemu wykonuje się w czasie stałym a podział i synteza odbywa się w czasie liniowym

- ▶ rząd funkcji $T(n)$ zależy od relacji między liczbami a i b

Zasada dziel i zwyciężaj

Złożoność obliczeniowa

Twierdzenie (O rekurencji uniwersalnej)

Niech $a \geq 1$ i $b > 1$ oraz

$$T(n) = \begin{cases} \Theta(1) & \text{dla } n = 1 \\ a \cdot T(\frac{n}{b}) + O(n) & \text{dla } n > 1 \end{cases}$$

Wówczas:

1. Jeśli $a < b$, to $T(n) = O(n)$.
2. Jeśli $a = b$, to $T(n) = O(n \log n)$.
3. Jeśli $a > b$, to $T(n) = O(n^{\log_b a})$.

© 2024-26 Przemysław Kobylański

Zasada dziel i zwyciężaj

Przykład: najmniejszy i największy element

- ▶ dla danej n -elementowej tablicy liczb całkowitych chcemy znaleźć jej element najmniejszy i największy
- ▶ wykonując $n - 1$ porównań można znaleźć element najmniejszy:

```
min = t[0];
for(int i = 1; i < n; i++)
    if(t[i] < min)
        min = t[i];
```

- ▶ analogicznie wykonując $n - 1$ porównań można znaleźć element największy
- ▶ zatem wykonując $2 \cdot n - 2$ porównań można znaleźć element najmniejszy i największy
- ▶ pokażemy jak stosując zasadę dziel i zwyciężaj znaleźć element najmniejszy i największy mniejszą liczbą porównań

© 2024-26 Przemysław Kobylański

Zasada dziel i zwyciężaj

Przykład: najmniejszy i największy element

```
void min_max(int n, int t[n], int* min, int* max) {
    if(n == 1)
        *min = *max = t[0];
    else if(n == 2)
        if(t[0] < t[1]) {
            *min = t[0]; *max = t[1];
        }
        else {
            *min = t[1]; *max = t[0];
        }
    else {
        int n2 = n / 2;
        int min1, min2, max1, max2;
        min_max(n2, t, &min1, &max1);
        min_max(n-n2, t+n2, &min2, &max2);
        *min = (min1 < min2) ? min1 : min2;
        *max = (max1 > max2) ? max1 : max2;
    }
}
```

© 2024-26 Przemysław Kobylański

Zasada dziel i zwyciężaj

Przykład: najmniejszy i największy element

- ▶ niech $C(n)$ będzie liczbą porównań wykonywanych w funkcji min_max dla n -elementowej tablicy:

$$C(n) = \begin{cases} 0 & \text{dla } n = 1 \\ 1 & \text{dla } n = 2 \\ 2 \cdot C(\frac{n}{2}) + 2 & \text{dla } n > 2 \end{cases}$$

- ▶ łatwo sprawdzić¹², że dla n będących potęgami liczby 2, rozwiązaniem powyżej definicji rekurencyjnej jest funkcja:

$$C(n) = \frac{3}{2} \cdot n - 2$$

- ▶ jak widać funkcja min_max wykonuje o 25% mniej porównań niż metoda naiwna

¹² $C(n) = 2 \cdot C(\frac{n}{2}) + 2 = 2 \cdot (\frac{3}{2} \cdot \frac{n}{2} - 2) + 2 = \frac{6}{4} \cdot n - 4 + 2 = \frac{3}{2} \cdot n - 2$

© 2024-26 Przemysław Kobylański

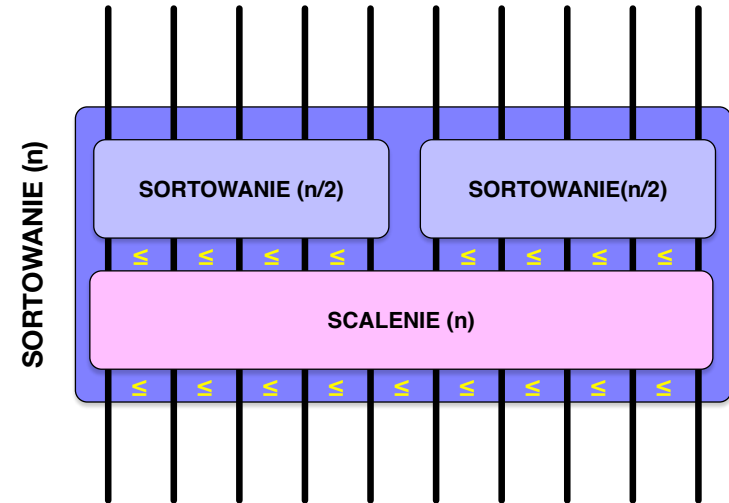
Zasada dziel i zwyciężaj

Przykład: sortowanie przez scalanie

- ▶ sortowanie przez scalanie działa zgodnie z zasadą dziel i zwyciężaj
- ▶ tablicę jednoelementową uznaje za posortowaną
- ▶ tablicę co najmniej dwuelementową dzieli na dwie połówki
- ▶ następnie rekurencyjnie sortuje każdą z połówek
- ▶ na koniec scala dwie posortowane połówki tablicy w posortowaną całość

Zasada dziel i zwyciężaj

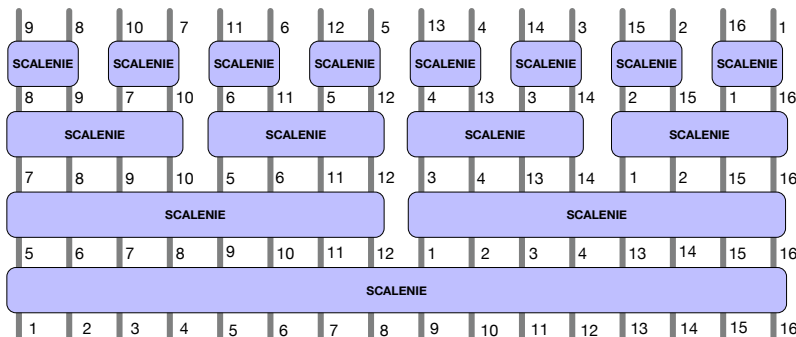
Przykład: sortowanie przez scalanie



Zasada dziel i zwyciężaj

Przykład: sortowanie przez scalanie

- ▶ rekurencyjna procedura sortowania przez scalanie zapewnia odpowiedni podział tablicy i kolejność scalania coraz to dłuższych fragmentów:



Zasada dziel i zwyciężaj

Przykład: sortowanie przez scalanie

- ▶ scalenie dwóch kawałków tablicy:

```
void scal(int n, int tab[n], int robocza[n],
         int pierwszy, int srodek, int ostatni) {
    int i = pierwszy, j = srodek + 1, k = pierwszy;
    while(k <= ostatni) {
        if(i <= srodek && j <= ostatni)
            if(tab[i] <= tab[j])
                robocza[k++] = tab[i++];
            else
                robocza[k++] = tab[j++];
        else
            if(i <= srodek)
                robocza[k++] = tab[i++];
            else
                robocza[k++] = tab[j++];
    }
    for(int i = pierwszy; i <= ostatni; i++)
        tab[i] = robocza[i];
}
```

Zasada dziel i zwyciężaj

Przykład: sortowanie przez scalanie

► sortowanie przez scalanie:

```
void reksort(int n, int tab[n], int robocza[n],
            int pierwszy, int ostatni) {
    if(pierwszy < ostatni) {
        int srodek = pierwszy + (ostatni - pierwszy) / 2;
        reksort(n, tab, robocza, pierwszy, srodek);
        reksort(n, tab, robocza, srodek + 1, ostatni);
        scal(n, tab, robocza, pierwszy, srodek, ostatni);
    }
}

void mergesort(int n, int tab[n]) {
    int robocza[n];
    reksort(n, tab, robocza, 0, n - 1);
}
```

Zasada dziel i zwyciężaj

Przykład: sortowanie przez scalanie

Twierdzenie

Funkcja mergesort dla n -elementowej tablicy wykonuje $O(n \log n)$ porównań i podstawień.

Dowód.

Niech $C(n)$ oznacza liczbę porównań i podstawień wykonywanych przez funkcję mergesort. Podział na dwie połówki tablicy odbywa się w czasie stałym a scalenie w czasie liniowym $O(n)$. Załóżmy, że n jest potęgą liczby 2. Wówczas:

$$C(n) = \begin{cases} 0 & \text{gdy } n = 1 \\ 2 \cdot C(\frac{n}{2}) + O(n) & \text{gdy } n > 1 \end{cases}$$

Z twierdzenia o rekurencji uniwersalnej $C(n) = O(n \log n)$. $\square$

Zasada dziel i zwyciężaj

Przykład: operacje na dużych liczbach

- zanim omówimy algorytm szybkiego mnożenia najpierw przedstawimy „ręczne” wyliczanie sumy i iloczynu dużych liczb
- liczby będziemy przechowywać w postaci tablic, których elementami są cyfry liczby:

```
N : constant Integer := 100;
type Liczba is array (0 .. N-1) of Natural;
type Duza_Liczba is array (0 .. 2*N-1) of Natural;
```

Zasada dziel i zwyciężaj

Przykład: operacje na dużych liczbach

► dodawanie liczb:

```
procedure Suma (A, B : Liczba; C : out Liczba) is
    Przeniesienie : Natural := 0;
    Sum : Natural;
begin
    for I in C'Range loop
        Sum := Przeniesienie + A(I) + B(I);
        C(I) := Sum rem 10;
        Przeniesienie := Sum / 10;
    end loop;
end Suma;
```

- powyższe dodawanie dwóch n -cyfrowych liczb ma złożoność $O(n)$

Zasada dziel i zwyciężaj

Przykład: operacje na dużych liczbach

- ▶ iloczyn liczb:

```

procedure Iloczyn (A, B : Liczba;
                   C : out Długa_Liczba) is
    Przeniesienie : Natural;
    Sum : Natural;
begin
    C := (others => 0);
    for J in B'Range loop
        Przeniesienie := 0;
        for I in A'Range loop
            Sum := Przeniesienie + C(I+J) + A(I) * B(J);
            C (I+J) := Sum rem 10;
            Przeniesienie := Sum / 10;
        end loop;
    end loop;
end Iloczyn;

```

- ▶ powyższe mnożenie dwóch n -cyfrowych liczb ma złożoność $O(n^2)$
- © 2024-26 Przemysław

© 2024-26 Przemysław Kobylański

◀ ◻ ▶ ◀ ☰ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

Zasada dziel i zwyciężaj

Przykład: operacje na dużych liczbach

```
Licznik : Liczba := (others => 0);
Temp    : Liczba;
Silnia  : Liczba := (1, others => 0);
Wynik   : Dluga_Liczba;
M : Integer;

...
Get (M);
for I in 1 .. M loop
    Suma (Licznik, (1, others => 0), Temp);
    Licznik := Temp;
    Iloczyn (Silnia, Licznik, Wynik);
    Silnia := Liczba (Wynik (Silnia'Range));
end loop;
for I in reverse Silnia'Range loop
    Put (Silnia(I), Width => 1);
end loop;
```

```
$ alr run
```

70

197857166996989179607278372168909873645893814254642585755536286462800958278984

5319680000000000000000

© 2024-26 Przemysław Kobylański

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ▶ ≡ ▶ ≡ ▶ ↺ 🔍 ↻

Zasada dziel i zwyciężaj

Przykład: operacje na dużych liczbach

- ▶ działanie procedury Suma i Iloczyn można w prosty sposób przyspieszyć jednak nie obniżając ich odpowiednio liniowej i kwadratowej złożoności
- ▶ dla każdej tablicy można pamiętać ile cyfr w tablicy jest znaczących
- ▶ wówczas pętle będą wykonywać się po znaczących cyfrach z pominięciem zer wypełniających tablicę
- ▶ można zmienić podstawę systemu liczbowego z 10 na 10000 pamiętając by drukując cyfrę w uzupełniać ją z lewej strony zerami do długości czterech cyfr

© 2024-26 Przemysław Kobylański

◀ ◻ ▶ ◀ 📄 ▶ ◀ 📊 ▶ ◀ 📈 ▶ ◀ 📉 ▶ ◀ 🔄 ▶

Zasada dziel i zwyciężaj

Przykład: algorytm szybkiego mnożenia Karacuby

- ▶ przedstawimy teraz szybki algorytm mnożenia dużych liczb opracowany przez Anatolija Karacubę w roku 1962
- ▶ założymy, że chcemy przemnożyć dwie duże liczby x i y zapisane w systemie dziesiętnym
- ▶ założymy, że obie liczby mają $m = 2n$ cyfr (jeśli nie, to uzupełniamy z lewej strony zerami), wówczas

$$x = 10^m x_1 + x_2, \quad y = 10^m y_1 + y_2,$$

gdzie x_1, x_2, y_1, y_2 są m -cyfrowymi liczbami

- wtedy

$$x \cdot y = 10^{2m}x_1y_1 + 10^m(x_1y_2 + x_2y_1) + x_2y_v$$

- ▶ musimy umieć szybko wyliczyć liczby $x_1 y_1$, $x_1 y_2 + x_2 y_1$ i $x_2 y_2$

© 2024-26 Przemysław Kobylański

◀ ◻ ▶ ◀ 📄 ▶ ◀ 📑 ▶ ◀ 📑 ▶ 🔍 ↺ ↻

Zasada dziel i zwyciężaj

Przykład: algorytm szybkiego mnożenia Karacuby

- ▶ Karacuba zauważył, że liczby x_1y_1 , $x_1y_2 + x_2y_1$ i x_2y_2 można wyliczyć wykonując trzy zamiast cztery mnożenia
- ▶ niech $A = x_1y_1$, $B = x_2y_2$, $C = (x_1 + x_2)(y_1 + y_2)$
- ▶ liczbę $x_1y_2 + x_2y_1$ można wyliczyć jednym dodawaniem i jednym odejmowaniem:

$$C - (A + B) = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2 - x_1y_1 - x_2y_2 = x_1y_2 + x_2y_1$$

- ▶ ten sam trik można zastosować rekurencyjnie do mnożenia liczb m -cyfrowych

Zasada dziel i zwyciężaj

Przykład: algorytm szybkiego mnożenia Karacuby

- ▶ niech $T(n)$ oznacza czas potrzebny na przemnożenie dwóch liczb n -cyfrowych metodą Karacuby:

$$T(n) = 3 \cdot T(n/2) + O(n)$$

Twierdzenie

Algorytm Karacuby działa w czasie $O(n^{\log_2 3})$.

Dowód.

Z twierdzenia o rekurencji uniwersalnej.



Zasada dziel i zwyciężaj

Przykład: algorytm szybkiego mnożenia Karacuby

- ▶ z tego, że $\log_2 3 \approx 1.585$ wynika, że $n^{\log_2 3} \ll n^2$, czyli algorytm Karacuby jest szybszy niż „ręczne” mnożenie w czasie $O(n^2)$
- ▶ najszybszy znany algorytm mnożenia dwóch długich liczb, to algorytm Schönhagena-Strassena z roku 1972
- ▶ działa on w czasie $O(n \log n \log \log n)$
- ▶ wykorzystuje zaawansowane narzędzia matematyczne jak szybka transformacja Fouriera (FFT)

WYKŁAD 14.

Przegląd z nawrotami

Przegląd z nawrotami

Sformułowanie problemu

- ▶ dany jest niepusty i skończony zbiór D nazywany **dziedziną**
- ▶ niech $\Omega = D^n$ będzie **przestrzenią rozwiązań**
- ▶ na zbiorze rozwiązań określona jest funkcja $C : \Omega \mapsto \{true, false\}$
- ▶ logiczny warunek C nazywać będziemy **ograniczeniami**
- ▶ ograniczenia wyznaczają **zbiór rozwiązań dopuszczalnych**:

$$S = \{ \langle x_0, x_1, \dots, x_{n-1} \rangle \in \Omega : C(x_0, x_1, \dots, x_{n-1}) \}$$

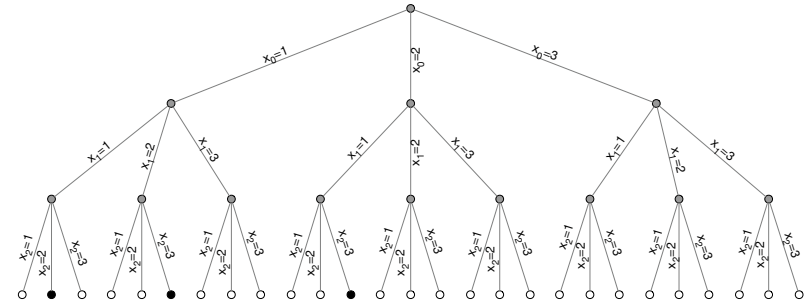
Definicja (Problem spełnienia ograniczeń)

Problem polega na znalezieniu jednego lub wszystkich elementów zbioru rozwiązań dopuszczalnych S .

Przegląd z nawrotami

Drzewo poszukiwań

- ▶ przykład dla $n = 3$, $D = \{1, 2, 3\}$ i ograniczenia $x_0 + x_1 = x_2$:



- ▶ czarnym kolorem oznaczono liście odpowiadające rozwiązaniom spełniającym ograniczenie $x_0 + x_1 = x_2$

Przegląd z nawrotami

Pełen przegląd

- ▶ poniższy program w C znajduje rozwiązania dopuszczalne wykonując **pełen przegląd**:

```
#include <stdio.h>
int main(void) {
    int x[3];
    for(x[0] = 1; x[0] <= 3; x[0]++)
        for(x[1] = 1; x[1] <= 3; x[1]++)
            for(x[2] = 1; x[2] <= 3; x[2]++)
                if(x[0] + x[1] == x[2])
                    printf("%d %d %d\n", x[0], x[1], x[2]);
    return 0;
}
```

- ▶ dla $n = 3$ program zawiera trzy zagnieżdżone w sobie pętle
- ▶ każda pętla przebiega całą dziedzinę wartości
- ▶ a gdyby n było zadany parametrem?
- ▶ należy użyć rekurencję

Przegląd z nawrotami

Pełen przegląd

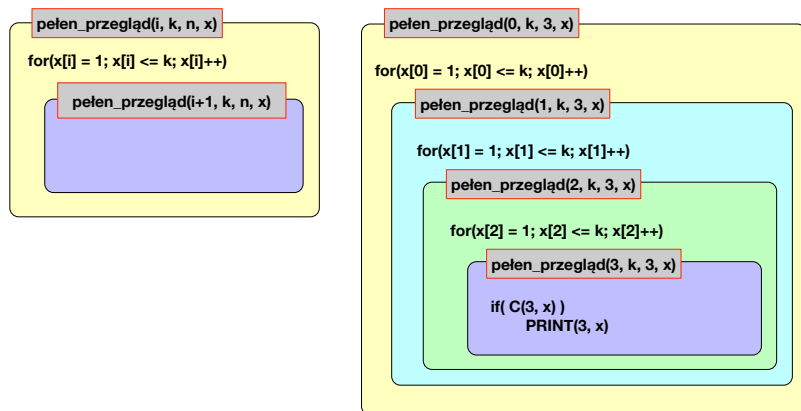
- ▶ niech $C(n, x)$ sprawdza czy wartości $x[0], \dots, x[n-1]$ spełniają ograniczenie
- ▶ niech $PRINT(n, x)$ drukuje znalezione rozwiązanie
- ▶ pełen przegląd dla $D = \{1, 2, \dots, k\}$ i $\Omega = D^n$:

```
void pełen_przegląd(int i, int k, int n, int x[n])
// ustalenie wartości x[i], x[i+1], ..., x[n-1]
{
    if(i == n)
        if(C(n, x))
            PRINT(n, x);
    else
        for(x[i] = 1; x[i] <= k; x[i]++)
            pełen_przegląd(i+1, k, n, x);
}
...
int n, k;
...
int x[n];
...
pełen_przegląd(0, k, n, x);
```

Przegląd z nawrotami

Pełen przegląd

- ▶ rekurencja umożliwia dowolne zagłębianie pętli:



Przegląd z nawrotami

Pełen przegląd

- ▶ przykład: generowanie ciągów n -bitowych w C:

```
#include <stdio.h>

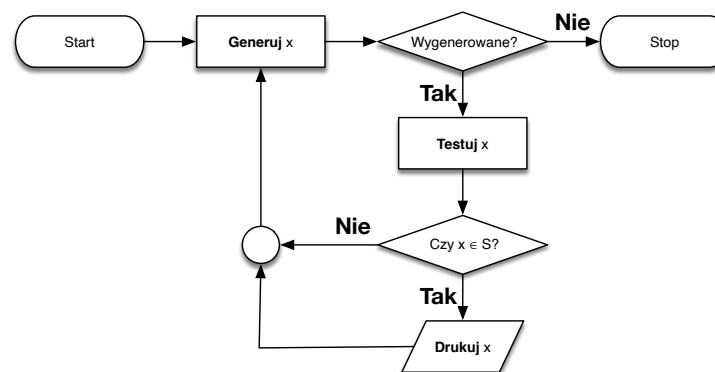
void bity(int i, int n, int b[n]) {
    if(i == n) {
        for(int j = 0; j < n; j++)
            printf("%d", b[j]);
        printf("\n");
    }
    else
        for(b[i] = 0; b[i] < 2; b[i]++)
            bity(i + 1, n, b);
}

int main(void) {
    int n;
    scanf("%d", &n);
    int b[n];
    bity(0, n, b);
    return 0;
}
```

Przegląd z nawrotami

Pełen przegląd

- ▶ funkcja pełen_przegląd(i, k, n, x) działa bardzo nieefektywnie
- ▶ najpierw generuje całą zawartość tablicy x[n] a dopiero na końcu sprawdza czy jej elementy spełniają ograniczenie
- ▶ takie poszukiwanie rozwiązania nazywa się **generowaniem i testowaniem**



Przegląd z nawrotami

Pełen przegląd

- ▶ przykład: generowanie ciągów n -bitowych w Pythonie z użyciem współprogramów:

```
def bit():
    yield 0
    yield 1

def bity(n):
    if n == 0:
        yield []
    else:
        for x in bity(n - 1):
            for y in bit():
                yield x + [y] # konkatenacja

n = int(input())
for x in bity(n):
    print(x)
```

Przegląd z nawrotami

Pełen przegląd

- ▶ przykład: generowanie ciągów n -bitowych w Adzie:

```
type Wektor is array (Integer range <>) of Integer;
procedure Bity (I : Integer; B : in out Wektor) is
begin
    if I > B'Last then
        for X of B loop
            Put (X'Image);
        end loop;
        New_Line;
    else
        for X in 0 .. 1 loop
            B(I) := X;
            Bity (I + 1, B);
        end loop;
    end if;
end Bity;
T : Wektor (-2 .. 1); -- T(-2), T(-1), T(0), T(1)
...
Bity (T'First, T); -- wydrukuje 16 ciągów
```

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Częściowy przegląd

- ▶ niech warunek MOŻLIWE(i , x), dla $i \leq n$, jest spełniony gdy istnieją takie wartości $x[i]$, $x[i+2]$, ..., $x[n-1]$, że zachodzi warunek $C(x[0], x[1], \dots, x[n-1])$
- ▶ zatem warunek MOŻLIWE ocenia węzeł drzewa poszukiwań i przewiduje czy w poddrzewie zakorzenionym w tym węźle znajduje się liść z rozwiązaniem dopuszczalnym
- ▶ inaczej mówiąc, sprawdza czy częściowe rozwiązanie $x[0]$, $x[1]$, ..., $x[i-1]$ można rozszerzyć do pełnego rozwiązania dopuszczalnego $x[0]$, $x[1]$, ..., $x[n-1]$
- ▶ warunek MOŻLIWE udziela niepewnej odpowiedzi, tj. gdy jest spełniony, to **może** da się rozszerzyć częściowe rozwiązanie do dopuszczalnego a w przeciwnym przypadku, **na pewno** się nie da rozszerzyć
- ▶ zakładamy, że dla pełnego rozwiązania warunek MOŻLIWE się nie myli:

$$\text{MOŻLIWE}(n, x) \equiv C(x)$$

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Częściowy przegląd

- ▶ częściowy przegląd z wykorzystaniem warunku MOŻLIWE:

```
1 void częściowy_przegląd(int i, int k, int n, x[n])
2 {
3     if(i == n)
4         PRINT(n, x);
5     else
6         for(x[i] = 1; x[i] <= k; x[i]++)
7             if(MOŻLIWE(i + 1, x))
8                 częściowy_przegląd(i + 1, k, n, x);
9 }
```

- ▶ jeśli częściowe rozwiązanie w wierszu 7. nie spełnia warunku MOŻLIWE, to nie próbuje się rozszerzyć go do pełnego rozwiązania
- ▶ zostaje zatem **odcięte** całe poddrzewo poszukiwań zakorzenione w bieżącym węźle

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Częściowy przegląd

- ▶ najczęściej trudno jest podać warunek MOŻLIWE, który w miarę rozsądnie ocenia węzeł bez przeglądu całego drzewa w nim zakorzenionego
- ▶ sprawdzenie tego warunku to trochę jak przewidywanie przyszłości (ocena rozwiązania przy nie znanych wszystkich jego składowych)
- ▶ dużo łatwiej jest zdefiniować warunek NIEMOŻLIWE(i , x), który jest spełniony gdy częściowe rozwiązanie $x[0]$, $x[1]$, ..., $x[i-1]$ narusza ograniczenia i **na pewno** nie może być rozszerzone do pełnego rozwiązania dopuszczalnego
- ▶ sprawdzanie warunku analizuje jedynie ustalone wartości składające się na częściowe rozwiązanie
- ▶ wówczas za warunek MOŻLIWE(i , x) można przyjąć negację warunku NIEMOŻLIWE(i , x)

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Częściowy przegląd

- ▶ częściowy przegląd z wykorzystaniem warunku NIEMOŻLIWE:

```
1 void częściowy_przegląd(int i, int k, int n, x[n])
2 {
3     if(i == n)
4         PRINT(n, x);
5     else
6         for(x[i] = 1; x[i] <= k; x[i]++)
7             if(!NIEMOŻLIWE(i + 1, x))
8                 częściowy_przegląd(i + 1, k, n, x);
9 }
```

- ▶ jeśli częściowe rozwiązanie w wierszu 7. nie narusza ograniczenia, to próbuje się rozszerzyć go do pełnego rozwiązania
- ▶ gdy częściowe rozwiązanie spełnia warunek NIEMOŻLIWE, to zostaje **odcięte** całe poddrzewo poszukiwań zakorzenione w bieżącym węźle

Przegląd z nawrotami

Częściowy przegląd

- ▶ przykład: generowanie kombinacji n z k w C:

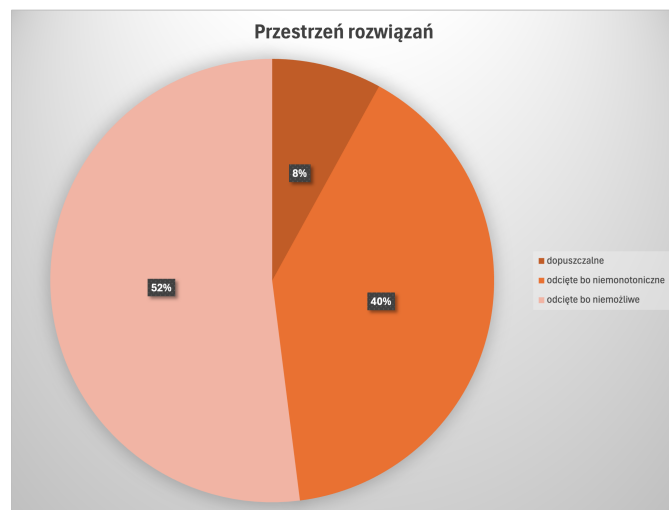
```
void kombinacje(int i, int k, int n, int x[n]) {
    if(i == n) {
        for(int j = 0; j < n; j++)
            printf("%d ", x[j]);
        printf("\n");
    }
    else
        for(x[i] = (i==0?1:x[i-1]+1); x[i] <= k; x[i]++)
            if(n - 1 - i <= k - x[i])
                kombinacje(i + 1, k, n, x);
}
```

- ▶ wyrażenie $i==0?1:x[i-1]+1$ gwarantuje, że wartości wpisywane w tablicę x są rosnące
- ▶ warunek $n-1-i \leq k-x[i]$ sprawdza czy pozostałych $k-x[i]$ wartości wystarczy do wypełnienia $x[i+1]$, $x[i+2]$, ..., $x[n-1]$

Przegląd z nawrotami

Częściowy przegląd

- ▶ drzewo poszukiwań dla kombinacji 3 z 5 ma 125 liści:



Przegląd z nawrotami

Problem n -hetmanów

Definicja (Problem n -hetmanów)

Ustawić n hetmanów na szachownicy $n \times n$ tak aby żadne dwa nie biły się.



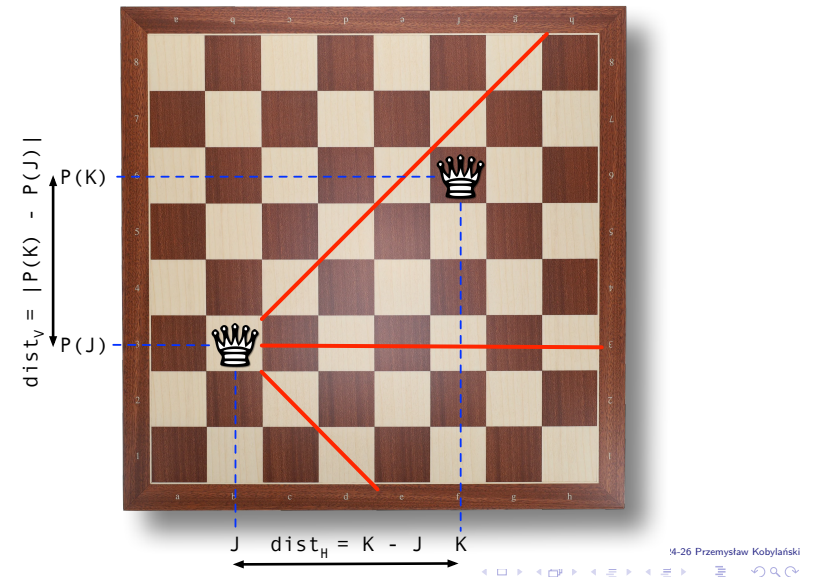
Przegląd z nawrotami

Problem n -hetmanów

- ▶ dziedzina $D = \{1, 2, \dots, n\}$
- ▶ przestrzeń rozwiązań $\Omega = D^n$
- ▶ krotka $\langle p_1, p_2, \dots, p_n \rangle \in \Omega$ opisuje rozwiązanie, w którym hetman stojący w i -tej kolumnie znajduje się w p_i -tym wierszu
- ▶ ograniczenia $C_1 \wedge C_2 \wedge C_3$:
 - ▶ $C_1 \equiv$ żadne dwa hetmany nie biją się w pionie
 - ▶ $C_2 \equiv$ żadne dwa hetmany nie biją się w poziomie
 - ▶ $C_3 \equiv$ żadne dwa hetmany nie biją się po ukosie
- ▶ reprezentacja rozwiązania gwarantuje spełnienie C_1
- ▶ aby spełnić C_2 musi dla dowolnych $1 \leq j < k \leq n$ zachodzić $p_j \neq p_k$, zatem krotka jest permutacją
- ▶ aby spełnić C_3 musi dla dowolnych $1 \leq j < k \leq n$ zachodzić $|p_k - p_j| \neq k - j$

Przegląd z nawrotami

Problem n -hetmanów



Przegląd z nawrotami

Problem n -hetmanów

- ▶ program w języku Ada
- ▶ procedura drukująca linię poziomą:

```
procedure Linia (N : Integer) is
begin
  for I in 1 .. N loop
    Put ("+- - -");
  end loop;
  Put_Line ("+" );
end Linia;
```

Przegląd z nawrotami

Problem n -hetmanów

- ▶ procedura drukująca jeden wiersz planszy:

```
procedure Wiersz (M, N, K : Integer) is
begin
  for J in 1 .. 2 loop
    for I in M .. K - 1 loop
      Put ("| ");
    end loop;
    Put ("|####");
    for I in K + 1 .. N loop
      Put ("| ");
    end loop;
    Put_Line ("|");
  end loop;
end Wiersz;
```

Przegląd z nawrotami

Problem *n*-hetmanów

- deklaracja typu i drukowanie planszy obróconej o 90°:

```
type Permutacja is
  array (Integer range <>) of Integer;

procedure Drukuj (P : Permutacja) is
begin
  New_Line;
  for K of P loop
    Linia (P'Length);
    Wiersz (P'First, P'Last, K);
  end loop;
  Linia (P'Length);
  New_Line;
end Drukuj;
```

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem *n*-hetmanów

```
procedure Rozmieść(K:Integer; P:in out Permutacja) is
  J : Integer;
begin
  if K > P'Last then
    Drukuj (P);
  else
    for I in P'Range loop
      if (for all J in P'First..K-1 => P(J)/=I) then
        J := P'First;
        while J<K and then abs(I-P(J)) /= K-J loop
          J := J + 1;
        end loop;
        if J = K then
          P(K) := I;
          Rozmieść (K + 1, P);
        end if;
      end if;
    end loop;
  end if;
end Rozmieść;
```

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem *n*-hetmanów

- deklaracja tablicy i wywołanie wstawiania hetmanów:

```
T : Permutacja (1 .. 8);
...
Rozmieść (T'First, T);
```

- dla *n* = 8 przestrzeń rozwiązań ma 8⁸ = 16 777 216 elementów
- tylko 92 rozwiązania są dopuszczalne
- wśród rozwiązań dopuszczalnych można wskazać 12, z których wszystkie pozostałe otrzymuje się przez obroty i symetrie

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem *n*-hetmanów

- pierwsze z 92 rozwiązań problemu 8-hetmanów:

#####							
#####							
				#####			
				#####			
							#####
							#####
					#####		
					#####		
		#####					
		#####					
						#####	
						#####	
	#####						
	#####						
			#####				
			#####				

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem n -hetmanów

- funkcje w Pythonie generujące rozwiązania:

```
def bezpieczny(j, p):
    for k in range(len(p)):
        if abs(j - p[k]) == k + 1:
            return False
    return True

def rozmieść(i, n):
    if i == n:
        yield []
    else:
        for p in rozmieść(i + 1, n):
            for j in range(1, n + 1):
                if not j in p:           # ograniczenie C2
                    if bezpieczny(j, p): # ograniczenie C3
                        yield [j] + p
```

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem n -hetmanów

- wywołanie generatora rozwiązań:

```
def hetmany(n):
    c = 0
    for x in rozmieść(0, n):
        c = c + 1
        print(c, x)

n = int(input())
hetmany(n)
```

```
$ python3 hetmany.py
5
1 [4, 2, 5, 3, 1]
2 [3, 5, 2, 4, 1]
3 [5, 3, 1, 4, 2]
4 [4, 1, 3, 5, 2]
5 [5, 2, 4, 1, 3]
6 [1, 4, 2, 5, 3]
7 [2, 5, 3, 1, 4]
8 [1, 3, 5, 2, 4]
9 [3, 1, 4, 2, 5]
10 [2, 4, 1, 3, 5]
```

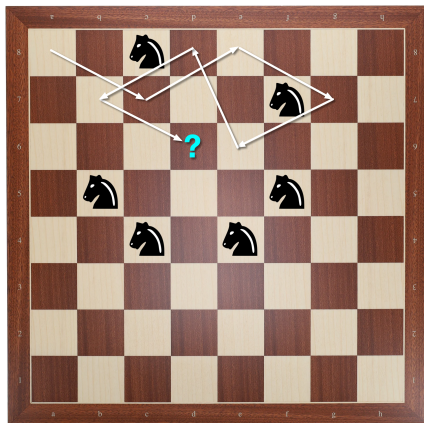
© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem skoczka szachowego

Definicja (Problem skoczka szachowego)

Przejdź planszę szachową ruchem skoczka szachowego odwiedzając każde jej pole dokładnie jeden raz.



© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem skoczka szachowego

- skoczek z pozycji (X, Y) może przejść na pozycję $(X1, Y1)$, przy czym dopuszczalne przejścia zamieszczono w tabeli poniżej:

DX	DY	X1	Y1
-2	-1	$X - 2$	$Y - 1$
-2	+1	$X - 2$	$Y + 1$
-1	-2	$X - 1$	$Y - 2$
-1	+2	$X - 1$	$Y + 2$
+1	-2	$X + 1$	$Y - 2$
+1	+2	$X + 1$	$Y + 2$
+2	-1	$X + 2$	$Y - 1$
+2	+1	$X + 2$	$Y + 1$

- dopuszczalne przesunięcia DX i DY są z zakresu $-2 \dots 2$ i spełniają równanie $\text{abs}(DX) + \text{abs}(DY) = 3$

© 2024-26 Przemysław Kobylański

Przegląd z nawrotami

Problem skoczka szachowego

- ▶ rozwiązanie reprezentować będziemy w Adzie w postaci dwuwymiarowej tablicy o wartościach całkowitych
- ```
type Plansza is
 array (Integer range <>, Integer range <>) of
 Natural;
```
- ▶ początkowo wszystkie elementy tablicy są równe 0
  - ▶ poruszając się po planszy będziemy w elementach tablicy, odpowiadających odwiedzanym kolejno polom szachownicy, wpisywać kolejne liczby 1, 2, 3, ...
  - ▶ zanim wpisze się kolejną liczbę do elementu tablicy sprawdzamy czy jest tam 0, co oznacza, że to pole nie było jeszcze odwiedzone
  - ▶ w przypadku wybrania złej drogi nie uda się obejść wszystkich pól planszy, w takiej sytuacji następuje nawrót
  - ▶ wykonując nawrót trzeba usunąć zapisaną liczbę, wrócić na poprzednio odwiedzone pole i wybrać kolejne dopuszczalne przejście spośród ośmiu możliwych

© 2024-26 Przemysław Kobylański

## Przegląd z nawrotami

### Problem skoczka szachowego

```
function Przejdź (X, Y, N : Integer; P : in out Plansza) return Boolean is
 X1, Y1 : Integer;
begin
 P (X, Y) := N;
 if N = P'Length (1) * P'Length (2) then
 return True;
 end if;
 for DX in -2 .. 2 loop
 for DY in -2 .. 2 loop
 if abs (DX) + abs (DY) = 3 then
 X1 := X + DX;
 Y1 := Y + DY;
 if X1 in P'Range (1) and then Y1 in P'Range (2) and then
 P (X1, Y1) = 0 and then Przejdź (X1, Y1, N + 1, P)
 then
 return True;
 end if;
 end if;
 end loop;
 end loop;
 P (X, Y) := 0;
 return False;
end Przejdź;
```

© 2024-26 Przemysław Kobylański

## Przegląd z nawrotami

### Problem skoczka szachowego

- ▶ deklaracja tablicy i wywołanie poszukiwania rozwiązania:

```
P : Plansza (1 .. 8, 1 .. 8) :=
 (others => (others => 0));
...
if Przejdź (P'First (1), P'First (2), 1, P) then
 for I in P'Range (1) loop
 for J in P'Range (2) loop
 Put (P (I, J), Width => 3);
 end loop;
 New_Line;
 end loop;
else
 Put_Line ("brak rozwiązania");
end if;
```

© 2024-26 Przemysław Kobylański

## Przegląd z nawrotami

### Problem skoczka szachowego

- ▶ znalezione rozwiązanie:

```
$ alr run
 1 12 9 6 3 14 17 20
 10 7 2 13 18 21 4 15
 31 28 11 8 5 16 19 22
 64 25 32 29 36 23 48 45
 33 30 27 24 49 46 37 58
 26 63 52 35 40 57 44 47
 53 34 61 50 55 42 59 38
 62 51 54 41 60 39 56 43
```

© 2024-26 Przemysław Kobylański



## Przegląd z nawrotami

Problem skoczka szachowego

- ▶ przedstawiona funkcja `Przejdź` poszukuje rozwiązania dla dowolnych prostokątnych plansz rozmiaru  $m \times n$
- ▶ plansze większego rozmiaru niż  $8 \times 8$  wymagają jednak długiego czasu poszukiwania
- ▶ czas ten można w istotnym stopniu zmniejszyć rozpoczynając od rogu planszy i stosując odpowiednią **heurystykę** wyboru kolejności sprawdzanych kierunków ruchu
- ▶ jedna z takich heurystyk jest następująca:  
*Dla każdego z pól na jakie możesz przejść oblicz ile jest dopuszczalnych sposobów jego opuszczenia. Wybieraj kierunki ruchu w rosnącej kolejności wyliczonych wartości.*
- ▶ powyższa heurystyka będzie starała się obchodzić pola planszy począwszy od brzegu ku środkowi

## Przegląd z nawrotami

Programowanie ograniczeń

- ▶ jedną z metodologii programowania jest **programowanie deklaratywne**
- ▶ program jest opisem problemu
- ▶ do przyspieszenia poszukiwania rozwiązania problemu opracowano **programowanie ograniczeń**
- ▶ definiuje się w nim dziedzinę wartości jakie mogą przyjmować zmienne
- ▶ wyraża się w nim ograniczenia jakie wartości te muszą spełniać
- ▶ dostarcza ono wiele heurystyk jak przeglądać drzewo poszukiwań aby przyspieszyć poszukiwania

## Przegląd z nawrotami

Programowanie ograniczeń

- ▶ wybrane narzędzia dostarczające programowanie ograniczeń:

| narzędzie         | język programowania |
|-------------------|---------------------|
| IBM ILOG Solver   | C++                 |
| JaCoP             | Java                |
| B-Prolog          | Prolog              |
| Ciao Prolog       | Prolog              |
| ECLiPSe           | Prolog              |
| GNU-Prolog        | Prolog              |
| IF/Prolog         | Prolog              |
| SICStus Prolog    | Prolog              |
| SWI-Prolog        | Prolog              |
| Yap               | Prolog              |
| Emma              | Python              |
| python-constraint | Python              |

## Przegląd z nawrotami

Programowanie ograniczeń

- ▶ przykład w języku **Prolog**<sup>13</sup>:

permutacją  $n$ -elementową jest lista długości  $n$ , złożona z liczb z zakresu od 1 do  $n$ , której elementy są parami różne

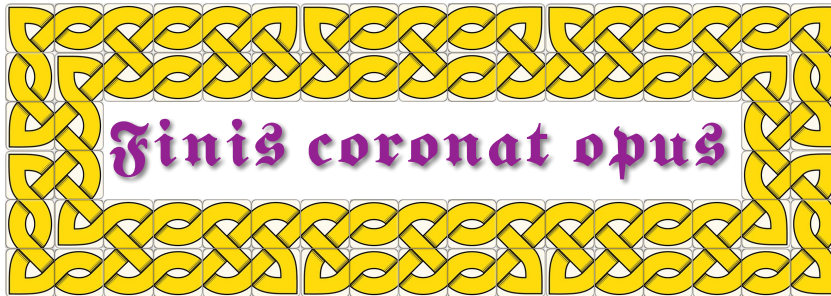
```
:- use_module(library(clpfd)).
permutacja1(N, X) :- length(X, N), X ins 1..N,
/* gen and test */ label(X), all_different(X).
permutacja2(N, X) :- length(X, N), X ins 1..N,
/* constr and gen */ all_different(X), label(X).
```

```
?- time(permutacja1(10, X)).
% 14,172,372,581 inferences, 357.286 CPU in 357.642 seconds (100% CPU, 39666714 Lips)
X = [1, 2, 3, 4, 5, 6, 7, 8, 9|...] .
```

```
?- time(permutacja2(10, X)).
% 3,189 inferences, 0.001 CPU in 0.001 seconds (96% CPU, 3060461 Lips)
X = [1, 2, 3, 4, 5, 6, 7, 8, 9|...] .
```

- ▶ zmiana kolejności warunków `label` i `all_different` przyspieszyło poszukiwania ponad 4 miliony razy

<sup>13</sup>Wersja SWI-Prolog <https://www.swi-prolog.org>.



## Co dalej?

- ▶ języki programowania obiektowego poznasz na obowiązkowym kursie **Kurs programowania** (II semestr)
- ▶ z językiem C spotkasz się na obowiązkowym kursie **Systemy operacyjne** (III semestr)
- ▶ transformację Fouriera poznasz na kursie **Statystyka i analiza danych** (IV semestr)
- ▶ o algorytmice będziesz się uczył na obowiązkowym kursie **Algorytmy i struktury danych** (IV semestr) oraz kursach wybieralnych **Algorytmy metaheurystyczne**, **Algorytmy optymalizacji dyskretnej**, **Wykład monograficzny - algorytmika**
- ▶ o komputerowej reprezentacji liczb i jej wpływie na obliczone wyniki dowiesz się na kursie obowiązkowym **Obliczenia naukowe** (V semestr)
- ▶ z językiem Ada możesz się spotkać na kursach wybieralnych **Programowanie współbieżne** i **Kurs wybranego języka programowania**

## Co dalej?

- ▶ z językiem SPARK możesz spotkać się na wybieralnym kursie **Niezawodne systemy informatyczne**
- ▶ z językiem Python możesz się spotkać na kursie wybieralnym **Metody sztucznej inteligencji**
- ▶ o maszynie Turinga i innych teoretycznych modelach obliczeń możesz dowiedzieć się na wybieralnym kursie **Teoretyczne podstawy informatyki**
- ▶ różnych paradygmatów programowania możesz nauczyć się na wybieralnych kursach **Języki i paradygmaty programowania**, **Programowanie funkcyjne**, **Programowanie w Logice**, **Programowanie współbieżne**
- ▶ język Prolog możesz spotkać na kursie wybieralnym **Programowanie w logice** a programowania ograniczeń możesz nauczyć się na studiach drugiego stopnia na kursie **Programowanie ograniczeń**