PARADYGMATY I JĘZYKI PROGRAMOWANIA

Podprogramy.

Treść

- Wstęp
- Podstawy
- Problemy projektowania podprogramów
- Lokalne środowisko wywołań
- Metody przekazywania parametrów
- Podprogramy jako parametry
- Niejawne wywołania podprogramów
- Podprogramy przeciążone
- Podprogramy ogólne (generic)
- Problemy projektowania funkcji
- Przeciążanie operatorów
- Domknięcia (closures)
- Współprogramy (coroutines)

Wstęp

- Dwa typy abstrakcji programistycznych
  - abstrakcje procesów
    - zauważone w początkach programowania
    - będziemy omawiać na dzisiejszym wykładzie
  - abstrakcje danych
    - pojawiają się w latach 1980

Podstawy podprogramów

- Podprogram posiada jeden punkt wejścia
- W czasie wykonywania podprogramu wykonywanie programu wywołującego jest zawieszone
- Po ukończeniu wykonywania się podprogramu sterowanie powraca zawsze do programu wywołującego

Definicje

- *Definicja podprogramu* obejmuje opis wywołania i działanie abstrakcji podprogramu
  - Python: definicja jest jednocześnie wykonywalna; w innych językach nie jest
  - Ruby: definicje funkcji podawane są wewnątrz lub na zewnątrz definicji klas. Zewnętrzne funkcje są metodami klasy **Object**. Można je wywoływać tak, jak funkcje, bez podawania obiektu.
  - Lua: funkcje anonimowe
- *Wywołanie podprogramu* jest poleceniem jego wykonania
- *Nagłówek podprogramu* jest początkiem jego definicji: nazwa, rodzaj podprogramu, parametry formalne

Definicje cd.

- *Sygnatura podprogramu* (profil) to liczba, kolejność i typ parametrów formalnych
- *Protokół podprogramu* to jego sygnatura oraz, w przypadku funkcji, typ zwracanej wartości
- *Deklaracje* funkcji w C, C++ noszą nazwę *prototypów*
- *Deklaracja podprogramu* dostarcza tylko protokołu (nie podprogramu)
- *Parametrem formalnym* jest zmienna wymieniona w nagłówku podprogramu
- *Parametr aktualny* reprezentuje wartość lub adres użyte w instrukcji wywołania podprogramu

Wywoływnie podprogramów

- Inicjalizacja
  - Czynności wstępne
  - Prolog
- Wykonanie
- Finalizacja
  - Epilog
  - Czynności końcowe

Inicjalizacja

- Czynności wstępne
  - zapamietanie na stosie potrzebnych rejestrów (używanych w podprogramie)
  - Obliczenie wartości parametrów aktualnych i umieszczenie ich na stosie
  - Organizacja miejsca na wyniki
  - Umieszczenie na stosie innych danych
  - Umieszczenie na stosie adresu ostatniej instrukcji; skok do podprogramu
- Prolog
  - Alokacje ramki przez podprogram
  - Alokacja rejestrów potrzebnych programowi wywołującemu

Finalizacja

- Epilog
  - Podprogram umieszcza w odpowiednich miejscach obliczone wyniki
  - Odtwarza się ze stosu umieszczone wcześniej rejestry
  - Zwalniana jest ramka stosu, odsłaniając adres powrotu
  - Wywołuje się instrukcję powrotu, która po pobraniu adresu powrotu ze stosu, wznawia pracę programu wywołującego
- Czynności kończące
  - Obliczone parametry trafiają na swoje miejsca i zwalnia się odpowiadający obszar stosu
  - Odtwarzane są rejestry umieszczone na stosie w fazie inicjalizacji

Odpowiedniość parametrów: formalne – aktualne

- Odpowiedniość pozycyjna
  - wiązanie parametrów aktualnych z formalnymi odbywa się następująco: pierwszy aktualny odpowiada pierwszemu formalnemu, drugi – drugiemu itd.
  - sposób bezpieczny i efektywny
- Odpowiedniość poprzez słowa klucze
  - podawana jest nazwa parametru formalnego, której ma być przypisany parametr aktualny
  - wygodne, bo nie musimy dbać o kolejność parametrów
  - źle, bo użytkownik musi znać nazwy parametrów formalnych

Wartości opcjonalne parametrów

- W wielu językach programowania (C++, Ada, Python, Ruby, PHP, Fortran90) parametry formalne mogą mieć z góry zadane, opcjonalne wartości (o ile nie zostana podane parametry aktualne)
  - w C++ są one podawane jako ostatnie (przekazywanie pozycyjne bez kluczy)
- Zmienna liczba parametrów
  - Ruby: parametry aktualne są przekazywane w tablicy asocjacyjnej i odpowiadający parametr formalny zaznacza się gwiazdką
  - Python: parametry aktualne są listą wartości; parametry formalne są podane jako nazwa poprzedzona gwiazdką
  - Lua: zmienną liczbę parametrów reprezentuje parametr formalny zakończony trzema kropkami ...
  - C#: formalny parametr poprzedzony słowem **param** , parametry aktualne muszą być tego samego typu

Parametry formalne. Przykłady

- Python. Parametry nazwane (klucze) i pozycyjne

```
dodaj(dlugosc=x_dlugosc, lista=x_lista,
      suma=x_suma)
```

gdzie formalnymi parametrami są: `dlugosc, lista i suma`. Ada, Fortran95 , Python pozwalają na parametry pozycyjne. Oba rodzaje parametrów można mieszać. Np.

```
dodaj(x_dlugosc, suma=x_suma, lista=x_lista)
```

Ograniczenie: jeśli na liście pojawi się parametr z kluczem to kolejne muszą być też z kluczem.

Parametry formalne. Przykłady

- Python, Ruby, C++, Fortran95+, Ada
  - parametry mogą mieć wartości domyślne.
    Jeśli nie ma parametru aktualnego, używana jest wartość domyślna, predefiniowana w nagłówku funkcji. Np.

```
def oblicz_podatek(kwota, odliczenie=1, stopa)
  ...
```

    Możliwe wywołanie:

```
xy = oblicz_podatek(kwota, stopa_procent)
```

    Ponieważ C++ nie wspiera parametrów z kluczem, więc parametry opcjonalne (*default*) podaje się na końcu listy parametrów (formalnych/aktualnych)

Parametry formalne. Przykłady

- Ruby
  - lista parametrów o różnej długości

```
lista = [1,2,3,4,5];
def test(p1,p2,p3, *p4)
   ...
end
...
test('jeden', pon=>70, wt=>87, sr=>40, *lista);
```

Wewnątrz 'test' parametry mają następujące wartości:

```
p1: 'jeden'
p2: {pon => 70, wt => 87, sr => 40}
p3: 1
p4: [2,3,4,5]
```

Bloki w Ruby

- W Ruby istnieją funkcje iteratorowe, których używa się często do przetwarzania tablic
- Iteratorów używa się wraz z blokami definiowanymi w aplikacjach
- Bloki są dołaczonymi wywołaniami metod; mogą posiadać parametry;
  są wykonywane gdy wykonywana jest instrukcja **`yield`**;

```
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first, second = second, first + second
  end
end

puts ”Liczby Fibonacciego mniejsze niż 100:"
fibonacci(100) {|num| print num, " "}
```

Wynik: `1 1 2 3 5 8 13 21 34 55 89 => nil`

Procedury i funkcje

- Dwie kategorie podprogramów
  - *procedury* – zbiory instrukcji definiujące obliczenia parametryzowane
  - *funkcje* – strukturalnie przypominają procedury; semantycznie są modelowane na funkcjach matematycznych
    - nie powinny produkować efektów ubocznych
    - w praktyce produkują efekty uboczne

Przykład: efekty uboczne


```
C
------------------
int a = 5;
int fun() {
    a=17;
    return 3;
}/* koniec fun */
void main(){
    a = a + fun();
}/* koniec main */
```

- Wartość a wyliczana w **`main`** zależy od kolejności realizacji obliczeń w wyrażeniu **`a+fun()`**. Wynikiem jest 8 (jeśli najpierw oblicza się **`a`**), lub 20 (jeśli najpierw wywoływana jest funkcja **`fun(a)`**)
- Funkcje matematyczne nie powodują efektów ubocznych
- Trudne do realizacji zadanie eliminacji efektów ubocznych

Problemy projektowania podprogramów

- Czy zmienne lokalne są statyczne czy dynamiczne?
- Czy można definiować podprogramy w innych definicjach podprogramów?
- Jakie są metody przekazywania parametrów?
- Czy sprawdzane są typy parametrów?
- Jeśli można przekazywać podprogramy jako parametry i jeśli podprogramy można zagnieżdżać to jakie jest środowisko referencyjne takich podprogramów?
- Czy podprogramy można przeciążać?
- Czy podprogramy mogą być ogólne (generic)?
- Jeśli dozwolone jest zagnieżdżanie to czy są wspomagane domknięcia?

Lokalne środowiska referencyjne

- Podprogramy definiują własne zmienne, określając w ten sposób lokalne środowisko referencyjne. Te zmienne są tworzone dynamicznie na stosie
  - Plusy
    - możliwość rekurencji
    - Pamięć dla zmiennych lokalnych może być dzielona między wieloma podprogramami
  - Minusy
    - czas alokacji/dealokacji zmiennych
    - adresowanie pośrednie (miejsce na stosie określa się dopiero w czasie wykonania; do statycznych bezpośrednie)
    - podprogramy nie mogą zależeć od historii (nie *pamiętają* wartości zmiennych między wywołaniami!) (Jeśli wszystkie zmienne lokalne są dynamiczne, powstają i znikają, to jak np. wygenerować liczby pseudolosowe? Patrz też: współprogramy.)

- Zmienne lokalne są statyczne
  - sytuacja odwrotna do poprzedniej (brak rekurencji,...)

Lokalne środowiska ref. Przykłady

- W większości języków zmienne lokalne są dynamiczne na stosie
- C++, jak wyżej ale można zmienne deklarować jako `static`
- metody w C++, Java, Python i C# uzywają tylko zmiennych lokalnie dynamicznych na stosie
- Lua: wszystkie jawnie deklarowane zmienne są globalne; zmienne zadeklarowane jako `local` są dynamicznymi zmiennymi stosu

Modele semantyczne przekazywania parametrów

- tryb in – wejściowy
- tryb out – wyjściowy
- tryb in-out – wejściowo-wyjściowy

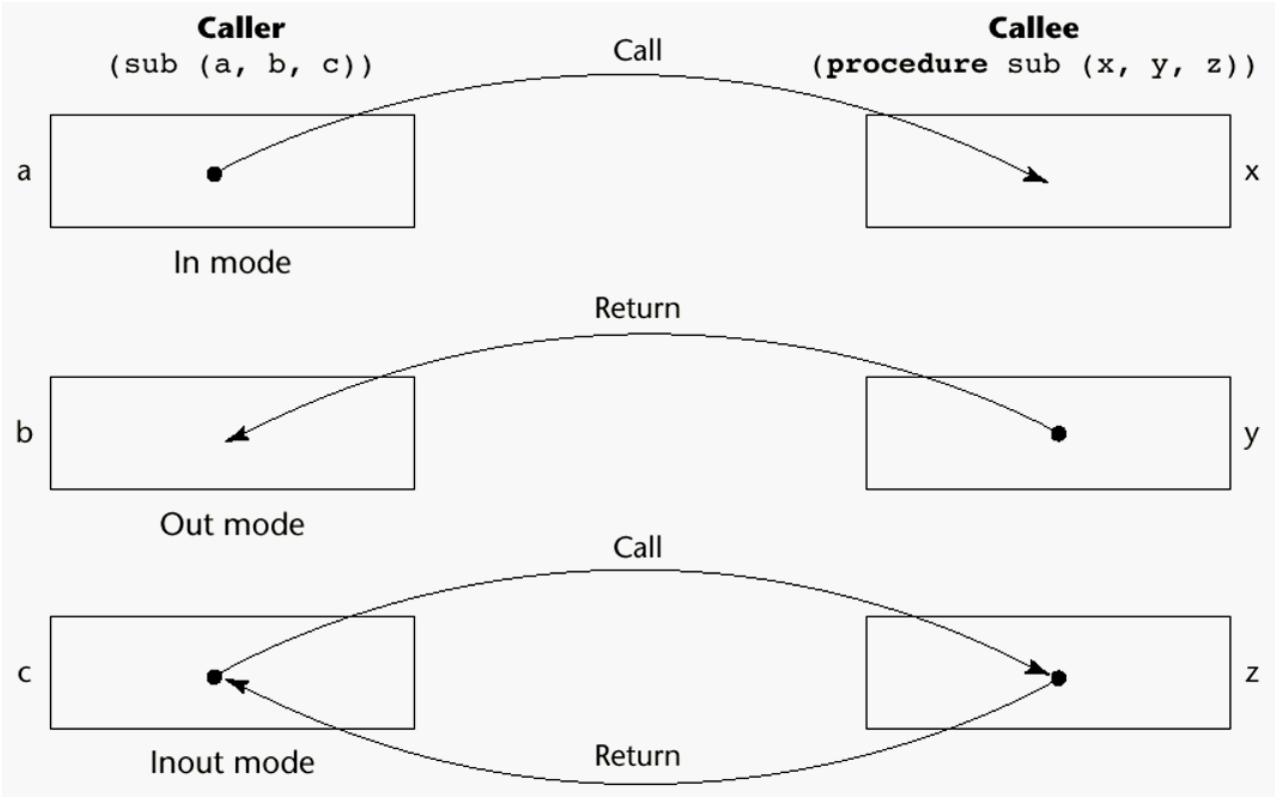
Sposoby przekazywania parametrów

Tryby

in (*a*)

out (*b*)

in-out (*c*)



Koncepcje przekazywania par.

- fizyczne przekazanie wartości
- przekazanie ścieżki do wartości

Przekazywanie przez wartość (in)

- W przekazywaniu parametrów zawsze bierze udział stos
- Przekazanie przez wartość.Wartość parametru aktualnego umieszczana jest na stosie i inicjuje wartość parametru formalnego, który zachowuje się jak zwykła zmienna lokalna
  - realizacja: zwykłe kopiowanie
  - Minus: (fizyczne przekazanie) wymaga dodatkowej pamięci; duży koszt czasowy w przypadku *dużych* parametrów
  - Minus: (przekazanie ścieżki dostępu) ochrona przed zapisem w programie wywoływanym i dostęp są drogie (pośrednie adresowanie)

Przekazywanie przez wynik (out)

- parametr formalny zachowuje się w podprogramie jak zmienna lokalna; jego wartość jest przekazywana do podprogramu wywołującego odwrotnie niż opisano to w trybie *in*, w drugą stronę
  - dodatkowa pamięć oraz koopiowanie
- Możliwe problemy: dwa jednakowe parametry aktualne lub parametry wpływające na siebie

```
sub(p1, p1);
```

Który z przekazanych parametrów będzie końcową wartością **`p1`**?

```
sub(list[s], s);
```

Czy adres `list[s]` wylicza się wcześniej, czy później w podprogramie?

„przez wynik” – przykład


```
procedure sub(int i, int j)
   i = 3;   j = 2;
end
...
sub(k,k);
print k;
  ...
v[1]=2;
v[2]=3;
v[3]=8;
s = 2;
sub(v[s], s);
print (s, v[1], v[2], v[3]);
```

Jakie będą wyniki tego programu w przypadku gdy parametry są przekazywane przez wynik? Zależy ...

Przekazywanie przez referencję

- Przekazuje się przez wartość ścieżkę dostępu do danych. Podprogram działa więc bezpośrednio na danych bo posiada do nich wskaźnik
- Plusy:
  - nie potrzeba dodatkowej pamięci ani kopiowania
  - wszystkie tryby semantyczne
- Minusy:
  - działa wolniej niż przekazywanie przez wartość
  - niepożądane efekty uboczne (kolizja)
  - niepożądane aliasowanie

```
fun(x, x); fun(list[i], list[j]);
fun(list[i], i);
```

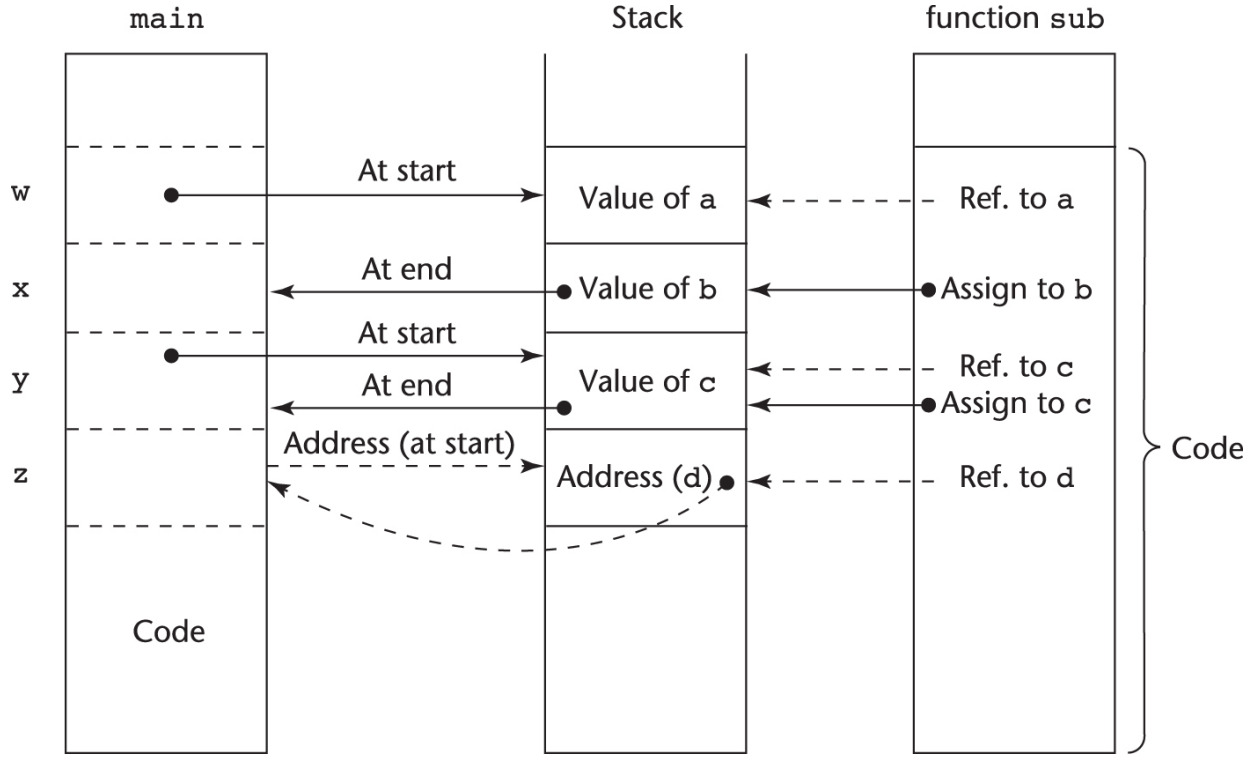
Przekazywanie przez nazwę

- Przekazywanie in-out – wartość parametru formalnego nie jest obliczana, lecz przekazywana jest jego *postać*, tzn. *przepis* na obliczenie wartości parametru; w razie potrzeby następuje obliczanie parametru przez podprogram (in) lub program wywołujący (out)

Realizacja metod przekazywania

- w większości języków przekazywanie odbywa się za pośrednictwem stosu
- metoda przekazywania przez referencję jest najłatwiejsza w realizacji; na stosie umieszczany jest tylko adres

Realizacja... cd



(Sebesta)

nagłówek funkcji: `void sub(int a, int b, int c, int d)`
wywołanie w main: `sub(w, x, y, z)`
(przekaz `w` przez wartość, `x` przez wynik, `y` przez wartość-wynik, `z` przez referencję)

Przekazywanie parametrów w głównych językach programowania

- C
  - Przekazywanie przez wartość
  - Przekazywanie przez referencję odbywa się poprzez wskaźniki jako parametry
- C++
  - Specjalny typ wskaźnika zwany referencją pozwala przekazać parametr przez referencję
- Java
  - Wszystkie parametry są przekazywane przez wartość
  - Parametry “object” są przekazywane przez referencję
- Ada
  - Trzy sposoby semantyczne przekazywania: **`in`**, **`out`**, **`in out`**; sposób in jest opcjonalny
  - Formalne parametry deklarowane jako **`out`** – możliwe przypisanie, niemożliwe odwołanie się;
    Parametry deklarowane jako **`in`** – możliwe odwołanie, niemożliwe przypisanie;
    Parametr przekazywany **`inout`** działa jak **`in`** oraz **`out`**

Przekazywanie parametrów w głównych językach

- Fortran 90
  - można deklarować wszystkie sposoby: **`in, out, inout`**
- C#
  - opcja: przekazywanie przez wartość
- Perl
  - wszystkie parametry umieszcza się w specjalnej tablicy @_
- Python, Ruby
  - parametry są przekazywane przez przypisanie; parametr aktualny jest przypisywany parametrowi formalnemu (wszystkie dane są obiektami)

Sprawdzanie typów parametrów

- podnosi poziom niezawodności
- Fortran 77, stare C – brak sprawdzania
- Pascal, FORTRAN 90+, Java, and Ada zawsze wymagane
- ANSI C and C++: zależy od użytkownika
  - Prototypy
- Perl, JavaScript, and PHP nie wymagają sprawdzania typów
- Python, Ruby: zmienne nie posiadają typów (tylko obiekty) i dlatego niemożliwe jest ich sprawdzanie

Przekazywanie tablic

- Kompilator musi posiadać informacje o wymiarach itp. aby przygotować operacje rezerwacji pamięci dla tablicy
- różne rozwiązania zależnie od języka

Podprogramy jako parametry

- Potrzeba przekazywania podprogramów jako parametrów (sytuacje)
- Problemy
  - czy sprawdzane są typy?
  - jakie jest właściwe środowisko referencyjne przesyłanych podprogramów?

Podprogramy zagnieżdżone

- zależność od typu wiązania
  - wiązanie *płytkie* – środowisko lokalne instrukcji wykonującej przekazany podprogram
  - wiązanie *głębokie* – środowisko, w którym zdefiniowano podprogram wywoływany
  - wiązanie *ad hoc* – środowisko instrukcji przekazującej wywoływany podprogram

Podprogramy zagnieżdżone

- Javascript

```
function sub1() {
  var x;
  function sub2() {
    alert(x);//dialogbox
  };
  function sub3() {
    var x;
    x = 3;
    sub4(sub2);
  };
  function sub4(subx) {
    var x;
    x = 4;
    subx();
  };
  x = 1;
  sub3();
};
```

- Rozpatrujemy wywołanie sub4(sub2) w funkcji sub3() w zależności od rodzaju wiązania
  - wiązanie płytkie – to co jest w sub4, a więc x=4 jest wynikiem.
  - wiązanie głębokie – sub1 jest środowiskiem referencyjnym, a więc x=1 i wynikiem jest 1.
  - wiązanie ad hoc – środowisko jest lokalne w sub3, x=3, wynik 3.

Wywołania pośrednie

- Gdzie? Tam gdzie jest wiele możliwych podprogramów do wywołania; decyzja w trakcie wykonania (obsługa zdarzeń, środowiska interfejsów graficznych; wywołania zwrotne)
- C, C++; wskaźniki typowane mogą wskazywać tylko funkcje o tych samych protokołach (np. `float (*pfun) (float, int);` - każda funkcja, która posiada dwa parametry formalne `int` i zwraca `float` może być wywołana wskazaniem `pfun`))

```
int mfun2(int, int);    // deklaracja
int (*pfun)(int, int) = mfun2;
      // kreacja wskaźnika i przypisanie
pfun2 = mfun2;    // przypisanie adresu funkcji
                  // do wskaźnika
```

Wywołanie:

```
(*pfun2)(jeden, dwa);
pfun2(jeden, dwa);
```

Wywołania pośrednie

- C# - delegaty
  - Wywołanie metod jest delegowane (czytaj: zlecane) delegatom – obiektom zawierającym przesyłane metody
  - Należy najpierw utworzyć klasę delegata z właściwym protokołem metody. Instancja, a więc obiekt delegata, przechowuje nazwę metody wraz z jej protokołem.
    `public delegate int Change(int x);`
    Delegata tej klasy można utworzony z metodą o jednym parametrze formalnym `int`, zwracającą wartość `int`. Weźmy np. `static int fun1(int x);`
    Delegata klasy `Change` tworzy się przez przekazanie do konstruktora klasy nazwy metody, a więc tutaj `fun1`:
    `Change chgfun1 = new Change(fun1); //` lub
    `Change chgfun1 = fun1;`
    Poniższy przykład ilustruje wywołanie delegowanej metody
    `chgfun1(12);`
  - Można delegować dodatkowe, nowe metody (dodawać do delegata), np: `Change chgfun1 += fun2; //` dodajemy nową metodę
  - Delegowane metody wykonują się w kolejności dodawania, jedna za drugą (*multicast delegate*)
  - Zwracana jest ostatnia wartość (najczęściej void)

Przeciążanie podprogramów

- Podprogram przeciążony (pp) posiada taką samą nazwę jak inny istniejący podprogram
  - każda wersja pp posiada unikatowy protokół
- Języki: C, C++, Java, Ada, ..., Fortran ...
- Ada: typ zwracanej wartości pozwala rozróżniać pp

Podprogramy ogólne (generyczne)

- Podprogramy ogólne (polimorficzne) przyjmują różnego typu parametry
- Podprogramy przeciążone zwyczajnie – polimorfizm *ad hoc*
- Polimorfizm dynamiczny (typów pochodnych) – typy pochodne od typu T traktowane są jak T (patrz OOP)
- Polimorfizm oparty na sygnaturach (typ określają stosowane metody; *kacze typowanie*)

Podprogramy ogólne

- C++
  - template (szablon)

```
template <class Type>
   Type max(Type first, Type second) {
   return first > second ? first : second;
   }
```

- Java
  - różnice w Java i C++:

Parametrami ogólnymi w Java są klasy
Metody ogólne są tworzone jednorazowo

Podprogramy ogólne

- Java 5 cd
  - `public static <T> T doIt(T[] list) { … }`
    Parametr jest tablicą ogólnych elementów; T jest typem;
  - Wywołanie:

```
doIt<String>(myList);
```

  - Parametry mogą posiadać zakres

```
public static <T extends Comparable> T
    doIt(T[] list) { … }
```

    Parametr ogólny jest klasą, która implementuje interfejs **Comparable**

Operatory przeciążone użytkownika

- Operatory można przeciążać w C++, Ada, Python, Ruby

Python:

```
def __add__ (self, second) :
      return Complex(self.real + second.real,
                     self.imag + second.imag)
```

Użycie, obliczenie `x + y: x.__add__(y)`

Domknięcia

- Domknięcie == podprogram plus jego środowisko referencyjne, w którym został zdefiniowany (wynik podprogramu jako podprogram)
  - środowisko referencyjne jest wymagane przy wywołaniu podprogramu z dowolnego miejsca programu
  - w językach o zakresach statycznych, w których nie zezwala się na zagnieżdżanie podprogramów domknięcia nie są wymagane
  - Domknięcia są potrzebne wtedy gdy podprogram używa zmiennych z zakresów zagnieżdżanych i jest wywoływany w dowolnym miejscu

Domknięcia. Przykład

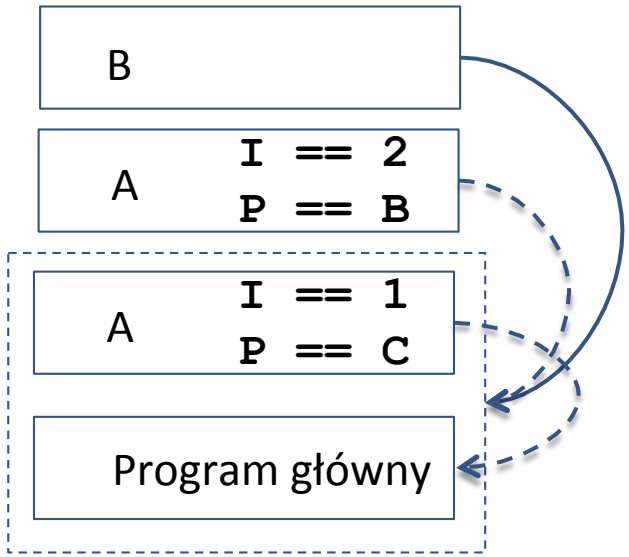
- Javascript

```
function makeAdder(x) {
   return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) +
               "<br />");
document.write("add 5 to 20: " + add5(20) +
               "<br />");
```

Domknięciem jest anonimowa funkcja zwracana przez `makeAddr`

Domknięcia. Przykład


```
program binding_example(input,
                        output);
procedure A(I:integer;
            procedure P);
  procedure B;
  begin
    writeln(I);
  end;
begin (* A *)
  if I > 1 then
     P
  else
     A(2, B);
end;
procedure C; begin end;
begin (* main *)
   A(1, C);
end.
```



Schemat stosu programu z lewej strony.

Pascal. Środowisko referencyjne zawarte w domknięciach oznaczono przerywaną linią.Po wywołaniu **B** poprzez formalny parametr **P** istnieją dwie kopie **I**. Ponieważ domknięcie dla P powstało w chwili pierwszego wywołania **A**, statyczny link **B** (ciągła strzałka) wskazuje na ramkę tego właśnie wywołania. **B** używa tej wartości **I** w instrukcji **writeln** i wynik jest równy **1**.

Domknięcia w OOP

- W językach OOP pola obiektów tworzą kontekst wywołań podprogramów (enkapsulacja)
- Java

```
Interface IntFunc() {
   public int calc(int i);
}
class PlusX implements IntFunc {
   final int x;
   PlusX (int n) {x = n};
   public int call(int i) {return i + x;}
}
...
IntFunc f = new PlusX(2);
System.out.println(f.call(3));   // drukuje 5
```

- C#
  - delegaty

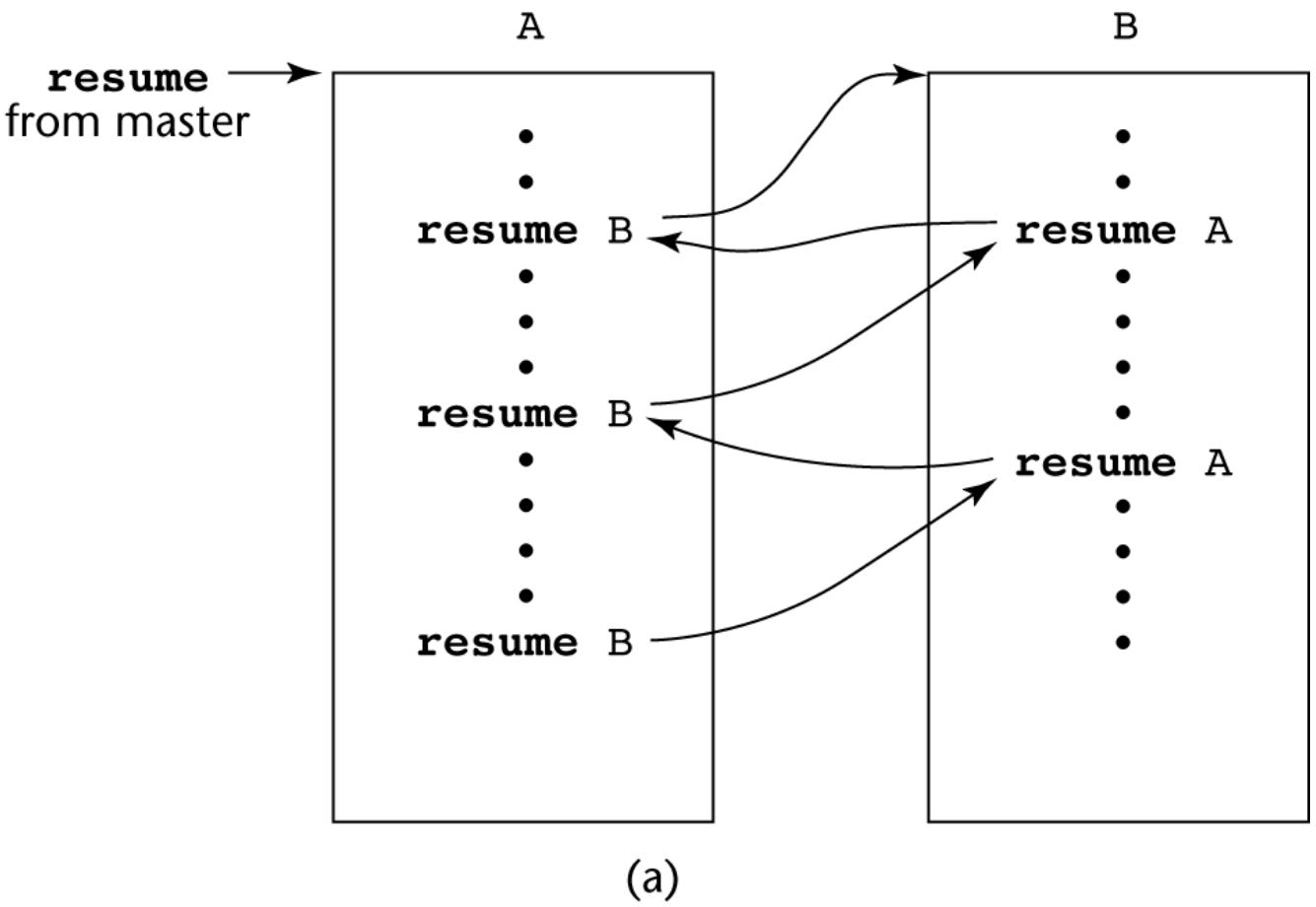
Współprogramy

- Współprogram jest podprogramem o wielu wyjściach, które sam potrafi kontrolować (coroutine)
- program wywołujący i wywoływany traktowane są równorzędnie
- Wywołanie współprogramu nosi nazwę wznowienia (*resume)*
- Pierwsze wywołanie powoduje start współprogramu od jego początku, następne od dowolnego punktu
- Współprogramy wznawiają się wzajemnie
- Jest to rodzaj wykonania prawie współbieżnego; wykonanie jest na zmianę i bez przekryć w czasie

Współprogramy

Wywołania wp.
A rozpoczyna.
Potem mamy kolejne wznowienia.

(Sebesta)

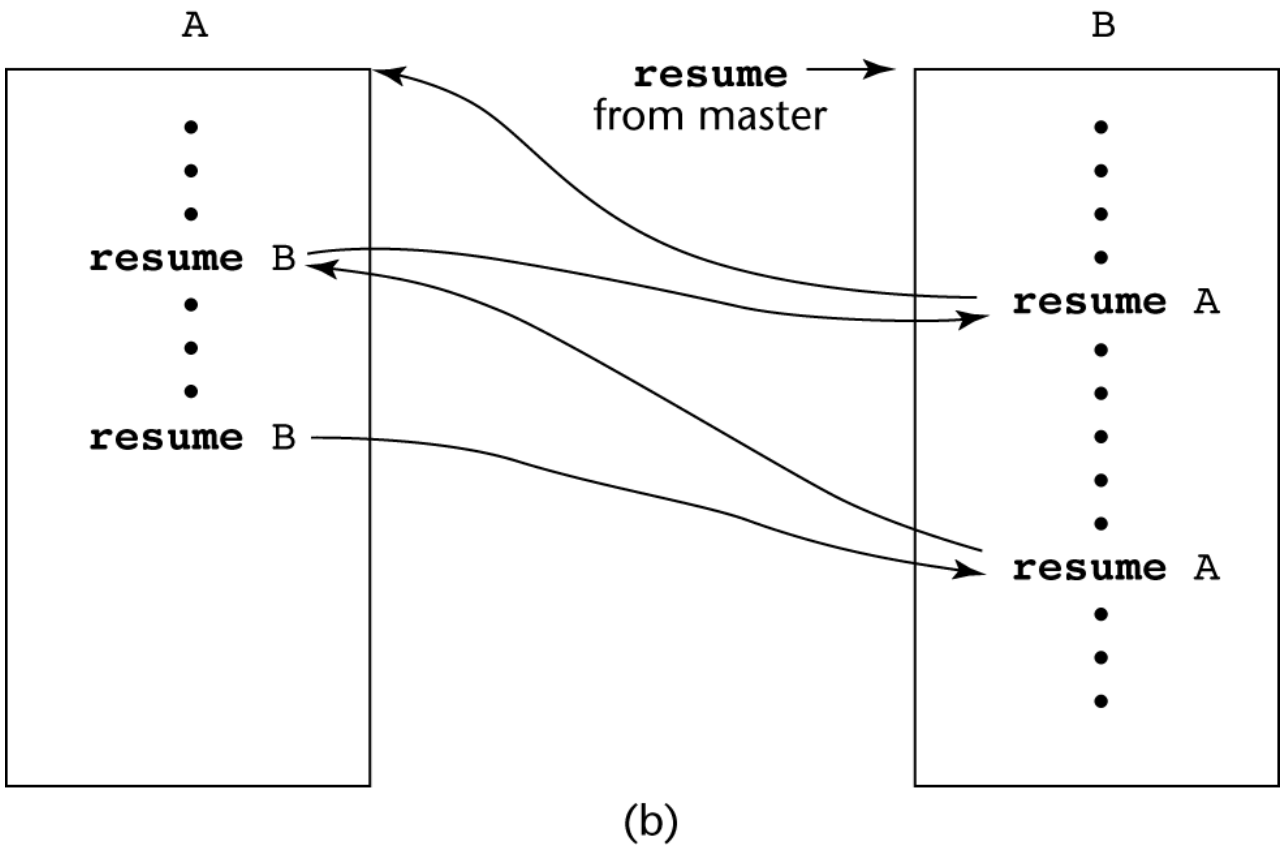


Współprogramy

Wywołania wp.

Rozpoczyna B. Wznawiane jest A itd.

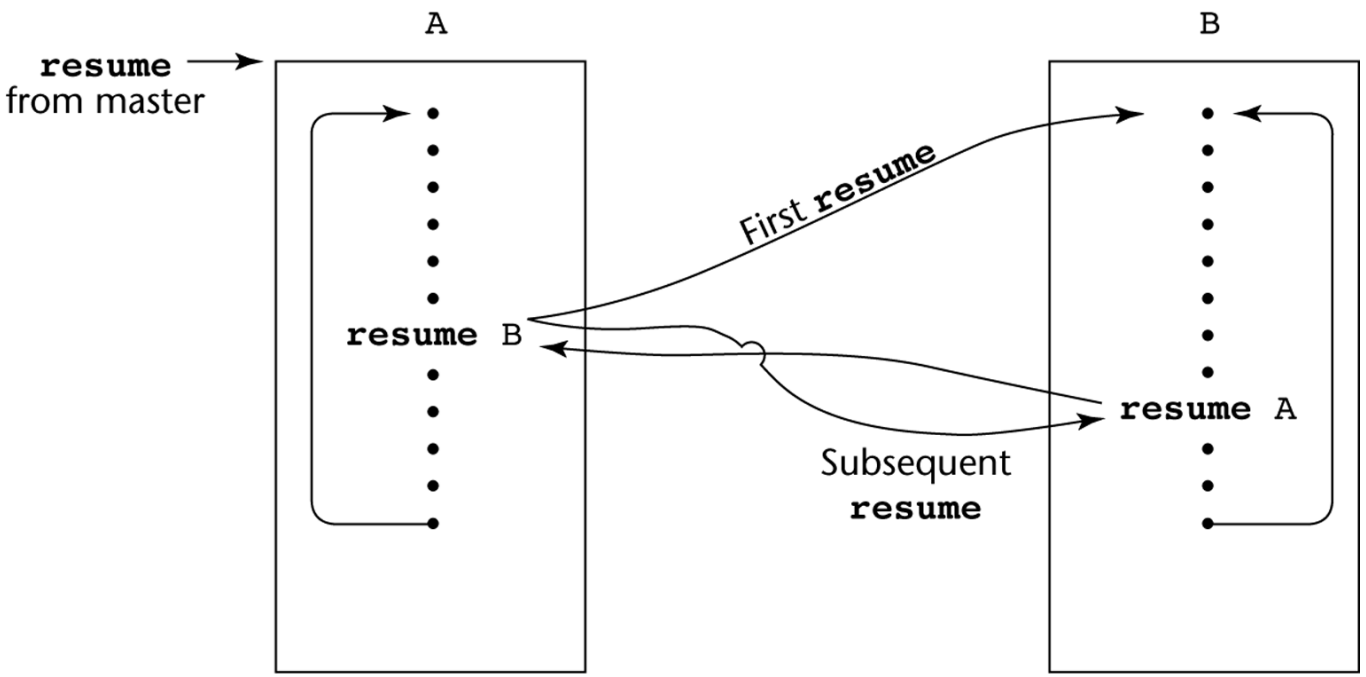
(Sebesta)



Współprogramy

Wywołania
wp.
w pętlach ...

(Sebesta)

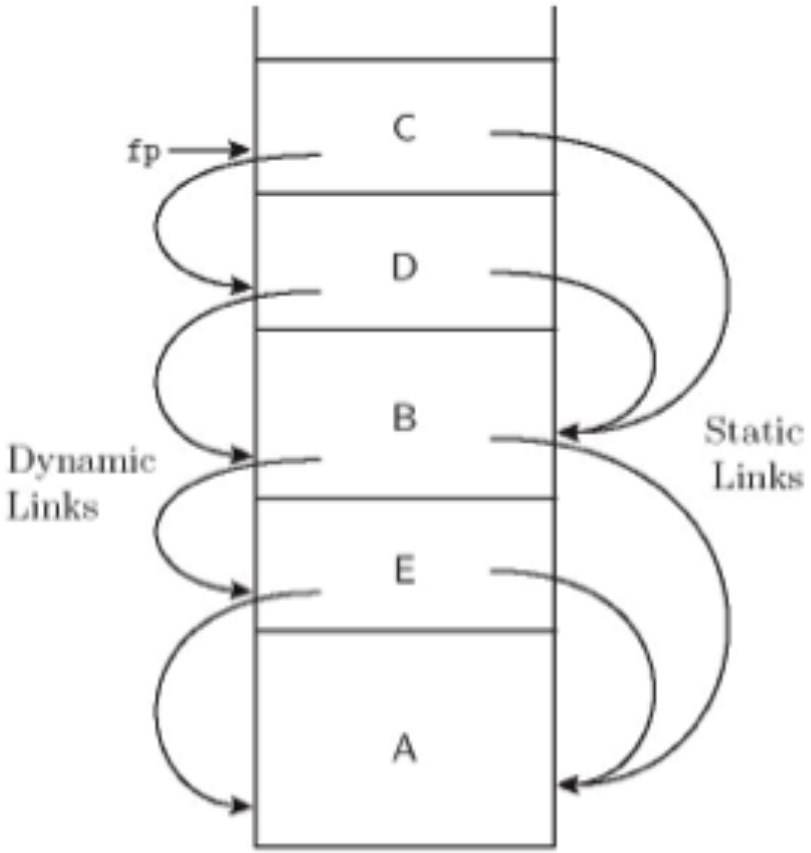


Organizacja stosu

Zagnieżdżanie procedur.

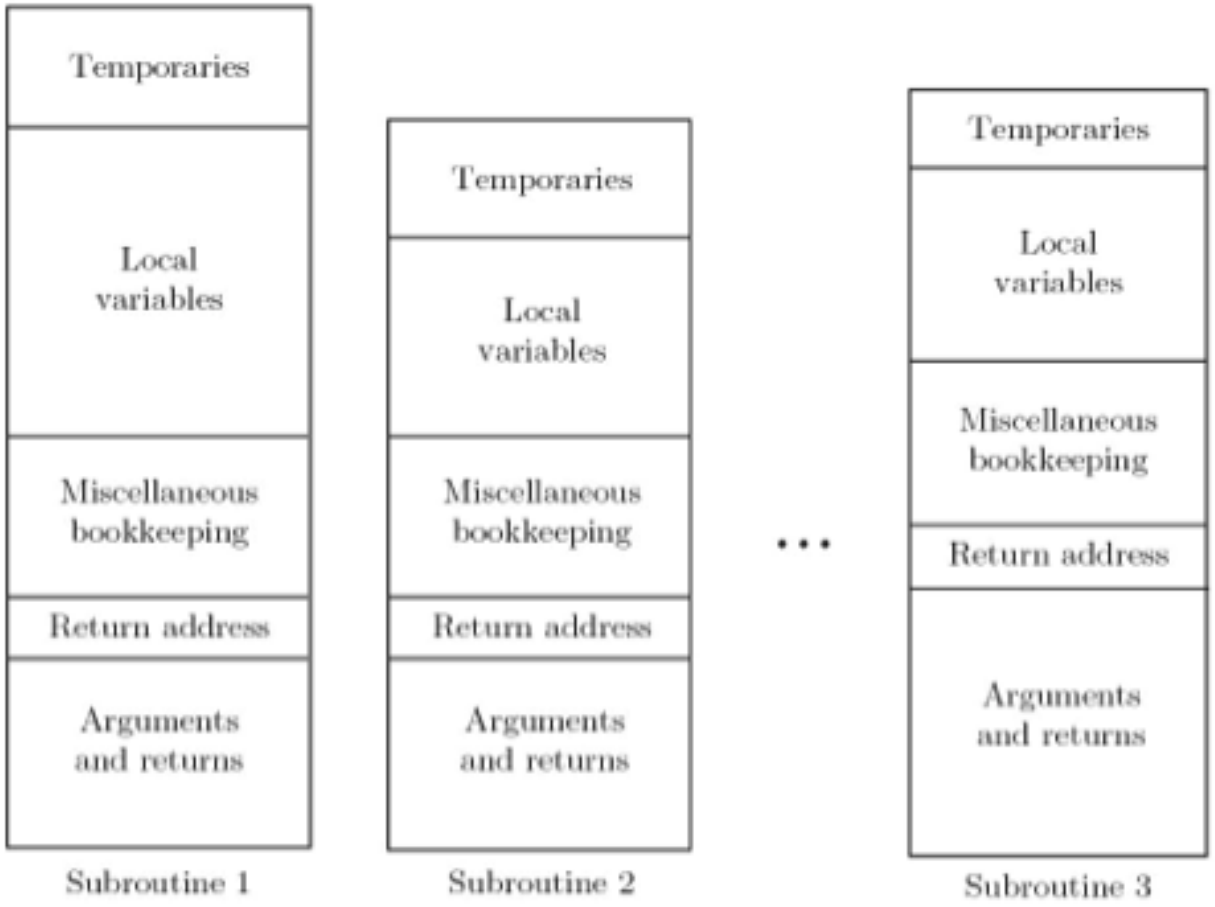
Z procedur B, C i D widać wszystkie podprogramy. Z A i E widać A, B, E. Stos dla wywołań A, E, B, D, C. Pokazano łącza statyczne i dynamiczne.

(Scott)



Alokacja podprogramów

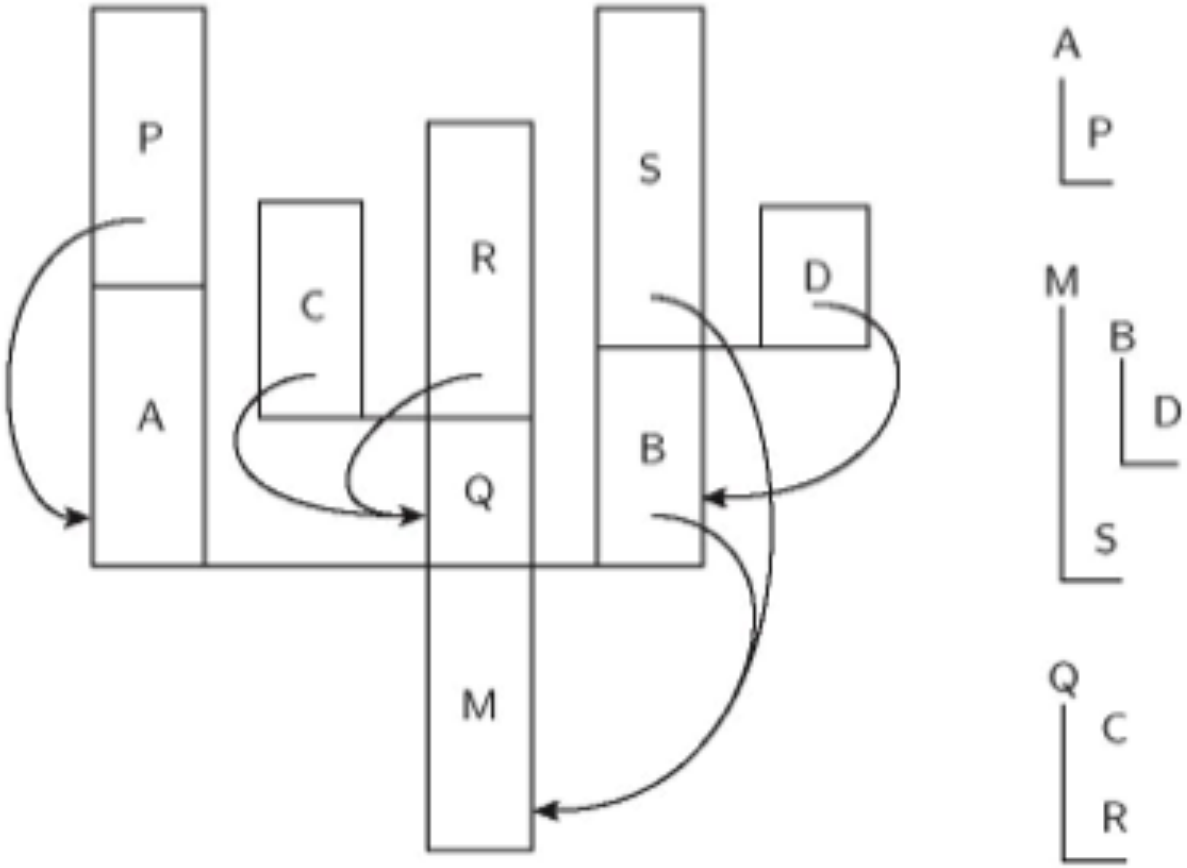
Statyczna alokacja pamięci dla podprogra-mów w językach bez rekurencji



Współprogramy

Stosy współpro-gramów pokazanych z prawej strony (kaktus)

(Scott)



za tydzień ... !?