PARADYGMATY I JĘZYKI PROGRAMOWANIA

Zmienne. Typy danych.

Treść

- Zmienne
  - nazwa, zakresy,...
- Typy danych
  - wstęp
    - definicje
    - klasyfikacja
  - Sprawdzanie typów
    - równoważność
    - wnioskowanie typów
  - Rekordy, warianty
  - Tablice
  - Napisy
  - Wskaźniki i typy rekurencyjne

Zmienne

Zmienna

- Abstrakcja komórki lub zbioru komórek pamięci komputera (w językach wysokiego poziomu)
  - atrybuty
    - nazwa
    - adres; aliasy: różne nazwy = ten sam adres;
    - wartość
    - typ (zakres wartości zmiennej i operacje)
    - czas życia
    - zakres

Nazwa

- Nazwa = napis identyfikujący pewien byt w programie; identyfikator alfanumeryczny
  - długość (Fortran 95: 31 znaków; Ada, C++ - brak ograniczeń;
  - rozróżnialność dużych i małych liter w nazwach (większość języków rozróżnia wielkość liter)
  - słowa specjalne, kluczowe (`int, integer, program, while;` Fortran I: `integer real` i `real integer`)
  - znaczenie zależne od sposobu zapisu – typowanie na podstawie nazw (Fortran: `a-h, o-z - real, i-n – integer`; `implicit none`; Ruby: `Klasa` – duża pierwsza litera; Perl - `$x`, `@x`, `%x` itd;)
  - nazwy predefiniowane w zewnętrznych pakietach (import nazw)
  - zalecany zapis nazw, czytelność; majuskuły czy minuskuły?

Wiązanie

- związek między atrybutem i bytem lub między operacją i symbolem; np. zmiennej nadajemy nazwę lub wartość – mówimy o wiązaniu zmienna-nazwa lub zmienna-wartość;
- czas wiązania = czas kiedy tworzy się ów związek; możliwości:
  - etap projektowania języka (typy podstawowe, konstruktory typów)
  - czas implementacji języka (realizacja typów podstawowych, i/o, organizacja stosu i sterty)
  - okres tworzenia programu
  - czas kompilacji
  - etap łączenia (import pakietów, dołączanie bibliotek)
  - ładowanie kodu (przydzielanie adresów fizycznych)
  - czas wykonania

Przykład

- `x = x + 7`
  - wiązanie x z jego typem powstało w czasie kompilacji
  - możliwe wartości x zostały określone jeszcze na etapie projektowania kompilatora (języka)
  - znaczenie operatora + określono w czasie kompilacji kiedy znane były typy operandów
  - wartość `x` została określona w czasie wykonania programu

Wiązanie zmienna-atrybut

- statyczne
  - wiązanie przed czasem wykonania; stałe, niezmienne w czasie wykonania
- dynamiczne
  - wiązanie w czasie wykonania programu

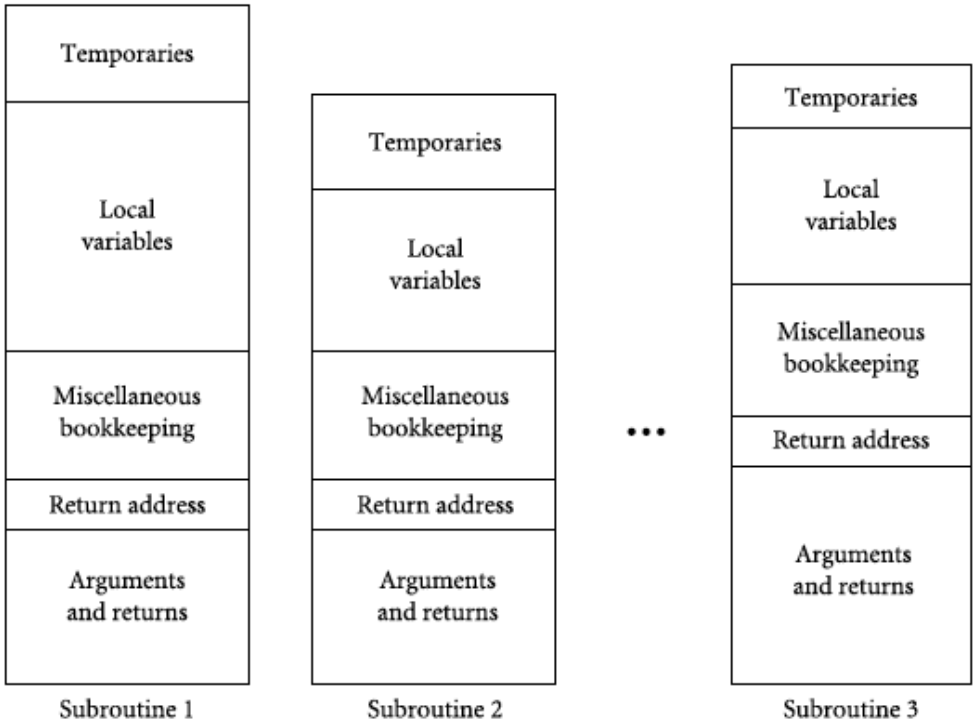
Wiązanie z typem

- jawne (*explicite*) i niejawne (*implicite*) deklaracje zmiennych
  - jawne: C, C++, Pascal;
  - niejawne: Perl, Fortran, JavaScript, Ruby)
- dynamiczne wiązanie typów (podczas przypisania wartości) (np. javascript: x = [1, 3.4, 7], a za chwilę x=25)
- wnioskowanie typów; np. na podstawie typu prawej strony określa się typ strony lewej (język ML: **`fun pole = pi*r*r;`**)

Wiązanie pamięci

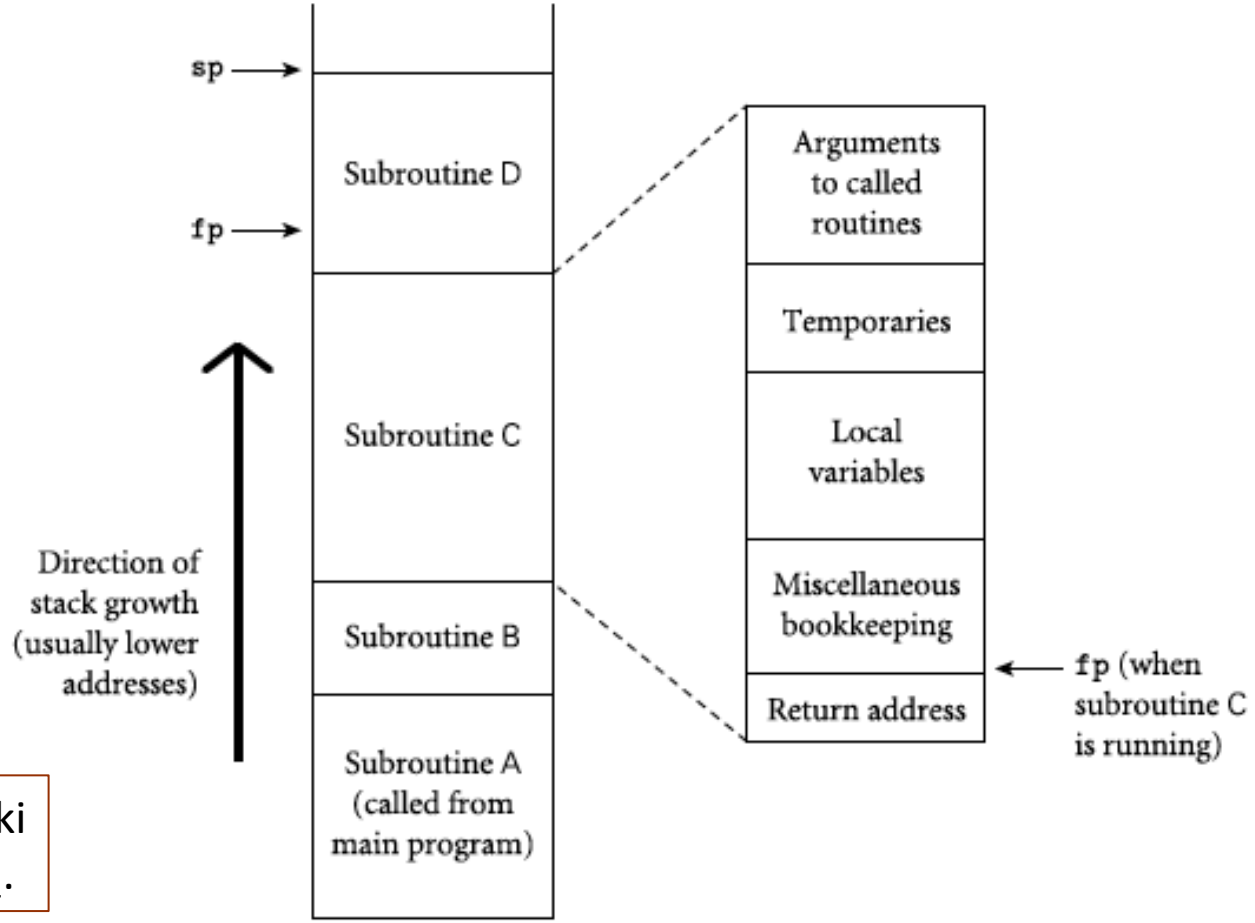
- rezerwacja/zwalnianie pamięci (allocation/deallocation)
- czas życia zmiennej = czas w którym zmienna związana jest z danym obszarem pamięci
  - kategorie wiązań:
    - statyczne (związki powstałe przed wykonaniem; zmienne globalne o adresach stałych: literały liczbowe, napisy, itd.; np. w C, C++: **`static`**)
    - dynamiczne na stosie (wywołania procedur)
    - jawne, dynamiczne na stercie (powstałe w dowolnym czasie wykonania)
    - niejawne, dynamiczne na stercie (powstałe w dowolnym czasie wykonania); związane z tym zbieranie śmieci = odzyskiwanie pamięci na stercie

Wiązanie pamięci - stos



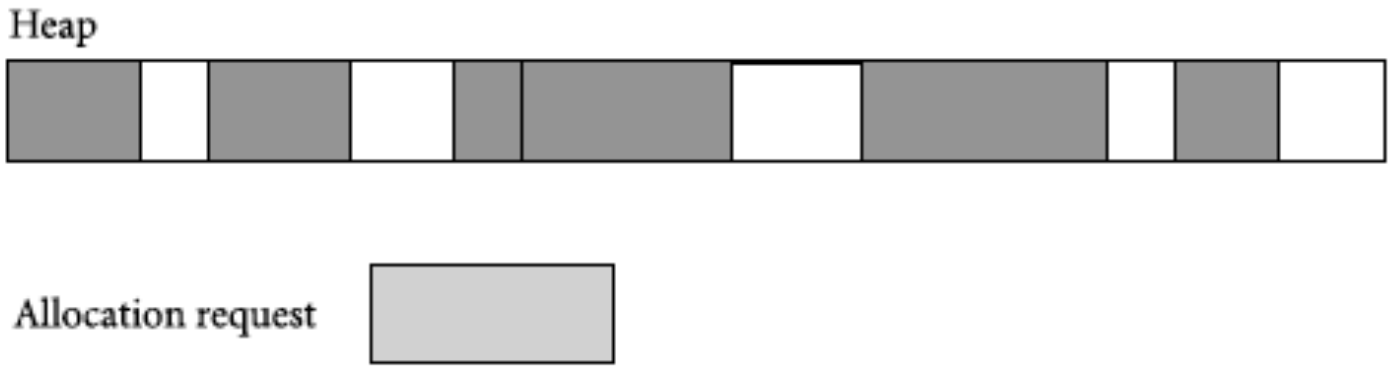
(Scott) Języki bez rekurencji.

Wiązanie pamięci - stos



(Scott) Języki z rekurencją.

Wiązanie pamięci - sterta



(Scott) Obraz sterty.

Sprawdzanie typów

- silne typowanie (po 1970) – zawsze wykrywane są błędy typów (Java, C# - silnie typowane; C, C++ nie jest silnie typowany ze względu na unie)
- zgodność typów (zmiennych, wyrażeń)
  - zgodność nazw typów (zmienne są tych samych typów gdy są zadeklarowane rezem lub w deklaracjach z tą samą nazwą typu)
  - zgodność strukturalna

Zakresy zmiennych

- zakres = zbiór instrukcji gdzie dana zmienna jest widoczna, tzn. można się do niej odwołać
  - zakres statyczny
  - bloki
  - zakres dynamiczny

Zakres statyczny

- Algol 60 – zakresy statyczne, leksykalne; zakres można wyznaczyć przed wykonaniem
- Dwa rodzaje języków z zakresami statycznymI:
  - podprogramy można zagnieżdżać (Pascal, Ada, JavaScript, LISP, Scheme, Fortran 2003+, F#, Python)
  - podprogramów nie można zagnieżdżać; zakresy zagnieżdżane przez klasy i bloki (języki C pochodne)

Przykład: zagnieżdżanie, Pascal


```
procedure P1(A1 : T1);
var X : real;
   ...
   procedure P2(A2 : T2);
      ...
      procedure P3(A3 : T3);
      ...
      begin
         ...      (* kod P3 *)
      end;
      ...
   begin
         ...       (* kod P2 *)
   end;
   ...
```

```
   procedure P4(A4 : T4);
      ...
      function F1(A5 : T5) : T6;
      var X : integer;
      ...
      begin
         ...       (* kod F1 *)
      end;
      ...
   begin
      ...         (* kod P4 *)
   end;
   ...
begin
...       (* kod P1 *)
end.
```

Przykład: zakres statyczny


```
function sub () {
    function sub1() {
        var x=3;
        sub2();
    }
    function sub2() {
        var y=x;
    }
    var x=4;
    sub1();
}
```

- Przy statycznych zakresach przypisanie `y=x` w `sub2` odnosi się do `x` zadeklarowanego w `sub`.

Bloki

- pozwalają określać zakresy statyczne wewnątrz wykonywalnego kodu; zmienne lokalne, dynamiczne na stosie (alokacja/dealokacja w chwili gdy rozpoczyna się/kończy wykonywanie bloku); JĘZYKI STRUKTURALNE (blokowe); C i jego pochodne;
- Przykład deklaracje w bloku

```
{
  int temp = a;
  a = b;
  b = temp;
}
```

Przykłady

- C, C++

```
void sub() {
    int licz;
    ...
    while (...) {
        int licz;
        licz++;
        ...
    }
}
```

Powyższy kod nie jest poprawny w Java, i C#.

- Scheme; bloki **LET**

```
(LET (
    (licznik (+ a b))
    (mianownik (- c d)))
    (/ licznik mianownik)
)
```

Kod oblicza **(a+b)/(c-d)**. Zmienne **a, b, c, d** są lokalne w bloku **LET**.

Bloki LET występują w j. funkcyjnych.

Zakres globalny

- Wiele języków ze strukturą podprogramów pozwala definiować zmienne poza podprogramami. Dostęp do nich mogą mieć wszystkie podprogramy. Są to zmienne globalne.
- C, C++: **`extern int calkowita;`**
- W C++ do zmiennych globalnych przesłoniętych przez lokalne można się odwołać używając operatora zakresu (**`::`**), np. jeśli **`x`** jest przesłonięta to używamy **`::x`** w odwołaniu do niej;

Zakres dynamiczny

- Zmienne w językach APL, SNOBOL4 i wcześniejszych wersji LISP, oraz Perl i Common LISP mają zakresy dynamiczne = decyduje kolejność wywoływania podprogramów, a nie relacje przestrzenne między zmiennymi;
  Zakres można wyznaczyć tylko w czasie wykonywania programu.

Przykład: zakresy dynamiczne


```
function sub () {
    function sub1() {
        var x=3;
        sub2();
    }
    function sub2() {
        var y=x;
        var z=7;
    }
    var x=4;
    sub1();
    sub2()
}
```

- znaczenie **x**, do którego odnosimy się w sub2 jest określane dynamicznie; zależy od sekwencji wywołań podprogramów.

- Najpierw sprawdza się lokalne deklaracje, a jeśli to się nie powiedzie wówczas sprawdza się deklaracje w podprogramie wywołującym dany podprogram. W przypadku niepowodzenia mamy błąd.

Rozważyć wywołanie

- sub1()
- sub2()

z funkcji sub().

Przykład: zakresy


```
1.  a : integer -- global
2.  procedure first
3.      a := 1
4.      procedure second
5.          a : integer -- local
6.          first()
7.  a := 2
8.  if read integer() > 0
9.      second()
10. else
11.     first()
12. write integer(a)
```

- Zakresy statyczne:
  wynik: 1
- Zakresy dynamiczne:
  `read_integer`>0
  `wynik:` 2
  `read_integer`<=0
  wynik: 1

Stałe

- Stała = zmienna związana z wartością tylko raz
  - Poprawienie czytelności programu i zmniejszenie ryzyka błędów
  - Deklaracje (różne języki)
    - `const` (Pascal)
    - `readonly`
    - `static` (C, C++)
    - `final` (Java)
    - `save` (Fortran)
    - `own` (Algol)

Przykład: Użycie `static` w C


```
/*
   Place into *s a new name beginning with the letter l and
   continuing with the ascii representation of an integer guaranteed
   to be distinct in each separate call. s is assumed to point to
   space large enough to hold any such name; for the short ints used
   here, seven characters suffice. l is assumed to be an upper or
   lower-case letter. sprintf 'prints' formatted output to a string.
*/
void gen_new_name(char *s, char l) {
   static short int name_nums[52];
      /* C guarantees that static local variables without explicit
      initial values are initialized as if explicitly set to zero. */
   int index = (l >= 'a' && l <= 'z') ? l-'a' : 26 + l-'A';
   name_nums[index]++;
   sprintf(s, "%c%d\0", l, name_nums[index]);
}
```

(Scott)

Wiązanie podprogramów

- Wywołania procedur o parametrach, będących procedurami (funkcjami)
- Środowisko odniesienia
- Domknięcie
- Płytkie i głębokie wiązania

Środowisko referencyjne

- Przykład

```
void sub1() {
   int a, b;
   ...              // 1
} /* end of sub1 */

void sub2() {
   int b, c;
   ...              // 2
   sub1();
} /* end of sub2 */

void main() {
   int c, d;
   ...              // 3
   sub2();
} /* end of main */
```

- Środowisko referencyjne = zbiór wszystkich zmiennych widocznych w miejscu danego polecenia, instrukcji.
- Dla programu obok:
  - 1: **a**, **b** z **sub1**, **c** z **sub2**, **d** z **main** (**c** z **main** i **b** z **sub2** są przekryte)
  - 2: **b**, **c** z **sub2**, **d** z **main** (**c** z **main** jest ukryte)
  - 3: **c**, **d** z **main**

Domknięcie

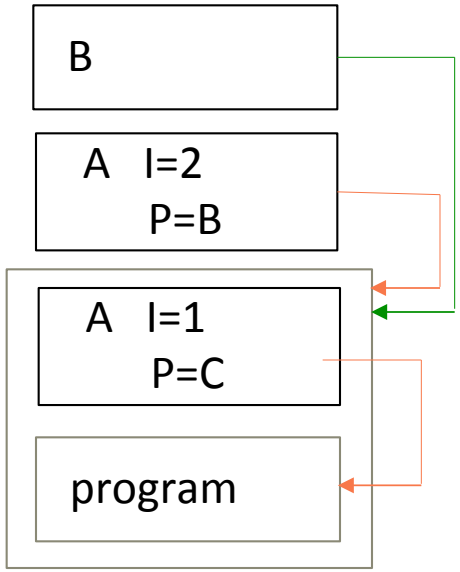
- Środowisko referencyjne + referencja do wywoływanego podprogramu
- Wiązanie płytkie
- Wiązanie głębokie

Przykład: Głębokie wiązanie w Pascalu


```
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 (* program główny *)
   A(1, C);
end.
```

W chwili wywołania B poprzez parametr formalny P istnieją dwie instancje zmiennej I. Ponieważ domknięcie P zostało utworzone przy pierwszym wywołaniu A, więc użyte będzie I z tego domknięcia. Wynikiem wykonania programu jest 1.

- Stos
  - środowiska wywołań
  - wiązania



Typy danych

Wstęp

- Fortran I – typy wbudowane (skalarne + tablice)
- COBOL – rekordy, precyzja liczb dziesiętnych
- Algol 68 – rozszerzenia dotyczące tworzenia struktur danych
- po 1980 – abstrakcyjne typy danych (interfejs + ukryte operacje)

- 1959 – LISP, listy

Wstęp

- Typy niejawnie dostarczają kontekstu, który pozwala programiście wykonywać operacje bez jawnego wprowadzania kontekstu. Np. **`a+b`** – może być dodawaniem liczb całkowitych lub dodawaniem liczb rzeczywistych; wywołanie **`new p`** oznacza rezerwację pamięci na stercie dla obiektu wskazywanego przez **`p`** – nie interesuje nas nawet wielkość tego obiektu;
- Typy nie pozwalają wykonać bezsensownych operacji w programie – nie możemy np. dodać napisu do rekordu i nie możemy obliczyć wartości **`sinus`** z napisu itp.
- W większości j.p. programista musi zadeklarować typ wszystkich obiektów w programie

Klasyfikacja typów danych

- typ danych = zbiór wartości + operacje
  - Typy proste
    - numeryczne: **`integer`**, **`float`**, **`double`**, **`real`**,...
    - boole'owskie: **`true, false | 0, 1`**
  - znaki, napisy
  - Typy wyliczeniowe
  - Typy tablicowe
  - tablice asocjacyjne
  - rekordy
  - unie
  - wskaźniki i referencje

Integer

- Java: **`byte, short, int, long`** (znakowane)
- C++, C# – integer bez znaku
- Realizacja liczby ze znakiem: jeden bit 0|1 znak
- F# i Python – nieograniczona długość liczb (Long): 232456719856483028373645L
- Reprezentacja znak – moduł
- Reprezentacja dopełnienia do 2 (większość języków programowania)
- Reprezentacja dopełnienia do 1

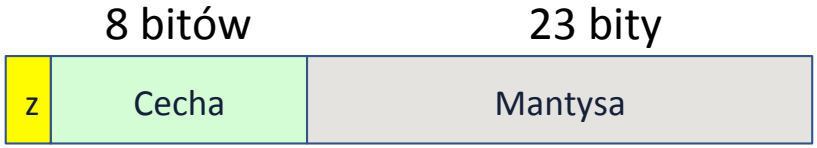
Reprezentacja **integer**

(Sebesta)

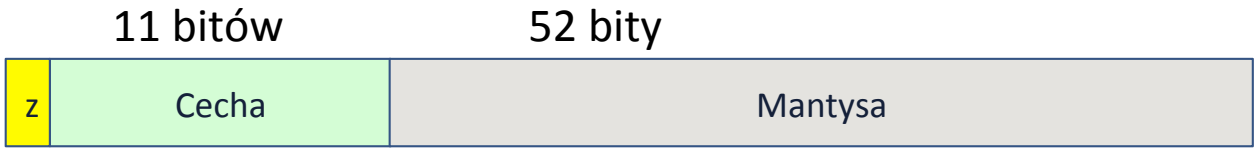
A negative integer could be stored in sign-magnitude notation, in which the sign bit is set to indicate negative and the remainder of the bit string represents the absolute value of the number. Sign-magnitude notation, however, does not lend itself to computer arithmetic. Most computers now use a notation called twos complement to store negative integers, which is convenient for addition and subtraction. In twos-complement notation, the representation of a negative integer is formed by taking the logical complement of the positive version of the number and adding one. Ones-complement notation is still used by some computers. In ones-complement notation, the negative of an integer is stored as the logical complement of its absolute value. Ones-complement notation has the disadvantage that it has two representations of zero. See any book on assembly language programming for details of integer representations.

Przykład: typ `float`

- Floating-Point Standard 754 format (patrz: internet)
- **`float`** (32 bity), (±1.18E-38, 3.4E38), (+,-,*,/)



- **`double`** (64), (±2.2E-308, 1.8E308), (+,-,*,/)



Typy booleowskie

- Wprowadzono w Algolu 60 (1960)
- exception is C89, in which numeric expressions are used as conditionals. In such expressions, all operands with nonzero values are considered true, and zero is considered false
- Although C99 and C++ have a Boolean type, they also allow numeric expressions to be used as if they were Boolean.
- Realizacja: wystarcza bit, ale zrealizowano jako bajt

Sprawdzanie typów

- równoważność typów
  - strukturalna (Algol 68, ML, Modula-3, C) – typy równoważne jeśli ich rozwinięta postać jest identyczna
  - leksykalna bazuje na deklaracjach (Java, C#, Pascal, Ada) – bardziej popularna we współczesnych językach
- zgodność typów (!)
- wnioskowanie typów (ML, Haskell cała informacja o typach jest wnioskowana)

Sprawdzanie typów

- Czy następujące typy są zgodne?

```
type r1 = record
   a: integer;
   b: integer
end;

type r2 = record
   b: integer;
   a: integer
end;
```

```
type student = record
   name, address : string
   age : integer

type school = record
   name, address : string
   age : integer

x : student;
y : school;
. . .
x := y;
```

— czy jest to błąd?

Sprawdzanie typów

- Przykład C++ (B. Stroustrup). Czy s1 i s2 są zgodne ze sobą?

```
struct s1 {int a;}
struct s2 {int a;}
...
s1 x;
s2 y=x;    // niezgodność!
...
int i=x;   // niezgodność!
```

W C++ typy zmiennych są zgodne jeśli zmienne zgłoszono w tej samej deklaracji lub takiej samej... zgodność leksykalna

Wnioskowanie typów

- podzakresy

```
type Atype = 0..20;    (* Pascal *)
     Btype = 10..20;
var a : Atype;
    b : Btype;
```

- Jaki jest typ wyrażenia **a+b**? Wynik może być w granicach od 10-40. Nie mieści się w żadnym z wymienionych typów. Czy kompilator może to stwierdzić? Czy będzie błąd wykonania? A jak jest w przypadku wyrażenia **a-b**? Dynamiczne sprawdzanie semantyki.
- Ada! Ale ... (dla przykładowych deklaracj OKREŚLONYCH wyżej)

```
function fun(i: integer) return integer is ...
...
a := b - fun(10);
```

Czy wymaga to dynamicznego sprawdzenia semantyki?

Wnioskowanie

- Coercion – wymuszenie, przymus
  Co się stanie jeśli...

```
var a : integer;
b, c : real;
          ...
    c := a + b;
```

- Wiele j. p. wymusza typ dokonując rzutowania

Wnioskowanie typu wyrażeń w C

- W C mamy przypadki wiele przymusowych rzutowań o prostych regułach:
  - **`float`** zamieniane jest na **`double`**
  - **`shortint`** i **`char`** przechodza w **`int`**
  - jeśli to konieczne traci się precyzję lewej strony podczas przypisania

Wnioskowanie

- Wymuszone rzutowania (niejawne) jest przeciwieństwem sprawdzania
- Języki Ada, Modula-2 nie stosują wymuszonych rzutowań
- C++ nadużywa rzutowań! Podobnie Fortran...
- Paradoksalnie w C, C++ mamy z jednej strony silne typowanie, a z drugiej strony wymuszone rzutowania

Typy wyliczeniowe i podzakresy

- Wprowadzone przez N. Wirtha w Pascalu (`type tydzien = (po, wt, sr, cz, pt, so, ni);`)
  Operacje: succ(), ord(), ...; mogą też być indeksami tablic;

- Pascal:
  - ```
    type odliczanie = 1..100;
         tydzien = (po,wt,sr,cz,pt,so,ni);
         dniRobocze = po..pi;
    ```
- Ada:

```
type odliczanie is new integer range 0..100;
subtype robocze is tydzien range po..pi;
```
```

Przykład: typ znakowy (char)

- Zapamiętywane w postaci kodów numerycznych
- Najczęściej: ASCII
- Alternatywnie: kody 16 bitowe (UCS-2)
  - Pierwotnie w Java
  - C#, JavaScript
- 32 bitowe kody (UCS-4)
  - Fortran 2003

Przykład: typ znakowy (char)

(Sebesta)

Character data are stored in computers as numeric codings. Traditionally, the most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters. ISO 8859-1 is another 8-bit character code, but it allows 256 different characters. Ada 95+ uses ISO 8859-1.

Because of the globalization of business and the need for computers to communicate with other computers around the world, the ASCII character set became inadequate. In response, in 1991, the Unicode Consortium published the UCS-2 standard, a 16-bit character set. This character code is often called Unicode. Unicode includes the characters from most of the world's natural languages. For example, Unicode includes the Cyrillic alphabet, as used in Serbia, and the Thai digits. The first 128 characters of Unicode are identical to those of ASCII. Java was the first widely used language to use the Unicode character set. Since then, it has found its way into JavaScript, Python, Perl, C#, and F#.

After 1991, the Unicode Consortium, in cooperation with the International Standards Organization (ISO), developed a 4-byte character code named UCS-4, or UTF-32, which is described in the ISO/IEC 10646 Standard, published in 2000.

Napisy

- Wartościami są ciągi znaków
- Problem realizacji
  - Czy są to typy proste, czy tablice?
  - Czy długość napisu jest statyczna, czy dynamiczna?
- Operacje: przypisanie, kopiowanie, porównywanie, katenacja (łączenie), pobieranie podnapisów, dopasowywanie wzorców

Przykład: operacje na napisach

- Zależnie od języka programowania
- operacja łączenia napisów (zależy od języka):
  - Java: +
  - Perl: .
  - awk: “a””b” lub ‘a’’b’
  - Fortran: //
- długość: **`len`**, **`length`**; np. Ruby: **`l=a.length`**;
- reprezentacja: znak = 1 b (ASCII); =1, 2, 3 b (UTF)
- przeszukiwanie: wyrażenia regularne (SNOBOL4 – pierwszy; Java, Perl, ..., prawie wszystkie j.p.)
- Napisy – niezmienialne, zmienialne (Java: **`String`**, **`StringBuffer`**)

Wsparcie dla napisów

- Java – klasy **`String`**, **`StringBuffer`**
- C#, Ruby – podobnie
- F# - napisy są klasą; łączenie (katenacja): +
- Biblioteki standardowe C, C++ (nieco niebezpieczne). Przykład: **`strcpy(gdzie, src).`** Jeśli długość **`gdzie`** jest 10, a **`src`** ma długość **`30`** to **`strcpy`** kopiuje napis **`src`** poza miejsce przeznaczenia!
- Napisy i tablice (przykład)

Dopasowywanie wzorców

(Sebesta)

Perl, JavaScript, Ruby, and PHP include built-in pattern-matching operations. In these languages, the pattern-matching expressions are somewhat loosely based on mathematical regular expressions. In fact, they are often called regular expressions . They evolved from the early UNIX line editor, ed, to become part of the UNIX shell languages. Eventually, they grew to their current complex form. There is at least one complete book on this kind of patternmatching expressions (Friedl, 2006). Consider the following pattern expression:

```
/[A-Za-z][A-Za-z\d]+/
```

This pattern matches (or describes) the typical name form in programming languages.

Next, consider the following pattern expression:

```
/\d+\.?\d*|\.\d+/
```

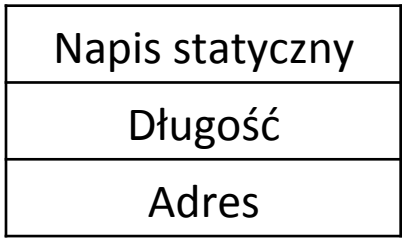
This pattern matches numeric literals.

SNOBOL 4 was the first widely known language to support pattern matching.

SNOBOL

SNOBOL (pronounced “snowball”; Griswold et al., 1971) was designed in the early 1960s by three people at Bell Laboratories: D. J. Farber, R. E. Griswold, and I. P. Polonsky (Farber et al., 1964). It was designed specifically for text processing. The heart of SNOBOL is a collection of powerful operations for string pattern matching. One of the early applications of SNOBOL was for writing text editors. Because the dynamic nature of SNOBOL makes it slower than alternative languages, it is no longer used for such programs. However, SNOBOL is still a live and supported language that is used for a variety of text-processing tasks in several different application areas.

Realizacja napisów



Opis napisu statycznego
(compile-time descriptor)

Opis napisu dynamicznego
(run-time descriptor)

Pamięć dla napisów dynamicznych

Dynamic length strings require more complex storage management. The length of a string, and therefore the storage to which it is bound, must grow and shrink dynamically.

There are three approaches to supporting the dynamic allocation and deallocation that is required for dynamic length strings. First, strings can be stored in a linked list, so that when a string grows, the newly required cells can come from anywhere in the heap. The drawbacks to this method are the extra storage occupied by the links in the list representation and the necessary complexity of string operations.

The second approach is to store strings as arrays of pointers to individual characters allocated in the heap. This method still uses extra memory, but string processing can be faster than with the linked-list approach.

The third alternative is to store complete strings in adjacent storage cells. The problem with this method arises when a string grows: How can storage that is adjacent to the existing cells continue to be allocated for the string variable? Frequently, such storage is not available. Instead, a new area of memory is found that can store the complete new string, and the old part is moved to this area. Then, the memory cells used for the old string are deallocated. This latter approach is the one typically used. (Sebesta)

cdn ...