Kifejezések

Kozsik Tamás

December 11, 2016

Kifejezés versus utasítás

C/C++:

- kifejezés plusz pontosvessző: utasítás
- kiértékeli a kifejezést
- jellemzően: mellékhatása is van
- például: értékadás

```
n = 5;
n++;
```

Ada:

- ilyen nincs, pl. az értékadás is egy utasítás

```
N := 5;
N := N + 1;
```

Mellékhatásos kifejezés

- Nem tiszta (impure)
- Például mellékhatásos operátorral

```
n = 1
```

-
  - értéke: `1`
  - mellékhatása: `n`-nek értékül adja az `1`-et
- Vagy mellékhatásos függvénnyel

Eljárás versus függvény

Szerencsés, ha egy nyelvben megkülönböztetjük a kettőt

- Mellékhatás: eljárás
- Függvény: legyen tiszta
  - védett egységeknél feltételezzük is ezt (író-olvasó szinkr.)
  - az Ada nem kényszeríti ki, hogy tiszta legyen
  - sőt, az Ada megengedi az `out` paramétert is függvényben

Kifejezések

- Lexika
- Szintaktika
- Szemantika

Lexika

- azonosítók (változók, típusok, programegységek, kivételek nevei)
- literálok
- operátorok, pl. +
  - Adában szövegesek is vannak, pl.: `abs`
  - sőt, két szóból álló is, pl.: `and then`
- zárójelek
  - Adában csak egyféle: `()`
  - C++-ban négyféle: `()`, `[]`, {}, <>
- aposztróf, vessző, pont...

```
Integer'Max(13,42)
Integer'(42) .. Today.Year
```

Lexika: literálok

- karakter, sztring, egész szám, valós szám, felsorolási típus

```
'c'
"hello"
12E3
-124.43E5
True
```

- túlterhelt literálok (származtatott és felsorolási típusoknál)
- számoknál előjel, exponens, számrendszer, méret (C: `1L`)
- tagolás az olvashatóság érdekében (pl. `1_000_000`)
- például Adában: kettes számrendszerben, tagolva 8 számjegy, hármas számrendszerben egész és valós szám exponenssel

```
2#1111_0011#        3#1111#E4        3#1000.1#E1
```

Szintaktika

- helyes zárójelezés
- operátorok arítása
  - unáris, pl. `-`
  - bináris, pl. `*`
  - ternáris
    - C++: `x>0 ? 2 : 3`
    - Ada, Haskell: `(if X > 0 then 2 else 3)`
  - *n*-áris: `()` (C++ függvényalk. op.), `[]` (Eiffel bracket-op.)
- operátorok fixitása
  - prefix, pl. `not`, prefix `++`
  - postfix, pl. postfix `++`
  - infix, pl. `*`
  - mixfix, pl. `(if . then . else .`, `.?.:.` és `(...)`

```
n = 4;
n++;    // értéke 4, mellékhatás: n vált. 5-re
++n;    // értéke 6, mellékhatás: n vált. 6-ra
```

Szintaktika: aggregátumok

- C-ben tömb aggregátum: {1,3,5,7}
- Adában tömb és rekord aggregátum
  - pozícionális
  - névvel jelölt
  - kevert

```
(1,3,5,7)
(1..5 => 2, 6|9 => 1, others => 0)
(1,2,3, others => 0)
(Re => 0.0, Im => 1.0)
```

Szemantika

- precedencia
  - pl. `1+2*3 = 1+(2*3) = 7`, `(1+2)*3 = 9`
  - redundáns zárójelezés segítheti a kód olvasását
- asszociativitás
  - azonos precedenciaszintű operátorok zárójelezése
  - pl. `4/2*5 = (4/2)*5 = 10` (balasszociatív operátorok)
  - pl. `x = y = 5` jelentése C-ben: `x = (y = 5)`
    - azaz `y`-ba 5 kerül, az `y=5` kifejezés értéke 5, `x`-be 5 kerül
    - jobbasszociatív operátor
  - hatványozás sok nyelvben jobbasszociatív (pl. Haskell), de az Adában nem (Adában minden balasszociatív)
- operandusok kiértékelési sorrendje
- tisztaság, mellékhatások
- lustaság, mohóság

Kiértékelés problémái

- véges értéktartomány
  - számábrázolás: véges sok biten
    - túl- és alulcsordulás
    - Ada: `Constraint_Error`
  - "nagy számok", pl. Haskell `Integer` is valójában véges
- nem terminálló számítások

A matematikusokhoz képest nehéz dolga van a programozóknak!

Két egész szám átlaga?

Túlcsordulás nélkül számítsuk ki: `(A+B)/2`

```
function Avg( A, B: Integer ) return Integer is
    Half_A: Integer := A/2;  Half_B: Integer := B/2;
begin
    if A >= 0 and B >= 0 then    -- A+B may be too large
        if Half_A + Half_A < A and Half_B + Half_B < B
            then return Half_A + Half_B + 1;
            else return Half_A + Half_B;
        end if;
    elsif A < 0 and B < 0 then   -- A+B may be too small
        if Half_A + Half_A > A and Half_B + Half_B > B
            then return Half_A + Half_B - 1;
            else return Half_A + Half_B;
        end if;
    else return (A + B) / 2;     -- different signums, safe!
    end if;
end Avg;
```

Számábrázolás

- Kettes komplemens egészek ábrázolására
  - értéktartomány n bit esetén: -2^{n-1} .. $2^{n-1}-1$
  - pl. $n = 8$ esetén `-128..127`
  - több negatív érték, mint pozitív, tehát az unáris `"-"` művelet is hibás eredményt adhat: `-Integer'First`
- Lebegőpontos ábrázolás valós számokhoz
  - $\approx$ racionális számok egy részhalmaza
  - leggyakrabban IEEE 754 szabvány szerint `https://en.wikipedia.org/wiki/IEEE_floating_point`
  - az Ada szabályai segítenek az elvárt pontosság megadásában

```
type Probability is digits 8 range 0.0 .. 1.0;
```

- Fixpontos ábrázolás

Fixpontos számábrázolás

- bináris fixpontos: fix kettedespont
  - pl. 1/8 voltos pontosság

```
type Volt is delta 0.125 range 0.0 .. 255.0;
```

- decimális fixpontos: fix tizedespont
  - pl. euro és cent

```
type Euro is delta 0.01 digits 15;
```

Tisztaság, mellékhatások

- mellékhatásos operátorok és függvények miatt
- C++-ban értékadó operátorok
  - szinte mindig a mellékhatása miatt használjuk
  - pl. =, +=, ++ (prefix és postfix)
- vannak idiómák, melyek a mellékhatásos operátorra építenek

```
// Java code reads input line-by-line
BufferedReader in = ...
String str;
while( (str=in.readLine()) != null ){ ... }
```

- a mellékhatás sérti a hivatkozásihely-függetlenséget (referential transparency)
- ökölszabály: kerüljük a mellékhatásos függvényeket!
- ökölszabály: ne legyen egy kifejezésben több is!

Operandusok kiértékelési sorrendje

- Egyes nyelvekben (pl. Java, C#) kötött: balról jobbra
  - pl. α op β kiértékelése:
    - először α
    - azután β
    - végül az op művelet a két kiértékelt operanduson
  - hasonlóan $f(\alpha, \beta, \gamma)$ kiértékelése
  - ez más, mint a precedencia és asszociativitás
- Sok nyelvben (pl. Ada, C++) a fordítóprogram dönti el, azaz nem feltétlenül egyértelmű, milyen eredményt kapunk
  - több lehetséges eredmény a mellékhatások miatt
  - a hatékonyságot befolyásolja, ezért ez egy optimalizációs módszer a fordító számára!
- Jobb, ha kerüljük az olyan kifejezéseket, ahol ez számít!

```
c++ + ++c
```

Lustaság versus mohóság

- Lusta nyelvekben call-by-need (igény szerinti paraméterátadás)
  - pl. Haskell
  - argumentumok nem értékelődnek ki a függvény meghívása előtt
- Operátorok lustasága
  - Nem (feltétlenül) értékel ki minden operandust
  - Mohó nyelvben is lehetnek lusta operátorok, pl. C++
    - `&&` és || rövidzár logikai operátorok, pl. α `&&` β kiértékeléséhez nem értékeli ki β-t, ha α hamis
    - `?:` operátor legfeljebb két operandust a háromból
  - Lehet logikai operátor lusta és mohó változatban is
    - Ada: `and`, `or`, `and then`, `or else`
    - Java, C#: `&`, |, `&&`, ||
  - Van, hogy csak lusta változat létezik (C++)
  - Van, hogy csak mohó (Standard Pascal)

Lehet más a lusta és a mohó?

- A logikában nem. De a programozóknak nehezebb dolguk van, mint a matematikusoknak.
- A különbség lehetséges okai, pl. $\alpha \wedge \beta$ esetben:
  - lehet mellékhatás a β részkifejezésben;
  - lehet, hogy β kiértékelése nem mindig terminál;
  - lehet, hogy β kiértékelése néha kivételt vált ki.
    - idióma:

```
while I <= T'Last and then T(I) /= 0 loop
    I := I+1;
end loop;
```

Lusta és mohó művelettábla

Jelölje $\uparrow$, $\downarrow$, $\bot$ és ∞ a négy lehetséges eredményt egy logikai kifejezés kiértékeléséhez: igaz, hamis, kivétel, nem terminálló számítás. Az $\alpha \wedge \beta$ kifejezés értéke az α és β értékének függvényében (a mellékhatásoktól itt eltekintünk):

α and then β	$\beta = \uparrow$	$\beta = \downarrow$	$\beta = \bot$	$\beta = \infty$
$\alpha = \uparrow$	$\uparrow$	$\downarrow$	$\bot$	∞
$\alpha = \downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$\alpha = \bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\alpha = \infty$	∞	∞	∞	∞

α and β	$\beta = \uparrow$	$\beta = \downarrow$	$\beta = \bot$	$\beta = \infty$
$\alpha = \uparrow$	$\uparrow$	$\downarrow$	$\bot$	∞
$\alpha = \downarrow$	$\downarrow$	$\downarrow$	$\bot$	∞
$\alpha = \bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\alpha = \infty$	∞	∞	∞	∞