Funkcionális programozási nyelvek fordítása

Fordítóprogramok előadás (A,C,T szakirány)

Funkcionális programozási nyelvek

- programok futása:
  - funkcionális eset: függvények kiszámítása
  - imperatív eset: állapotváltozások sorozata
- tiszta függvény: olyan függvény, amelynek nincs mellékhatása
- tisztán funkcionális nyelv: csak tiszta függvények írhatók benne
  - akár imperatív nyelveken is lehet „tisztán funkcionális" stílusban programozni
  - az imperatív nyelvekben megszokott változók nem léteznek
- példák: Haskell, Clean, Erlang, OCaml, F#, Lisp, Scheme, ...

Funkcionális nyelvek sajátosságai

- (általában) nagyon kifinomult a típusrendszerük
  - többalakúság (polimorf típusrendszer)
  - típusellenőrzés helyett típuslevezetés
  - típusosztályok
- a függvények „teljes jogú tagjai a nyelvnek" („first class citizens"):
  - függvényeket is lehet paraméterként átadni
  - függvény is lehet visszatérési érték
- lusta (lazy) / szigorú (strict) kiértékelés:
  - lusta: csak akkor értékel ki egy kifejezést, amikor arra valóban szükség van
  - szigorú: minden paramétert kiértékel még a függvényhívás előtt (az imperatív nyelvek szigorúak)

Funkcionális nyelvek fordítása

- lexikális / szintaktikus / szemantikus elemzés eredménye egy transzformált program, amely
  - típusozott
  - néhány egyszerű nyelvi konstrukcióból épül fel
- ezen a transzformált programon optimalizációs lépéseket hajtunk végre
- kódgenerálás: általában C-re
  - így nem kell újraírni a C fordítókban meglévő sok kifinomult imperatív jellegű optimalizációt
- a végrehajtáshoz szükséges egy futtatókörnyezet is

A margó szabály

- az utasítások végét és a blokkszerkezetet a sorok behúzása definiálja

Margó szabállyal

```
f x y = a + b where
  a =
    x * y
  b = x - y
g x = 2 * x
```

Explicit jelöléssel

```
f x y = a + b where {
  a =
    x * y;
  b = x - y; }
g x = 2 * x;
```


A margó szabály megvalósítása

- a lexikális elemző feladata
- minden lexikális elemhez megjegyezzük, hogy hányadik oszlopban van az első karaktere (a fehér szóközöket is figyelembe kell venni)
- egy verembe beletesszük a legelső utasítás behúzását
- bizonyos kulcsszavak után (Haskellben pl. `where`, `of`) beszúrunk egy „{" tokent és a következő token behúzását a verembe tesszük
- minden sortörésnél meg kell vizsgálni az új sor behúzást:
  - ha egyenlő a verem tetején lévő értékkel, akkor beszúrunk egy „;"-t
  - ha nagyobb a behúzás, akkor nincs teendő
  - ha kisebb a behúzás, akkor beszúrunk egy „;"-t, majd addig szúrunk be egy-egy „}"-t és veszünk ki egy-egy értéket a veremből, amíg a verem tetején nagyobb érték van az aktuális sor behúzásánál

Polimorf típusok

- A típusok lehetnek „több alakúak”:
  - pl. egészek listája, karakterek listája, listák listája...

```
[1,2,3] :: [Int]
['x'] :: [Char]
[[5],[1,2]] :: [[Int]]
[] :: [a]
[[]] :: [[b]]
```

- A példában az *a* és *b* típusváltozók.
- Típuslevezetéskor nem a típusok *egyenlőségét* kell vizsgálni, hanem az *egyesíthetőségét*.
  - Egyesíthetőség (kb.): A típusváltozók helyettesítésével azonos alakúra hozható-e a két típus?

Példák típuslevezetésre

- A `[['x','y'],[]]` kifejezés típusát szeretnénk levezetni.

```
'x' :: Char
'y' :: Char
```

- *Char* és *Char* egyesíthetőek és az eredmény *Char*, ezért:

```
['x','y'] :: [Char]
```

- Az üres lista típusa:

```
[] :: [a]
```

- *[Char]* és *[a]* egyesíthetőek az $a \rightarrow Char$ helyettesítéssel, ezért:

```
[['x','y'],[]] :: [[Char]]
```

- A `[['x','y'],'z']` kifejezés esetén egyesíteni kellene a *[Char]* és *Char* típusokat, ami nem lehetséges. $\Rightarrow$ Típushiba!

Szigorú és lusta kiértékelés

- C-ben kiértékelődik mindkét paraméter:

```
int f( int x, int y )
{
  return x + 1;
}
int a = 4321;
int b = f( a/2, a*515341 )
```

- Haskellben csak az első paraméter értékelődik ki:

```
f x y = x + 1
a = 4321
b = f (a/2) (a*515341)
```

- Nem csak hatékonysági kérdés!
  - `f( 5, a/0 )` futási idejű hiba C-ben
  - `f 5 (a/0)` hiba nélkül lefut, és 6-ot ad Haskellben

Függvények mint paraméterek vagy visszatérési értékek

- A map egy lista minden elemére végrehajtja az f függvényt:

```
map f [] = []
map f (első:többi) = (f első) : (map f többi)
```

- Az addToAll egy olyan függvényt ad eredményül, ami egy lista minden elemét növeli n-nel:

```
addToAll n = map (n+)
```

- A függvények tehát adatokként kezelhetőek, azaz teljes jogú tagjai a nyelvnek.
- Ez C-ben elég nehézkesen, függvénypointerekkel oldható csak meg.

A lusta kiértékelés és a függvények teljes jogú tagságának megvalósítása

- Az `f (a/2) (a*515341)` hívásban „kiértékeletlenül” kell átadni a paramétereket, hogy az `f` függvény dönthesse el, melyikre van ténylegesen szüksége.

```
f x y = x + 1
```

- A `map` függvény első paraméterében egy függvényt kell átadnunk, ami szintén egy „kiértékeletlen” kifejezés.
- Az `addToAll` függvény eredménye egy függvény, amit még alkalmaznunk kell egy listára, hogy kiértékelhető legyen.
- Hogyan lehet kiértékeletlen kifejezéseket kezelni?

A „kiértékeletlen kifejezések” kezelése

- Készítsünk egy objektumot, amely tartalmaz
  - egy pointert a függvényre
  - további pointereket a függvény meglévő paramétereire
- Az ilyen objektumok (closure) átadhatók függvényeknek, visszaadhatók visszatérési értékként.
- Ha mindegyik paraméter rendelkezésre áll egy closure-ben és szükségessé válik a kiértékelése, akkor elvégezhető a művelet.

Gráfátírás

- Egy closure-ben a paraméterek is és a függvény is lehet további kiértékeletlen kifejezés.
- Ezért egy teljes kifejezés ábrázolható egy fával, általánosabb esetben egy gráffal.
- Kódgenerálás a modern lusta kiértékelésű funkcionális nyelveken írt programokhoz:
  - a programban lévő main függvényből felépítünk egy gráfot
  - a programban lévő függvénydefiníciókból egy-egy gráfátíró szabályt készítünk
- A programok lefutása:
  - a futtatórendszer minden lépésben kiválaszt egy megfelelő részt a gráfból
  - átírja valamelyik szabály alapján
  - ezt addig ismételgeti, amíg
    - a gráf egy adattá egyszerűsödött (ez a végeredmény)
    - vagy már nem egyszerűsíthető tovább