|  |  |
| --- | --- |
|  | **EÖTVÖS LORÁND TUDOMÁNYEGYETEM**  Informatikai Kar  Programozási Nyelvek és Fordítóprogramok Tanszék |

**Praktikus programozás SKI kombinátor kalkulussal**

**Témavezető: Készítette:**

Kaposi Ambrus Fülöp Olivér - PG8SJ5

egyetemi docens programtervező informatikus MSc szak

**Budapest, 2023**

# 1. Tartalomjegyzék

[1. Tartalomjegyzék 1](#_Toc145283749)

[2. Bevezetés 2](#_Toc145283750)

[3. A kutatás ismertetése 3](#_Toc145283751)

[3. 1. SKI kombinátor kalkulus 3](#_Toc145283752)

[3. 1. 1. Története 3](#_Toc145283753)

[3. 1. 2. Elemei és működése 3](#_Toc145283754)

[3. 1. 3. Kombinátorok használata a gyakorlatban 5](#_Toc145283755)

[3. 1. 4. Előnyei, hátrányai 6](#_Toc145283756)

[3. 2. Típusos SKI 7](#_Toc145283757)

[3. 2. 1. Típusok 7](#_Toc145283758)

[3. 2. 2. Termek 11](#_Toc145283759)

[3. 2. 3. Parser (Lexikális elemző) 11](#_Toc145283760)

[3. 2. 4. Típusellenőrző 11](#_Toc145283761)

[3. 2. 5. Kiértékelő 12](#_Toc145283762)

[3. 2. 6. Sekély implementáció 12](#_Toc145283763)

[3. 3. Példák 12](#_Toc145283764)

[4. Összegzés/Konklúzió 12](#_Toc145283765)

[5. Irodalomjegyzék 12](#_Toc145283766)

# 2. Bevezetés

Diplomamunkámként egy gyakorlatban is használható, az SKI kombinátorokra épülő típusos kifejezésnyelvet készítettem, amely nem tartalmaz változókat. A nyelvhez parsert, típusellenőrzőt és egy kiértékelőt is implementáltam Java nyelven.

Az elkészült nyelvben példaként egyszerű programokat adtam meg és értékeltem ki és azt vizsgáltam, hogy egy ilyen nyelvben milyen módon lehet nagyobb programokat minél egyszerűbben és olvashatóan megadni. A kutatásban felhasznált módszerek elsősorban a típusparaméterek használata és a típuskikövetkeztetés, valamint összehasonlítottam a nyelv két különböző, mély és sekély implementációját.

# 3. A kutatás ismertetése

## 3. 1. SKI kombinátor kalkulus

### 3. 1. 1. Története

Az SKI kombinátor kalkulus az elméleti számítástudomány és matematikai logika egyik alapvető számítási módszere. Az 1920-as években Moses Schönfinkel mutatta be és később az 1930-as években Haskell Curry kiegészítette.

Történelmileg, az SKI kalkulus eredete David Hilbert munkájára vezethető vissza. Egy formális rendszert szeretett volna létrehozni matematikai bizonyítások kifejezésére, amely a lehető legkevesebb axiómát és következtetési szabályt tartalmazza. Schönfinkelnek Hilbert munkája adta az ihletet az SKI kalkulus elkészítéséhez, amellyel változók használata nélkül tudott függvényeket kifejezni. Az SKI elnevezés a három alapkombinátorból következik: S, K és I.

### 3. 1. 2. Elemei és működése

A kombinátor egy magasabb rendű függvény, amely csak függvény alkalmazásokat és korábban meghatározott kombinátorokat használ az eredmény kiszámításához. Az SKI kalkulus primitív kombinátorai az S, K és I, melyekkel az alapvető müveletek végezhetőek el. E három alapkombinátor kombinációjával és a paramétereken való alkalmazásával, komplex számításokat lehet kifejezni és kiértékelni. A fő ötlet, hogy bármely kiszámítható függvény reprezentálható a három alapkombinátorral, mivel a lambda-kalkulus esetén a Church-Rosser tétel garantálja, hogy a számítások eredményei nem függenek az alkalmazott redukciós lépések sorrendjétől és hogy a lambda-kalkulus és a kombinátor logika algebrailag ekvivalens egymással, tehát bármely lambda kifejezésnek van egy ekvivalens SKI kombinátor kifejezése, és fordítva, bármely SKI kombinátor kifejezésnek van egy ekvivalens lambda kifejezése.

Az S kombinátor függvények kompozícióját és alkalmazását fejezi ki. Három paramétert vár, az első paramétert alkalmazza a harmadikra, majd ezt alkalmazza a második és a harmadik paraméterek alkalmazásának eredményére.

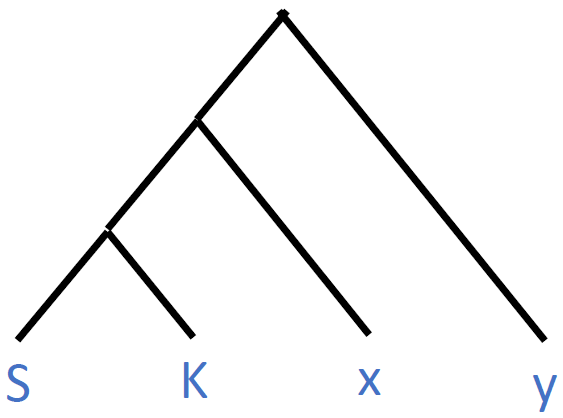
A példákban az egyes termek közötti szóközök a függvényalkalmazás műveletét jelölik.

A K kombinátor a konstans függvényként viselkedik. A két paramétere közül mindig az elsővel tér vissza, a másodikat figyelmen kívül hagyja.

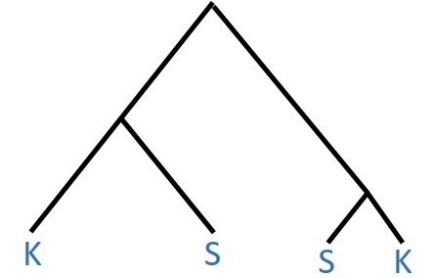
Az I kombinátor pedig az identitás függvényt reprezentálja, vagyis egyszerűen csak a kapott paraméterével tér vissza.

Az SKI kalkulus nem a minimális rendszer, ami megegyezik a lambda-kalkulus számítási képességeivel, mivel az I kombinátor kifejezhető az S K K vagy S K S kifejezésekkel. I kombinátort akár el is lehetne hagyni, csak kényelemből használjuk és azért, hogy a kifejezések rövidebbek legyenek.

Az SKI kifejezések bináris fákként reprezentálhatóak, amelynek a gyökere és csúcsai az egyes függvény alkalmazások, levelei pedig a kombinátorok és paramétereik. A közös csúcsokhoz tartozó részfák az írásos formában bezárójelezhetőek, de az olvashatóság kedvéért csak abban az esetben írjuk ki a zárójeleket, ha azok tényleg szükségesek. Zárójelek nélkül a kifejezések bal asszociatívak, tehát az kifejezést -ként kell értelmezni.



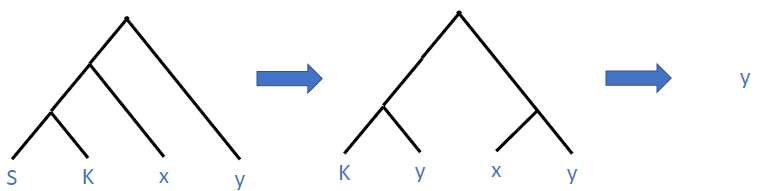
Például a helyett elég csak -t írni.

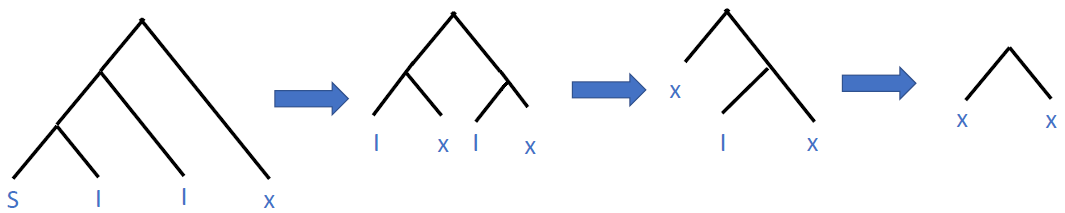


Az egyes kombinátorokon csak akkor lehet átírási szabályokat alkalmazni, ha azok a megfelelő számú paramétert kapták, S: 3 db., K: 2 db. és I: 1 db.

Például a következő kifejezések esetén nem tudunk átírási szabályt alkalmazni: vagy , de a esetben már igen: . Ha egy kombinátornak a szükségesnél több paraméter áll rendelkezésére, akkor csak a kellő számút használja fel az átírási szabály alkalmazásához, a maradék paramétereket pedig helyben hagyja.

Néhány további példa az SKI kifejezések kiértékelésére az átírási szabályok használatával:





### 3. 1. 3. Kombinátorok használata a gyakorlatban

Az SKI kalkulus tulajdonságai már jóval az első számítógép megjelenése előtt ismertek voltak. Szorosan kapcsolódik a lambda-kalkulushoz és funkcionális programozási nyelvek alapjául szolgál.

* Az első kombinátor alapú programozási nyelv az APL volt, az 1960-as évekből. Arra lett tervezve, hogy műveletsorozatokat lehessen definiálni nagy méretű adathalmazokra. Az alapvető adattípusa a többdimenziós tömb volt. Hatással volt későbbi programozási nyelvekre, többek között a MatLab-ra és MapReduce-ra.
* NumPy: egy kiegészítő csomag a Python programozási nyelvhez, főleg többdimenziós tömbök műveleteinek elvégzésére, kombinátorok segítségével.
* FP (Functional Programming) egy kombinátor alapú funkcionális nyelv.
* MapReduce: a Google által készített kombinátor nyelv big data feldolgozására, párhuzamos, elosztott számítási algoritmusok használatával.
* A Haskell fordítóprogramja, a GHC bizonyos esetekben a kódot egy kombinátor alapú formára fordítja optimalizáció céljából.
* Az Ulambda egy majdnem tisztán funkcionális programozási nyelv. Kombinátor logikán alapszik, egy lambda operátorok és szabad változók nélküli kifejezés rendszer.
* Iota és Jot: két nagyon minimalista formális rendszerrel rendelkező nyelv, úgy tervezve, hogy még egyszerűbb legyen, mint az ismertebb lambda és SKI kombinátor kalkulusok.

### 3. 1. 4. Előnyei, hátrányai

Előnyei:

* A kombinátor kalkulus a legegyszerűbb számítási formák közé tartozik.
* Nem használ változókat és nem kell foglalkozni a láthatósággal, névütközéssel és átnevezéssel
* Jól használható párhuzamos és elosztott tömeges adatműveletekhez, általában gyűjteményeken/tömbökön, pl.:
  + Egy függvény alkalmazása egy adathalmaz minden elemére
  + Egy adathalmaz egyetlen értékké redukálása egy asszociatív művelettel
  + Mátrix transzponálás

A párhuzamos/elosztott műveleteknél a változók használata nehézséget okozhat.

Hátrányai:

* Bár a kombinátorok általánosan alkalmazhatók, nem minden probléma megoldására alkalmasak hatékonyan. Bizonyos típusú problémákhoz más programozási módszerek lehetnek jobban illeszkedőek. Például nem használható szekvenciális számítási modellként.
* Néhány esetben a kombinátor alapú megoldások lassabbak vagy kevésbé hatékonyak az imperatív vagy objektumorientált megközelítéssel szemben. Számítógépeink memóriával és belső adattárolóval rendelkeznek, ezek a tulajdonságok a változókat használó programozási nyelveknek hasznosak.
* A kombinátor kalkulussal megfogalmazott kifejezések gyakran bonyolultak és nehezen olvashatóak lehetnek.

## 3. 2. Típusos SKI

Ebben a fejezetben az általam elkészített kifejezésnyelvet mutatom be, a Típusos SKI-t. Ezzel a nyelvvel főleg olyan kifejezéseket lehet megadni, amelyek az S, K és I kombinátorokból állnak, de a nyelv rendelkezik még néhány további beépített termmel is. A termek szintén kombinátorokként viselkednek és a kifejezésekben nincs lehetőség a változók használatára. A nyelv egy egyszerű típusrendszerrel is rendelkezik, ami biztosítja, hogy a kiértékelés csak helyesen típusozott utasításokra lesz alkalmazva. A Típusos SKI-ban megadott kifejezéseket először egy Parser olvassa be és elkészíti belőle a részben már típusos szintaxisfát. A szintaxisfában a típusellenőrző megpróbálja kikövetkeztetni a hiányzó típusokat, ha szükséges és elvégzi a típusellenőrzést. Végül az így elkészült jól típusozott fában a kiértékelő végrehajtja a lehetséges kombinátor alkalmazásokat és visszatér a végeredménnyel.

### 3. 2. 1. Típusok

A Típusos SKI kifejezések kiértékelésekor muszáj, hogy a kifejezés minden tagjának a típusa egyértelmű legyen. A nyelv a következő típusokat támogatja:

**Nat**: A természetes számok típusa. Mivel ez a nyelv kombinátorokat használ és a benne leírt kifejezéseket gyakran egyszerűbb először lambda-kalkulussal megadni, majd átírni kombinátorokká, ezért a számok reprezentációja Church számokkal történik.

A Church számokat Alonzo Church-ről nevezték el, aki először kódolt el adatokat lambda-kalkulus segítségével. Ez a módszer nagyon hasonlít a természetes számok funkcionális ábrázolásához, ahol adott egy természetes szám 0 és egy függvény, ami a paraméteréül kapott szám rákövetkezőjét adja vissza. A Church számok ennek a kiterjesztése. Minden Church szám egy függvény két paraméterrel:

Az első paraméter , az alkalmazandó ,,rákövetkezés’’ függvény. A második paraméter , az érték, ami a nullát jelöli. Így a 0 Church számokkal kifejezve:

Itt az bármilyen függvény lehet, mert nem lesz alkalmazva, a kifejezés -szel, a 0-át jelölő értékkel tér vissza. Az 1-et kódoló Church szám pedig pontosan 1-szer alkalmazza a rákövetkezési függvényt -en:

A Típusos SKI esetében a nulla értéket a ZERO term jelöli, a számok rákövetkezőjét előállító függvényt pedig a Succ term.

Például:

|  |  |  |
| --- | --- | --- |
| **Természetes számok** | **Church számok** | **Típusos SKI kifejezések** |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
| n |  |  |

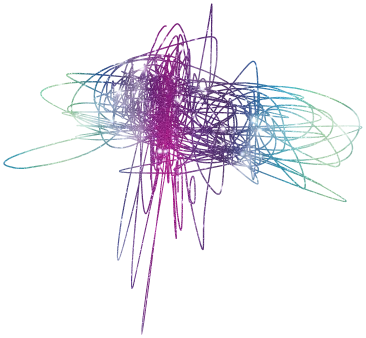
Egyszerűbb aritmetikai műveletek, mint összeadás és szorzás definiálhatóak a Church számokhoz, de azt be kell látni, hogy az eredmény értelmezéséhez mindig meg kell számolni, hogy a rákövetkezési függvény hányszor lett alkalmazva. Ez a fajta számábrázolás hatással van a teljesítményre is, mert ha hozzá szeretnénk férni egy Nat típusú értékhez, a művelet költsége helyett lesz, ahol a rákövetkezési függvény alkalmazásainak száma.

Példa az összeadásra:



* Church kódolással: Azt lehet megfigyelni, hogy a Church számok mindent a 0-hoz képest csinálnak. 1-gyel több mint , és 1-gyel több mint , ezért az 1-et jelenti -hoz képest és szintén 1-et jelent -hoz képest. Ha Church számokkal szeretnénk összeadni 3-at és 4-eat, akkor egy új Church számot kell létrehoznunk, úgy, hogy a két összeg közül az egyiknek vesszük a Church szám megfelelőjét (pl.: 3 = ) és ebben a 0 helyére beírjuk a másik összeg Church szám megfelelőjét (4 = ).

egy függvény két paraméterrel, ami -at alkalmazza a rákövetkezési függvényen, majd az eredményt alkalmazza -en, ami itt a 0-át helyettesíti.





Így tehát az M és N Church számokat összeadó függvény a következő:

* A Típusos SKI-hoz a legkönnyebben a fenti lambda kifejezés átírásával adható meg.

A kifejezés és a benne szereplő Rec és C kombinátorok működése később, a termek ismertetésénél lesz elmagyarázva.

**Bool**: A logikai értékek típusa. A Típusos SKI-nak mindkét logikai értékre van beépített termje, True – igaz és False – hamis, viszont nem tartalmazza a logikai aritmetikai műveleteket, de természetesen kombinátorok segítségével definiálni lehet őket.

**LP1.LP2.ITE P1 (ITE P2 True False) False 🡪 átírás**

A típus nélküli SKI kalkulus logikai műveletei jóval egyszerűbben kifejezhetőek, mint a típusosé, de ezeket nem használhatjuk a Típusos SKI-ban a típusrendszer megszorításai miatt. Itt a logikai értékek reprezentációja a Church kódoláshoz kötődik:

Néhány logikai művelet a típus nélküli SKI esetében:

* AND, postfix operátor:

Használata:

* OR, infix operátor:

Használata:

* NOT, postfix operátor:

Használata:

**List**: A listák típusa. A Típusos SKI-ban a listák reprezentációja hasonlít az egész számokéhoz. A különbség az, hogy a 0 elem helyett az üres lista konstruktora használható, aminek a jelölése és a rákövetkezés függvény helyett egy konstrukciós függvényt használunk a további listaelemek képzéséhez, ennek a jelölése . A Típusos SKI-ban a listák nem módosíthatóak, vagyis nem lehet elemeket ki és betenni, helyette a segítségével lehet új listát készíteni egy adott elemből és listából. Például egy egész számokat tartalmazó lista, amely az 1-es számot tartalmazza:

Egész számok listáit tartalmazó lista :

**Function**: A függvény típus. A legtöbb kombinátor függvény típusú, hiszen egy vagy több paramétert felhasználva készít egy új kifejezést. Például a korábban már említett kombinátor függvény típusú:

A Típusos SKI függvényei csak egy paraméteresek, ezért a több paraméteres kombinátorok típusa az úgynevezett Curry-zés módszerrel adhatók meg. A Curry-zés azt jelenti, hogy egy több argumentumot elfogadó függvényt függvények sorozatává alakítunk, amik csak egy-egy argumentumot kaphatnak. Például a kombinátor típusa , így a kifejezés típusa pedig , ami Curry-zéssel lesz. Az első paraméter alkalmazása után a függvénnyel tér vissza, amire alkalmazni lehet a második paramétert is.

**Str**: A String literálok típusa. A nyelvben konstans literálokat adhatunk meg, viszont a nyelv nem tartalmaz beépített műveleteket a literálok kezelésére a ki íratáson kívül, vagyis nem tudjuk darabolni/összefűzni őket. Azért került a nyelvbe, hogy változatosabbá tegye azt és több lehetőség legyen példák bemutatására.

**Unknown**: Az ismeretlen típus, ami akkor jön létre, ha egy kifejezésben nem definiáltuk explicit módon minden kombinátor típusát vagy ha a kontextusból nem lehet kikövetkeztetni. A típusellenőrző megpróbál minden ismeretlen típust kikövetkeztetni, ha ez nem sikerül, akkor a kifejezés nem kerül kiértékelésre és a program hibával tér vissza. Például a következő kifejezésben kombinátor típusa nem lett explicit megadva és a kontextus alapján sem lehet a típusát teljesen kikövetkeztetni:

definíciója: , tehát a típusa: . az első argumentumának és visszatérési értékének a típusa, a példában . pedig a második argumentum típusa, ami a kifejezésből nem derül ki, ezért Unknown lesz. Gondolhatjuk, hogy típus kiderítése nem is szükséges, mert kombinátor a második paraméterét ignorálja, de a típusrendszer megköveteli, hogy minden típus egyértelműen meg legyen határozva. A fenti példa jól típusozott lesz, ha kap egy második paramétert is:

Vagy ha explicit határozzuk meg típusát:

Most már tudjuk, hogy második paraméterként egy Bool típusú kifejezést vár és -n az első paramétert alkalmazva egy típusú kifejezést kapunk.

### 3. 2. 2. Termek

### 3. 2. 3. Parser (Lexikális elemző)

Szintax, Típusok megadásának módjai, Definíciók használata, milyen volt még a PreType-ok előtt

### 3. 2. 4. Típusellenőrző

típus következtetés, infer-check, unification, típusparaméterek használata

### 3. 2. 5. Kiértékelő

### 3. 2. 6. Sekély implementáció

Hogyan működik, összhasonlítani a deeppel: implementáció, használata, előnyök, hátrányok

## 3. 3. Példák

InsertionSort

Faktoriális számítás

# 4. Összegzés/Konklúzió

# 5. Irodalomjegyzék