Szakdolgozat

Sarkadi Csaba

Debrecen
2007

Debreceni Egyetem
Informatikai Kar

Egy programozási nyelv interpreterének készítése Java programozási nyelven

Témavezető:
Simon Gyula
Egyetemi oktató

Készítette:
Sarkadi Csaba
Programtervező Informatikus szak

Debrecen
2007

Tartalomjegyzék

Bevezetés

Napjainkban sok programozási nyelv található, és napról napra egyre több jelenik meg, főleg a script nyelvek csoportjában.

A legelterjedtebb nyelvek mind rendelkeznek interpreterrel, a script nyelvek, pedig nagy általánosságban csak azzal.

Az interpreterek képezik a legtöbb fordítóprogram alapját, a programkódot soronként értelmezik és azt a megadott sorrendben végrehajtják.

A funkcionális és logikai nyelvek is szinte csak interpreterrel rendelkeznek, fordítóprogrammal nagyon ritkán.

Egy programozási nyelv konstruálásának sok lépése, és része van, ezek közül több lépéssel is fogok foglalkozni dolgozatom során, ilyen lépések, pl. a lexikális, a szintaktikai és szemantikai elemzők elkészítése, illetve ezek alapján az interpreter és a fordítóprogram konstruálása.

Ezen interpreterek és fordítóprogramok architektúrája még mindig a korábbi, évekkel ezelőtti elgondolás szerint készülnek, készültek. És ebből származik a legnagyobb probléma is ezen programokkal, hogy a futtatható kód elkészítése, vagy annak végrehajtása még mindig a korábbi hardware architektúrákra vannak tervezve, így nem képesek kihasználni a legújabb fejlesztések, számítási forrásokat.

Egyik legnagyobb, most már mindenki által elérhető fejlesztés a kettő és többmagos processzorok. Mint korábban említettem, az interpreterek és fordítóprogramok többsége még a régi, egy processzormagos hardware architektúrákra vannak tervezve, nem tudják kihasználni a többlet teljesítményt, nem képesek párhuzamosan gondolkodni, és végrehajtani.

Itt jegyezném meg a software gyártók egyik legnagyobb problémáját. A legtöbb nagy számításigényű software architektúrájából, tervezésükből és régebbi elkészülésükből adódóan nagyon nehéz lenne átírni csak úgy olyan kódra, mely képes lenne kihasználni a számítási többletkapacitást és a párhuzamosítás lehetőségét. Ezen programok esetében az egész architektúrát újra kell tervezni és az alapján újra megírni a softwaret.

Ez fejlesztési költségben sokszor majdnem ugyanakkora pénzösszeget jelent a nagyobb cégeknek, mint az eredeti software kifejlesztése jelentett.

Az interpreteres nyelvek közül a funkcionális és a logikai nyelvek sokszor igényelnek hatalmas számítási kapacitást. Ilyen nyelvek pl. a prolog, a lisp, haskel és társaik. Ezen nyelvek nagyon jól párhuzamosíthatóak, sőt a legelterjedtebb nyelvek rendelkeznek párhuzamos változattal is.

Itt szeretnék egy picit kitérni a funkcionális nyelvekre, melyeknél egy program típus-, osztály- és függvénydeklarációk, illetve függvénydefiníciók sorozatából, valamint egy kezdeti kifejezésből áll. A kezdeti kifejezésben tetszőleges hosszúságú (esetleg egymásba ágyazott) függvényhívás-sorozat jelenhet meg. A program végrehajtását a kezdeti kifejezés kiértékelése jelenti.

A következő részletet Juhász István: Programozás 2 jegyzetből vettem.

„A kezdeti kifejezés redukálása (a nyelv által megvalósított *kiértékelési stratégia* alapján) mindig egy redukálható részkifejezés (egy *redex*) átírásával kezdődik. Ha a kifejezés már nem redukálható tovább, akkor *normál formájú* kifejezésről beszélünk.

A kiértékelés lehet *lusta kiértékelés*, ekkor a kifejezésben a legbaloldalibb legkülső redex kerül átírásra. Ez azt jelenti, hogy ha a kifejezés egy függvényhívás, akkor az aktuális paraméterek kiértékelését csak akkor végzi el a rendszer, ha szükség van rájuk. A lusta kiértékelés mindig eljut a normál formáig, ha az létezik.

A *mohó kiértékelés* a legbaloldalibb, legbelső redexet írja át először. Ekkor tehát az aktuális paraméterek kiértékelése történik meg először.

A mohó kiértékelés gyakran hatékonyabb, de nem biztos, hogy véget ér, még akkor sem, ha létezik a normál forma.”

Vagyis, egy lusta kiértékelést valló funkcionális nyelven írt komolyabb programok esetén hosszú órákon keresztül is eltarthat egy-egy kifejezés kiértékelése.

A magam szemszögéből igazi kihívásnak tartom egy számításorientált, funkcionális elemekkel rendelkező programozási nyelv és a hozzá tartozó interpreter megtervezését és elkészítését.

Az architektúráját ezen programnak úgy szeretném megtervezni, hogy később is felhasználható, illetve újrahasznosítható legyen.

Dolgozatom elkészítésében nagy segítségemre lesz a korábbi, Programozás 2 tárgyból elkészítette beadandó feladat, mely egy kisebb tudású interpreter készítéséről szólt, a Kalandozó magyarok nevezetű feladat, melyet a függelékben leírtam.

A rendszer kódolására a Java programozási nyelvet fogom használni, mely napjaink egyik legelterjedtebb objektum orientált nyelve, és rengeteg előnye is van.

Az elmúlt években a Java nyelv hatalmas változásokon ment át. Az 1.5-ös változatban megjelent a változó paraméterszámú függvény, a generikus programozás és az autoboxing. Mindemellett az újabb és újabb változatok megjelenésével a Java programokat futtató JVM is hatalmas változásokon ment át, sokszorosan újra lett írva, és verzióról verzióra egyre inkább nőtt a futtatás sebessége. Az 1.6-os JVM minimum 30%-kal gyorsabb alkalmazások futtatásában, mint az 1.5-ös, köszönhetően a jobb memóriakezelésnek. Összességében elmondható, hogy sebességben a Java nyelv már egy szinten van elődjével, a C++-al.

Mindemellett a Java nyelv egyre terebélyesebbre nő. Ez sokkal könnyebbé teszi a programozók dolgát. Ez a dolgozatom szempontjából a szál és osztott memóriakezelést segíti. Mindemellett a Class és Object osztályok bizonyos módszerei hatalmas segítségemre lesznek. Gondolok itt arra, hogy pl a program kódjában az van, hogy fakt(3), akkor ha van egy fakt nevű függvényem, azt közvetlenül megtudom hívni a Class osztály segítségével, nincs hozzá

szükségem hosszú és méretes switch-case elágazásokra, amiket egyszerűen pár sor kóddal lehet helyettesíteni, köszönhetően a Java nyelv lehetőségeinek.

Az objektum orientált nézetnek köszönhetően a tervezés is egyszerűsödik, osztályokban kell gondolkodni és öröklődésekben, melyek az utóbbi évek egyik leginkább használt programozói irányvonalát a legjobban jelképezik.

1. Az interpreter tervezése

1. 1 – Általánosságban a rendszerről és a fejlesztés menetéről

A célom egy olyan software megalkotása, mely képes egy általam kreált programozási nyelvet interpretálni, és az adott nyelvben írt programok kódját futtatni.

Az interpreter kifejlesztése során az inkrementális fejlesztési modellt fogom alkalmazni. Ezen rendszerfejlesztési modell lényege, hogy az implementálás része kis méretű egységekben, inkremensekben történik, a mindenkori rendszerbe inkremenseket építünk, inkremensekkel bővítjük, és ha szükséges, nyilván megismételjük az inkremenshez akár a specifikációs tervezés lépését is.

Első lépésben meg kell határozni a program követelményeit, ezután a rendszerhez szükséges inkremenseket kell körülhatárolni.

Jelen esetben az inkremensek az alábbi működési egységeket jelentik:
Controller – a rendszer központja, ez irányítja a többi egység működését.
GUI – a felhasználói felület.
ThreadManager – a párhuzamos végrehajtáshoz szükséges szálakat vezérli.
SharedMemory – osztott memória, a konkurenciavezérléshez szükséges.
Functions – egy függvénykönyvtár, itt találhatóak a megalkotott programozási nyelv kódjában felhasználható függvények.

Az inkremensek közül az első 4 valójában osztályokat reprezentál. A Functions inkremens pedig a java.lang.Math osztály beépített függvényeit képes felhasználni.

A függelékben található 1. ábra megfelelően szemlélteti az általam elgondolt rendszer működését, illetve a tervezett osztályok egymás közötti kommunikációját. A nyilak jelentik az információ-áramlás irányát.

Az MVC design pattern alapján tervezett rendszerben a központi szerep a Controller osztályé. Elsődleges feladata a GUI-tól érkező utasítások értelmezése, feldolgozása, a többi osztály működésének vezérlése.

A felhasználó a GUI szövegszerkesztőjében megírja a programkódot, az OK gombra kattint, és a kód továbbítódik a Controllernek.

A kódot a vezérlő osztály soronként interpretálja, és továbbítja feldolgozott formában a Threadmanagernek, ami pedig arról gondoskodik, hogy a megfelelő szálak létrejöjjenek, fussanak, és sikeres futás esetén értesíti a vezérlőt a részeredményről, ami a felhasználói felületen megjeleníti azt.

A szálkezelő létrehozza a szálakat, amelyek már a megfelelően formázott, kiszámítandó kifejezést tartalmazzák, és a szálak a függvények közül a megfelelőeket alkalmazzák.

Azok a szálak, melyek befejezték működésüket, az osztott memóriában tárolják a változók értékét, és felszabadítják a változókról a zárakat.

Itt említeném meg a zárak jelentését a programomban. Az osztott memóriában tárolom a változók nevét, típusát, értékét és a hozzájuk tartozó zárakat.
Ha egy végrehajtási szál egy változó értékéhez szeretne hozzáférni, akkor 3 lehetőség van:

1. Nincs zár az adott változón. Ez azt jelenti, hogy nincs másik szál, amelyik ezen változó értékét használná. Ebben az esetben a szál szabadon hozzáférhet a változóhoz. Ha az értékét módosítani akarja, akkor olvasási zárat rak rá, ha csak olvasni akarja az értékét, akkor írási zárat.
2. Írási zár van az adott változón. Ekkor az értéke szabadon olvasható, de semmi mást nem lehet vele csinálni.
3. Olvasási zár van az adott változón. Ekkor az értéke nem írható és nem olvasható, mivel egy másik szál éppen felhasználja és módosítja a változó értékét.

1. 2 Az inkremensek fejlesztése

A megadott inkremensek közül a Functions rész gyakorlatilag kész van, azt már csak a kész rendszerbe kell majd végül integrálni.

A GUIt a legegyszerűbb elkészíteni. Szükség van egy szövegszerkesztő felületre, amiben tudok menteni, betölteni a szerkesztésen felül, továbbá szükséges egy programkód futtatási opció és végül egy olyan rész, ahol a végrehajtott program kód kimenete található.

A SharedMemory feladata egyszerű, jól körül határolt, ezért könnyen implementálható. Ez lesz a következő lépcső.

A SharedMemory osztályhoz felállított követelmények egyszerűek:
1. Tudni kell változók értékét, típusát, zárait beállítani/lekérdezni.
2. Szükséges egy metódus, mellyel lekérdezhető, hogy egy adott változónév megtalálható-e már az osztott memóriában.
3. Tudni kell lekérdezni az összes változó listáját.
4. Szükség van egy deklarációs metódusra (implicit változó deklarációhoz).

Ezen követelmények alapján a SharedMemory osztályt egyszerűen és gyorsan lehet implementálni.

Negyedik lépésben a vezérlőt készítettem el. A vezérlő feladata komplex, egyrészt fogadnia és kezelnie kell a felhasználói felületről érkező interakciókat, tudnia kell feldolgozni a programkód szövegét, és végül azt soronként elemezni és végrehajtatni.

Az utolsó lépésben a szálkezelőt implementáltam. A szálkezelő feladata a vezérlőtől érkező szálindítási utasítások fogadása, végrehajtása, a szálak memóriakezelésének koordinálása, és végül a vezérlő értesítése a végrehajtott feladatokról.

A továbbiakban a programom legtöbb figyelmet igénylő részeit fogom elemezni és részletezni, főleg a szintaktikát, és az elemzést. Végül a szálkezelésről írnék pár sort és az 5. fejezetben néhány példa kódot ismertetnék.

2. A rendszer áttekintése

2. 1 A változók tárolása, és azok deklarációja

Az általam kitalált programozási nyelv, mivel matematikai számításokra tervezem, csak numerikus változókat lehet használni, tehát elég, ha csak egész és valós változókat lehet deklarálni.

A változókat egy Hashtable-ben fogom tárolni. Ez a Hashtable a Java programozási nyelvben a hash tábla, mint absztrakt adatszerkezet konkrét megfelelője.

A kulcs az maga, a feldolgozandó programkód szövegében szereplő változó neve lesz, a hozzá tartozó érték pedig kétféle. Az egyik érték a null – ezt az értéket fogom használni a nem meghatározott értékű változók tárolásánál. A másik érték-lehetőség pedig a változó konkrét, már kiszámolt értéke.

Ez a megoldás arra lesz igazán jól használható, ha például olyan kifejezéssel találkozunk, hogy x = (3y + 9z -5)^2

A numerikus típusok mellett a tömb típus lesz még használható, abból is csak az 1 dimenziós.
Deklarációja:
tomb [méret] tipus változónév;

A tömb eleminek a típusa float, double vagy integer lehet.

A méret határozza meg a tömb elemeinek a számát.

A változóknál nem csak explicit, hanem implicit deklaráció is lehetséges, ez viszont csak akkor, ha értékadó utasítás van, ebben az esetben a változó típusa double lesz.

2. 2 Vezérlési szerkezetek

Az általam elkészített programozási nyelvben megtalálható egy for ciklus, mely egy adott változó segítségével megadott lépéseket, a for ciklusban deklarált számban.

Meglátásom szerint egy minimális utasításokkal, képességgel ellátott nyelvben, mint amivel én is foglalkozom, bőven elég lesz a for ciklus is, nincs szükség több féle ciklusra.

A for ciklus szintaktikája:

For változó = kezdőérték to végérték do [by lépésköz]
Begin
Utasítások…;
End;

Amennyiben az elemző egy for ciklust talál, úgy az elemzés során felismeri a kezdőértéket, a végértéket, és a lépésközt, majd a ciklusmagját képező utasításokat ennek alapján végrehajtja.

Annak érdekében, hogy teljes legyen a vezérlés a ciklusok során, egy brek; utasítást is hozzáadtam az interpreter utasításaihoz, lényege, hogy a for ciklus magjában kiadva a ciklus végrehajtását befejezi.

2. 3 Feltételes kifejezések

A feltételes kifejezések során csak az if … then … else szerkezeteket fogom megvalósítani.

Amennyiben a vezérlő feltételes szerkezetet tartalmazó sorral találkozik, úgy a program kód elkövetkező sorait veszi folyamatosan.
Egy változó segítségével számolja a program szövegének megfelelő részében található begin-end utasítás párokat. Ha begint talál, akkor a változó értékét eggyel növeli, ha endet, akkor eggyel csökkenti, ha a változó értéke 0 akkor megtaláltuk a feltételes szerkezethez tartozó programblokk végét.

Az így felismert kód részletet továbbítjuk a ThreadManagernek.

A feltételes utasítás szintaktikája:
If (feltétel) then
Begin
Utasítások…;
End;
Else
Begin
Utasítások…;
End;

A feltétel egyszerű kifejezés lehet, melynek alakja:
X > Y, X < Y, X >= Y, X <= Y, X = Y

Úgy gondolom, hogy ennél több feltétel vizsgálati lehetőségre nincs szükség, ezek segítségével bármit ki lehet fejezni.

2. 4 Szintaktika és egyéb utasítások

Egy program szintaktikája az általam kigondolt programozási nyelvben a következő:

Az első és legfontosabb dolog, hogy egy sorban egy utasítás állhat. Ez a megszorítás a program kódját átláthatóvá és könnyen értelmezhetővé teszi.

Üres utasítás: null;

Megjegyzések: { és } között helyezhetőek el.
Blokk kezdete és vége jelzése: begin … end;

Feltételes vezérlés:
if (feltétel) then begin utasítások; end;
else begin utasítások; end;

A feltételek egyszerű hasonlítások lehetnek, tehát a következő vizsgálatok egyike:
X > Y, X < Y, X >= Y, X <= Y, X = Y.

Meglátásom szerint ezeknél több hasonlítás vizsgálatára nincs szükség, illetve más nem merülhet fel.

For ciklus:
1. Variáció:
for változó = kezdőérték to végérték do
begin utasítások; end;

Ebben az esetben a lépésköz 1, és csak pozitív irányban haladhatunk, különben hiba.

2. Variáció:
for változó = kezdőérték to végérték by lépéskőz do
begin utasítások; end;

Ekkor a lépésközt mi adjuk meg, tehát nem csak egyesével haladunk a változó kezdőértékétől a végértékéig, hanem az általunk megadott lépésközökkel.

For ciklus magjában adhatjuk ki a break; parancsot, ennek hatására a for ciklusból kilép a végrehajtás.

Példák:

```
For i = 1 to 5 do
Begin
X = x + i ;
End;
```

```
For i = 1 to 10 by 3 do
Begin
```

X = x + i;
End;

Feltételezve, hogy x = 0, mindkettő ciklus kezdete előtt, az első esetben x = 15 lesz a ciklus lefutása után, a második esetben x = 22 lesz.

A változók deklarációja a programkód kezdetét jelző begin és variable kulcsszavak megadása után történhet a következő féle képpen:
Variable
Float x,y;
Integer z;

A típus megadása után szóköznek kell állnia, a változók neveinek felsorolása között pedig csak vesszők állhatnak.

A változók deklarálása után a Begin kulcsszót követően állhat a program kódja. A változók lehetséges típusai: float, double, integer, tomb. A konverziók miatt ajánlott a double típus használata.

Egyéb utasítások:
Kiir (változó);

Ezen utasítás hatására a végrehajtás befejezettségétől függetlenül az adott változó, ha értéke meghatározott, kiíródik a képernyőre, egyébként, pedig nem meghatározott üzenetet kapunk.

Ezen felül még a java.lang.Math osztály függvényei lesznek használhatóak, azok viszont a konkrét, Java nyelvbeli szintaktikájuk alapján.
Ezen függvények:

`static double` **`abs`**`(double a)`
Returns the absolute value of a `double` value.
`static float` **`abs`**`(float a)`
Returns the absolute value of a `float` value.
`static int` **`abs`**`(int a)`
Returns the absolute value of an `int` value.
`static long` **`abs`**`(long a)`
Returns the absolute value of a `long` value.
`static double` **`acos`**`(double a)`
Returns the arc cosine of an angle, in the range of 0.0 through *pi*.
`static double` **`asin`**`(double a)`
Returns the arc sine of an angle, in the range of -*pi*/2 through *pi*/2.
`static double` **`atan`**`(double a)`
Returns the arc tangent of an angle, in the range of -*pi*/2 through *pi*/2.
`static double` **`atan2`**`(double y, double x)`
Converts rectangular coordinates (`x`, `y`) to polar (r, *theta*).
`static double` **`cbrt`**`(double a)`
Returns the cube root of a `double` value. `S`

`tatic double` **`ceil`**`(double a)`

Returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the argument and is equal to a mathematical integer.

`static double` **`cos`**`(double a)`

Returns the trigonometric cosine of an angle.

`static double` **`cosh`**`(double x)`

Returns the hyperbolic cosine of a `double` value.

`static double` **`exp`**`(double a)`

Returns Euler's number *e* raised to the power of a `double` value.

`static double` **`expm1`**`(double x)`

Returns e^x -1.

`static double` **`floor`**`(double a)`

Returns the largest (closest to positive infinity) `double` value that is less than or equal to the argument and is equal to a mathematical integer.

`static double` **`hypot`**`(double x, double y)`

Returns sqrt(x^2 +y^2) without intermediate overflow or underflow.

`static double` **`IEEEremainder`**`(double f1, double f2)`

Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.

`static double` **`log`**`(double a)`

Returns the natural logarithm (base *e*) of a `double` value.

`static double` **`log10`**`(double a)`

Returns the base 10 logarithm of a `double` value.

`static double` **`log1p`**`(double x)`

Returns the natural logarithm of the sum of the argument and 1.

`static double` **`max`**`(double a, double b)`

Returns the greater of two `double` values.

`static float` **`max`**`(float a, float b)`

Returns the greater of two `float` values.

`static int` **`max`**`(int a, int b)`

Returns the greater of two `int` values.

`static long` **`max`**`(long a, long b)`

Returns the greater of two `long` values.

`static double` **`min`**`(double a, double b)`

Returns the smaller of two `double` values.

`static float` **`min`**`(float a, float b)`

Returns the smaller of two `float` values.

`static int` **`min`**`(int a, int b)`

Returns the smaller of two `int` values.

`static long` **`min`**`(long a, long b)`

Returns the smaller of two `long` values.

`static double` **`pow`**`(double a, double b)`

Returns the value of the first argument raised to the power of the second argument.

`static double` **`random`**`()`

Returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`.

`static double` **`rint`**`(double a)`

Returns the `double` value that is closest in value to the argument and is equal to a mathematical integer.

`static long` **`round`**`(double a)`

Returns the closest `long` to the argument.

`static int` **`round`**`(float a)`

Returns the closest `int` to the argument.

`static double` **`signum`**`(double d)`

Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.

`static float` **`signum`**`(float f)`

Returns the signum function of the argument; zero if the argument is zero, 1.0f if the argument is greater than zero, -1.0f if the argument is less than zero.

`static double` **`sin`**`(double a)`

Returns the trigonometric sine of an angle.

`static double` **`sinh`**`(double x)`

Returns the hyperbolic sine of a `double` value.

`static double` **`sqrt`**`(double a)`

Returns the correctly rounded positive square root of a `double` value.

`static double` **`tan`**`(double a)`

Returns the trigonometric tangent of an angle.

`static double` **`tanh`**`(double x)`

Returns the hyperbolic tangent of a `double` value.

`static double` **`toDegrees`**`(double angrad)`

Converts an angle measured in radians to an approximately equivalent angle measured in degrees.

`static double` **`toRadians`**`(double angdeg)`

Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

`static double` **`ulp`**`(double d)`

Returns the size of an ulp of the argument. `static float` **`ulp`**`(float f)`

Returns the size of an ulp of the argument.

3. Kifejezések, kiértékelésük, és kezelésük

3. 0 A kifejezések

A kifejezések képezik az alapját az egész programnak. Ezeket tudnia kell feldolgozni és kiértékelni az elemzőnek. Ez az egész rendszer legnagyobb méretű alrendszere is, hiszen tudni kell feldolgozni bármilyen bonyolult kifejezést, függvényhívást.

A kiértékelés során rengeteg elemzésre és sztring-manipulációra lesz szükség. Elgondolásom szerint első lépésben feldarabolom tokenekre az aktuális kifejezést, majd a Rutishauser módszer segítségével megállapítom a kifejezésben elhelyezkedő zárójelek helyét, végül a lengyel forma segítségével megállapítom a kifejezés értékét.

Mindig a legbaloldalibb, legbelső kifejezés részletet fogom kiértékelni és átírni, ezt a funkcionális programozási nyelvekben lusta kiértékelésnek hívják.

A zárójelek helyének meghatározásával megállapíthatóak a szintmélységek, azaz a kifejezésben mely részletek találhatóak zárójelek között, és pontosan hány zárójelen belül található az adott részlet.

Az éppen aktuális legbelső zárójelpár közti alkifejezés kiértékelésekor már rögtön figyelembe veszem, hogy vajon az a zárójelpár egy függvény része-e, amennyiben igen, az azonnal kiszámításra is kerül.

3. 1 A Rutishauser módszer

Ezt a módszert több okból is szeretném ismertetni. Egyrészt a legelső Fortran fordító ezt használta, másrészt ennek, és a lengyelformára hozás módszerének egy keverékét fogom használni a kifejezések kiértékelésére.

A Rutishauser módszer a teljesen zárójelezett kifejezések kiértékelésére szolgál.
Az algoritmus a kifejezés minden egyes szimbólumához egy egész számot rendel a következő módon:
Sorban megvizsgálja a kifejezés szimbólumait balról jobbra haladva. A kifejezés elejét és végét külön szimbólumok jelzik: ⊥. Például:
⊥ a + (b * c) ⊥
Jelölje n(i) az i-edik szimbólumhoz rendelt egész számot. A számozást 0-val kezdjük. A hozzárendelés algoritmusa:

1. i = 0; n(0) = 0;
2. i++;
3. ha si = ⊥, akkor 6. pont
4. ha si = (, vagy si egy operandus, akkor n(i) = n(i-1)++;
5. n(i) = n(i-1)--;
6. n(j) = 0; vége.

Például

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
si:	⊥	(	(	a	+	b	)	*	c	)	/	(	e	-	f	)	⊥
n(i):	0	1	2	3	2	3	2	1	2	1	0	1	2	1	2	1	0

Az így kiegészült formulának az az előnye, hogy az n(i) értékek alapján egyértelműen meghatározható a műveletek sorrendje. Ez úgy történik, hogy balról jobbra megkeressük a legnagyobb n(i) értékét. Ha több ilyen is van, akkor balról a legelsőt. Ekkor n(j) = k, ezután a kifejezés aktuális részlete:

j-1	j	j+1	j+2	j+3
α	x	Ω	y	β
k-1	k	k-1	k	k-1

Ekkor x és y operandusok, Ω műveleti jel, α, β: (,) vagy ⊥,⊥
Hajtsuk végre x Ω y-t. Ha α, β pár megegyezik (,) párral, akkor α x Ω y β helyett vesszük a művelet eredményét, és ehhez k-1-et rendelünk.

⊥ (A * c) / (e – f) ⊥ A := a+b
0 1 2 1 2 1 0 1 2 1 2 1 0
α x Ω y β B := A*c

⊥ B / (e – f) ⊥
0 1 0 1 2 1 2 1 0
α x Ω y β C := e-f

⊥ B / C ⊥ D :=B/C
0 1 0 1 0

⊥ D ⊥
0 0 0

A program írása során egy más jellegű problémába futottam bele a kifejezések kiértékelése során: a java.lang.Math csomag függvényei között találhatóak 2 változós függvények is. Ezek általános alakja:
Függvény_név(paraméter1, paraméter2)

Vagyis a Rutishauser módszer működése során a ,-t is értékelnünk kell. Kérdés, hogy milyen értéket kapjon. Az eredeti algoritmus alapján az azt megelőző változóhoz tartozó értékhez képest eggyel kisebb számot kap értékül.
Ez így akár jó is lehetne, de hadd vázoljam itt annak azt az algoritmust, mely az aktuális kifejezés sztringből kiemeli az aktuálisan kiértékelendő részletet, azaz a legbelsőbb zárójelek mögötti alsztringet.

1. Legyen a legelső, legnagyobb eleme a Rutishauser módszer által előállított számtömbnek max, és az ehhez tartozó tömbindex pedig maxindex!
2. A sztringben a maxindex pozíciótól kezdjük el másolni a tokeneket egy másik sztringbe, egészen addig, amíg az adott tokenekhez tartozó számok a Rutishauser módszer által meghatározott tömbben max vagy max-1 értékűek!
3. Az így előállt sztringet értékeljük ki, az eredményt írjuk az eredeti sztringben a kifejezés helyére, a hozzá tartozó (és) jeleket töröljük.

Így már látható, hogy ebben az esetben a vessző értéke nem lesz jó az eredeti algoritmus szerint. Ha a vessző értéke eggyel nagyobb lenne, mint az előtte álló szám, akkor az az előbb ismertetett algortimus működését felborítja, mert nyilván akkor a vesszőt akarná kiértékelni, ez nem lesz jó.
Ha a vesszőhöz tartozó érték eggyel kisebb, mint az azt megelőző szám értéke, az pedig az egész Rutishauser algoritmust bonyolítja túl. Hosszas gondolkodás után végül úgy gondolom, hogy a kettővel kisebb szám lesz a megfelelő érték.

Az így átalakított Rutishauser módszer a következő algoritmust adja:

1. i = 0; n(0) = 0;
2. i++;
3. ha si = ⊥, akkor 6. pont
4. ha si = , , akkor n(i) =n (i-1)-2 és 2. pont
5. ha si = (, vagy si egy operandus és s(i-1)=, , akkor n(i) = n(i-1)+2, egyébként n(i)=n(i-1)+1 és 2. pont
6. n(i) = n(i-1)-1;
7. n(j) = 0; vége.

3. 2 A lengyel forma

A lengyel forma a kifejezések felírásának egy olyan módja, melyben a műveletek végrehajtása egyértelmű, nincs szükség az operátor precedencia figyelembe vételére és zárójelmentes.

J. Lukasiewicz lengyel matematikus használta először a kifejezések felírásának prefix illetve postfix alakját.

Vegyünk egy összetettebb, zárójelezett kifejezést:
(a + b) * (c - d)
Ennek a postfix formája:
a b + c d - *
Prefix formája
* + a b – c d

Ezután a postfix alakkal fogok dolgozni. A postfix alak jellemzői:

1. A műveleti jelek olyan sorrendben követik egymást, amilyen sorrendben végre kell azokat hajtani.
2. A műveleti jel közvetlenül az ő operandusai után áll.
3. Teljesen zárójelmentes forma.

A postfix kifejezések kiértékelését a [2] a következő módon szemlélteti, egy veremautomatával:
1. Az adott szövegben balról jobbra haladunk.
2. Ha a soron következő szimbólum operandus, akkor helyezzük el azt a verembe.
3. Ha a soron következő szimbólum műveleti jel, amit jelöljünk ¤ -vel, akkor vegyük ki a veremből a két legfelső operandust, x-et és y-t. Ezután számítsuk ki x ¤ y értékét, és a végeredményt helyezzük el a verem tetején.

Ez eddig szép és jó, most már csak megfelelő alakra kell hozni az eredeti kifejezést.
Az alábbi leírást a [2] könyvben találtam:

„Tekintsük az a + b kifejezést!
Válasszuk el egymástól vesszővel az egyes szimbólumokat! Nevezzük a vesszővel elválasztott stringeket elemeknek:
a , + , b
Ez a kifejezés két nagyon egyszerű művelettel lengyel formára hozható:
a.) Cseréljük meg a két utolsó elemet:
a , b , +
b.) Konkatenáljuk a két utolsó elemet:
a, b +
c.) Konkatenáljuk a két utolsó elemet:
a b +”

Ez a módszer az egyszerű kifejezéseket nagyon könnyen, és gyorsan átalakítja lengyel formára. A probléma akkor jön, amikor már nem egy, hanem több műveleti jel is van a kifejezés-

ben. Ezen probléma megoldására először a [2] könyvben található leírást próbáltam alkalmazni:

„1. Másoljuk a következő szimbólumot a verembe.
2. Cseréljük meg a verem tetején lévő két legfelső elemet.
3. Konkatenáljuk a verem tetején lévő két elemet eggyé, és helyezzük vissza a verembe.

Ezen módszer bemutatása egy példával:
a * b - c
a.) Átmásoljuk „a” -t a verembe.
b.) Átmásoljuk „*” -t a verembe.
c.) Átmásoljuk „b”-t a verembe.
d.) Megcseréljük a verem tetején lévő két legfelső elemet:
verem: a , b , *
e.) Konkatenáljuk a verem tetején lévő két elemet.
verem: a , b *
f.) Konkatenáljuk a verem tetején lévő két elemet.
verem: a b *
g.) Másoljuk „-” -t a verembe.
h.) Másoljuk „c” -t a verembe.
i.) Cserélünk.
verem: a b *, c , -
j.) Konkatenálunk.
verem: a b *, c –
k.) Konkatenálunk.
verem: a b * c –„

Végül rá kellett jönnöm, hogy ez a leírás így nem megfelelő. Egy egyszerű példán keresztül szemléltetném a hibáját:
4 – 5 * 3 = 12
A megadott algoritmus szerint ennek a kifejezésnek az átalakított alakja a következő:
4 5 – 3 *
Amit ha postfix módon kiértékelünk, akkor 4 5 – 3 * = -3
A helyes alak: 4 5 3 * -

A megoldást a [7]-ben találtam. Eszerint a műveleti jeleket prioritással kell kezelnünk. Mivel az eredeti cikk a zárójelekkel is foglalkozik, és azok kezelését én máshogy oldom meg, így az [7]-ben leírt megoldást a következőre dolgoztam át:
Prioritási szintek:

0. +, -
1. *, /
2. ^ (hatványozás)

Így a műveleti jelek sorrendje már megfelelően meghatározható és egy adott kifejezés lengyel formára hozható, azon egyszerű tényeket figyelembe véve, hogy az operandusok sorrendje nem változhat, és az operátorok a postfix formában olyan sorrendben követik egymást, amilyen sorrendben végre kell hajtani őket.
Az általam kigondolt algoritmus egy vektorral dolgozik. Az elemei operátorok vagy operandusok lehetnek kezdetben (az eredeti kifejezés tokenekre darabolva).

1. Haladjunk balról jobbra a vektorban, és ha találunk ^ jelet, akkor:
Az előtte lévő tömbelemet jelöljük x-szel, az utána következőt x-nal.
Vagyis a vektorban a következő 3, egymás utáni elemmel dolgozunk: x, ^, y
Ezt a 3 elemet töröljük a vektorból, és írjunk a helyükre egyetlen új elemet, x y ^ -t.
2. Ismételjük az 1-es lépést addig, amíg a vektorban találunk ^ elemet.
3. Az 1. lépést hajtsuk végre ismételten, de most már a * és / jeleket eltüntetve a vektorból, addig, amíg találunk még ilyen műveleti jeleket.
4. Az 1. lépést most a + és – jelekre alkalmazzuk, amíg találunk ilyen elemeket a vektorban;

Az algoritmus végül a számunkra szükséges postfix kifejezést állítja elő.
Egy példa:
6 * 3 ^ 3 / 9
1. lépés:
6, *, 3 3 ^, /9
2. lépés:
6 3 3 ^ * 9 /

Kiértékelve: 6 3 3 ^ * 9 / = 18

A zárójeles kifejezések kiértékelésére egy saját módszert fogok alkalmazni.

3. 3 A saját megoldásom az előbbi problémákra

A kifejezések kiértékelésére egy általam kigondolt módszert fogok használni, mely egy fajta keveréke a Rutishauser módszernek és a lengyel formának.
A módszer során egy függvény rekurzív hívásaival fogom kiszámolni az adott kifejezés értékét.

0. Lépés
   A kiértékelendő teljes kifejezés vizsgálata.
   Megszámoljuk, hogy ugyanannyi (jel van-e, mint) jel.
   Ha nem, akkor hibás a kifejezés, kivételkezelés.
   Egyébként megnézzük, hogy a kifejezés zárójelek között van-e, ha nem, akkor az elejére és a végére elhelyezünk egy nyitó és egy záró zárójelet.
   Az így vizsgált stringre meghívjuk a kiértékel (string) függvényt.
1. Lépés
   A kiértékel(string) függvény kezdetén vagyunk.
   Megnézzük első lépésben, hogy a kifejezés tartalmaz-e egyenlőség jelet. Ha igen, akkor a kiértékelendő stringet felbontjuk két alstringre, melyek közül az első az egyenlőség jel baloldala és egy záró zárójel, a második, pedig egy nyitó zárójel és az egyenlőség jel jobboldala.
   Ezután értékeljük ki a kifejezés mindkét oldalát rekurzívan, utána, pedig az egyenlőségjelnek megfelelően állítsuk be a változók értékét.

2. Lépés

   A Rutishauser módszernek megfelelően lássuk el számokkal a kapott kifejezést.

3. Lépés
   A legnagyobb számmal ellátott zárójelek közötti sztringeket alakítsuk át lengyel formára. Ezek a kifejezések már nem tartalmaznak zárójelet, így átalakításuk egyszerű. Az átalakítás után értékeljük ki, és legyen a kiértékel függvény visszatérési értéke az adott string által reprezentált kifejezés értéke. És ugrás az 1. pontra.

A 3 lépés megfelelő számú ismétlése során a kifejezés értéke előáll.
Amennyiben az eredeti kifejezés nem tartalmazott egyenlőségjelet, és nem tartalmazott ismeretlen értékű változókat, úgy írjuk ki a végeredményt, egyébként pedig jelezzünk hibát.

4. Szálkezelés, párhuzamosítás

4. 1 Az elgondolás

A célkitűzésem során említettem, hogy szeretném minél magasabb szinten párhuzamossá tenni a programok végrehajtását. Ezen célból a programkód végrehajtását nem soronként fogom elvégeztetni az interpreterrel, hanem párhuzamosan, vagyis minden egyes sor végrehajtására egy külön szálat fogok indítani.

Amennyiben a program egy sorának értelmezése során olyan értékre van szükség, melyet egy korábbi sor határoz meg, és a korábban indított szál még nem fejezte be a futását, úgy az érték miatt a szálat várakoztatni fogom.

Ez viszont felvet egy újabb problémát, méghozzá azt, hogy ha pl a 2. sorban meghatároztuk egy x változó értékét, ami a 3. sorban megváltozna, és a 4. sorban igényelnénk, úgy előfordulhat olyan, hogy a 3. sorhoz tartozó szál még nagyban dolgozik, mikor a korábban kiszámolt x értékkel a 4. sor már rég végzett.

Ezen probléma megoldására egy újabb Hashtable-t fogok alkalmazni, pontosabban egy zártáblázatot.
A kulcsok, mint korábban, most is a változók nevei lesznek, a hozzájuk tartozó értékek pedig kétféle zártípust fognak jelölni. Az egyik az olvasási zár. Ezt akkor fogom alkalmazni, amikor egy szál a végrehajtás során az adott változó értékét csak olvassa. Ez arra lesz jó, hogy ha egy másik szál is szeretné felhasználni a változó értékét, úgy olvashatja azt, amennyiben csak olvasási zár van rajta, de nem írhatja.
Ha írási zár van egy változón, akkor azt más szál se nem olvashatja, se nem írhatja.

Ha egy szál befejezte a működését, akkor az általa használt változókhoz tartozó zárakat kinullázza.

4. 2 Szálak és száltípusok

A programkód végrehajtása soronként történik, így a szálak szerepkörét is ehhez fogom igazítani. Az utasítás-típusok alapján a következő száltípusokat különböztetem meg:
1. Egyszerű végrehajtási szál
2. Feltételes végrehajtási szál
3. Ciklus végrehajtási szál

Az egyszerű szál felelős az olyan utasításokért, melyek egyszerű számolások (nem tartalmaznak értékadást), vagy értékadó utasítások.

A feltételes szál hajtja végre a feltételes utasítások magját.

A ciklus végrehajtási szál pedig a programban fellelhető for ciklusokat hajtja végre.

A vezérlő a programkód feldolgozása során a feldolgozott programrészekre létrehoz egy-egy szálat, a programkód feldolgozása után pedig elindítja azokat.

Az osztott memória segítségével a szálak futás közbeni konkurenciavezérlése megoldott.

5. Példakódok

5. 0

Ebben a fejezetben néhány alapvetőbb algebrai és informatikai algoritmus kódját közölném, az interpreter tudásának bemutatására.

5. 1 Legnagyobb közös osztó

```
variable
integer x,y,i,csere;
begin
	x=1024;
	y=768;
	for i = 0 to 100 do
	begin
		if(x<y) then
			begin
				csere=x;
				x=y;
				y=csere;
			end;
		x=x-y;
		if(x=1) then
			begin
			break;
			end;
		if(y=1) then
			begin
			break;
			end;
	end;
end;
```

5. 2 Faktoriális számítás

```
variable
double fakt;
integer x;
begin
fakt = 1;
for x = 1 to 10 do
        begin
                fakt = fakt*x;
        end;
end;
```

5. 3 Minimum-kiválasztásos rendezés

```
variable
integer min,minindex,x,y;
double csere;
tomb[10] double t;
begin
t[1]=5;
t[2]=4;
t[3]=3;
t[4]=2;
t[5]=10;
t[6]=1;
t[7]=8;
t[8]=7;
t[9]=6;
t[0]=0;
for i = 0 to 8 do
        begin
          min = i;
          for j = i+1 to 9 do
                    begin
                              if(t[j] <t[i]) then
                                        begin
                                                  min = j;
                                        end;
          end;
          csere = t[i];
          t[i]=t[min];
          t[min]=csere;
        end;
end;
```

5. 4 Buborék rendezés

```
variable
integer min,minindex,x,y;
double csere;
tomb[10] double t;
begin
t[1]=5;
t[2]=4;
t[3]=3;
t[4]=2;
t[5]=10;
t[6]=1;
t[7]=8;
t[8]=7;
t[9]=6;
t[0]=0;
for i = 9 to 0 do by -1
        begin
         for j = 1 to i do
                begin
                 if(t[j+1]<t[j]) then
                        begin
                         csere = t[j];
                         t[j]=t[j+1];
                         t[j+1]=csere;
                        end;
                end;
        end;
end;
```

Összegzés

Úgy érzem, a bevezetés részben vállaltakat sikerült maradéktalanul megvalósítanom. A program végső, működőképes állapota számomra kifejezetten tetszett, és büszke vagyok, hogy szinte a semmiből sikerült egy ilyen tudású interpretert készítenem.

Nagy örömömre szolgált, hogy a Fordítóprogramok és az Automaták és Formális nyelvek nevű tantárgyak során elsajátított tudást is konstruktívan fel tudtam használni.

A program implentálása során, úgy érzem, hogy elég sokat tanultam, mind a szövegfeldolgozás, a formális nyelvek, és a fordítóprogramok területéről, illetve még jobban elmélyíthettem a gyakorlati tudásomat a Java programozási nyelvben.

Az elkészített interpreter végállapotát tekintve úgy érzem, hogy ezen programot a jövőben is fejlesztgetni fogom. Elsődlegesen saját alprogram és függvény készítési lehetőséggel bővíteném, majd a későbbiekben egy még jobban áttekinthető program szintaktikát szeretnék létrehozni.

További elképzeléseim közt szerepel magának a programnak a későbbiekben olyan irányú bővítése, hogy akár oktatási segédeszközként is alkalmazható legyen, akár már az általános iskolák számára is.

Köszönetnyilvánítás

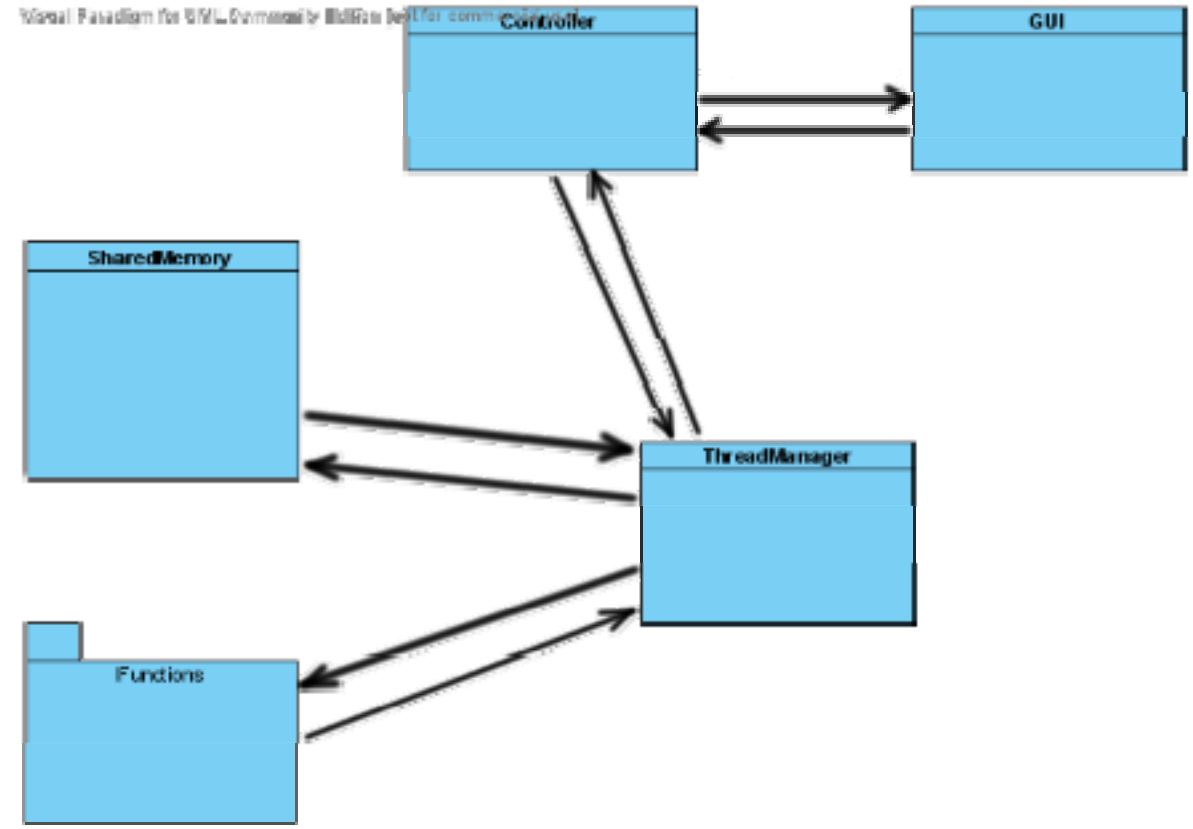
Elsődlegesen az Informatikai Kar vezetőségének szeretném megköszönni, hogy az általam választott témában írhattam, és ismereteimet mélyíthettem.

Továbbá szeretném megköszönni azt a tudást tanáraimnak, melyet az elmúlt években átadtak nekem és többi diáktársam számára, legfőképpen Dr Juhász Istvánnak, a Magasszintű programozási nyelvek, és Dr Dömösi Pálnak, a Fordítóprogramok területén megszerzett tudásért.

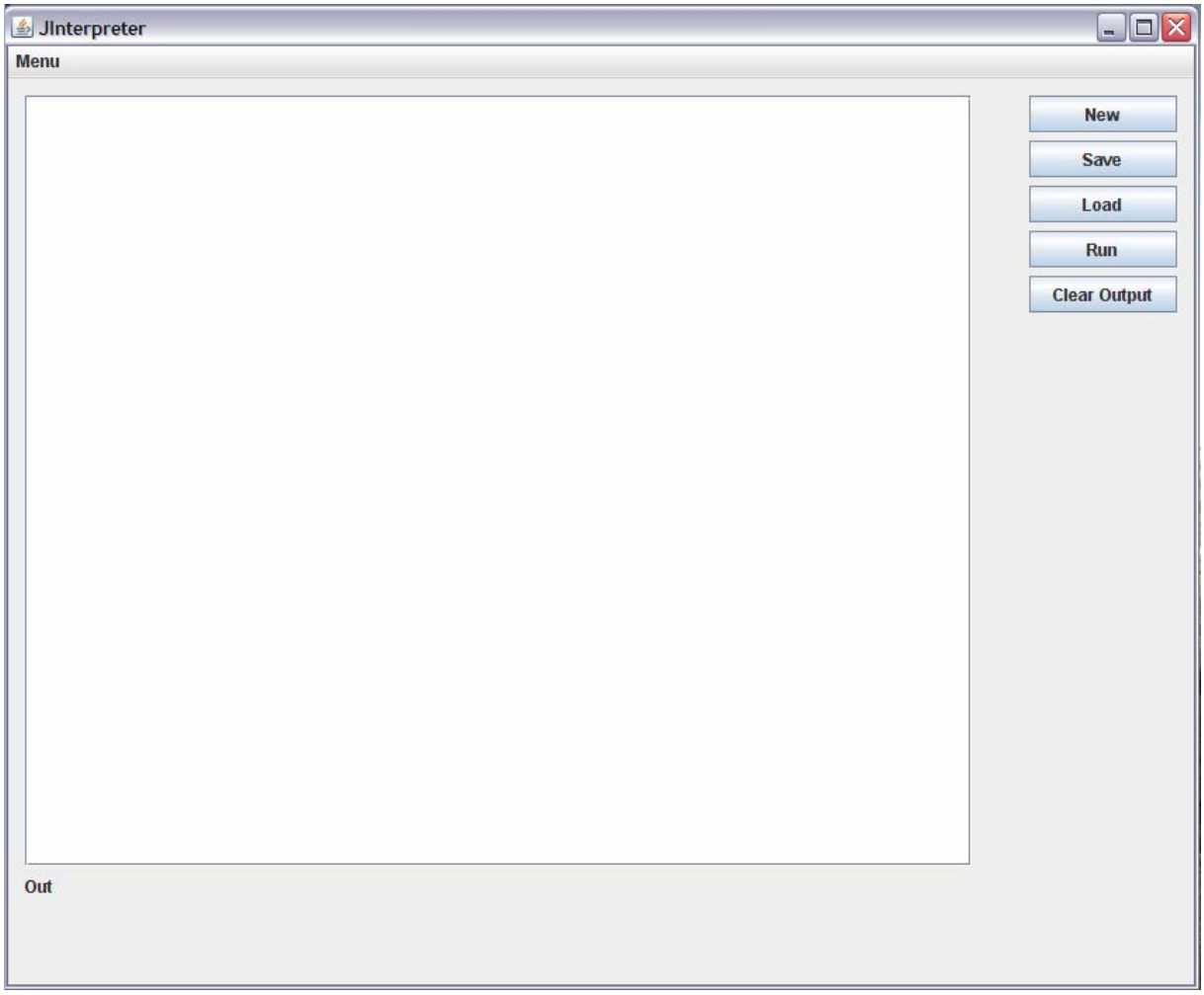
Úgy érzem, hogy nagy köszönettel tartozom még a segítségért, és az apró ötletekért a témavezetőmnek, Simon Gyulának, aki nélkül mindez a dolgozat, és a hozzá tartozó program nem jöhetett volna létre.

Függelék

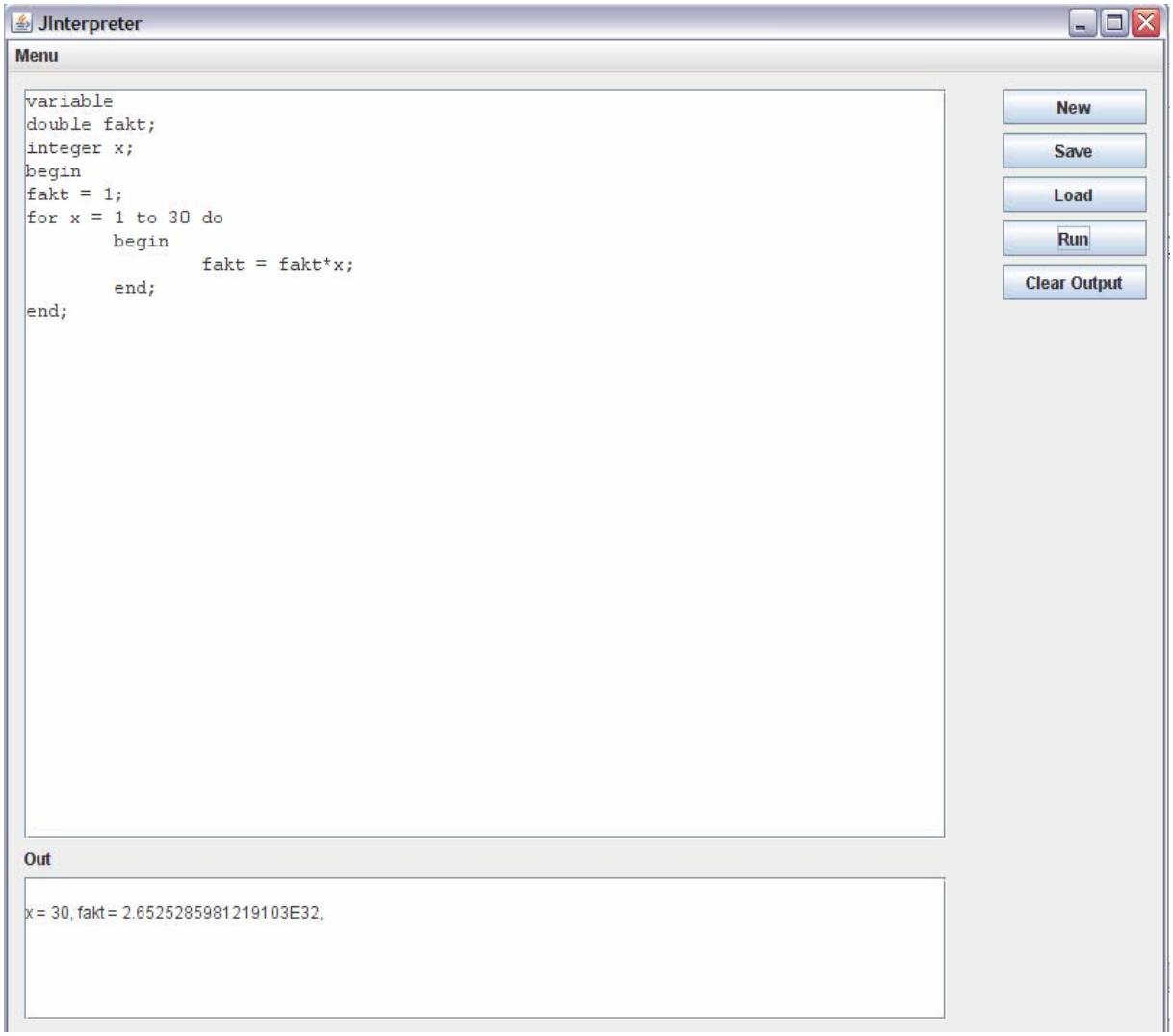
1. Ábra – A rendszerterv



2. Ábra – Kép a programból



Ábra 3.


JInterpreter
Menu
variable
double fakt;
integer x;
begin
fakt = 1;
for x = 1 to 30 do
begin
fakt = fakt*x;
end;
end;
New
Save
Load
Run
Clear Output
Out
x= 30, fakt= 2.6525285981219103E32,

Kalandozó magyarok

Írj kalandjátékot! A játék során a főszereplőnek egy küldetést kell teljesítenie. Ennek során egyszerű parancsokkal mozoghat a játék helyszínén, tárgyakat vehet magához, illetve egyéb, a konkrét játék által kitűzött feladatokat hajthat végre.

A játék használata

Minden játék rendelkezik a következő négy beépített paranccsal:

* megyek <merre>
* felveszem <tárgynév>
* leltár
* vége

A megyek hatására a játékos egy másik helyre jut, feltéve hogy az adott helyről abba az irányba el lehet jutni valahova. A helyváltoztatáson kívül a játék kiírja ilyenkor az új helyhez tartozó adatokat: a leírást, a lehetséges utakat (pl. ajtó, lépcső), és a földön fekvő tárgyakat. A felveszem hatására a játékos tárgyat felveszi, feltéve hogy a tárgy az azon a helyen van, és szerepel a használható tárgyak listájában (l. később). A leltár kilistázza a játékosnál lévő tárgyakat (amiket felvett eddig). A vége pedig befejezi a játékot. A fenti parancsok mellett használhatók az egyes játékok parancsai a következő formában:

* <parancsszó> <eszköz> <másik_valami>

Az ilyen parancsok akkor hajtódnak végre, ha a játék definiál <parancsszó> nevű parancsot, az <eszköz> a játékosnál van és a játékos a kiadáskor éppen a parancshoz szükséges helyen van (l. parancs megadása a .kal-ban).

A játék definíciója

A játékos által felhasználható tárgyak, a pálya, a tárgyak kezdeti helye és a használható parancsok definíciói egy .kal kiterjesztésű állományban vannak tárolva. Az állomány elején szerepel a játék során felhasználható tárgyak listája. Ezeket a játékos felveheti, és magával viheti. Ezután következik a térkép, amely helyek listájából áll. Minden helynek van egy neve, amely azonosítja, tartozik egy szöveges leírás hozzá, végül egy hármasokból álló lista írja le, melyik irányba, min keresztül, melyik helyre juthatunk el onnan. Ezután következik a tárgyak helyének megadása (tárgynév-helynév párosok listájaként), majd a játékos indulási helye. Az állomány végén a parancsok definíciói szerepelnek.

A szabályok feje négy névből áll, ezt követi a szabály törzse. Az első név a parancs neve, a második az eszközé, amivel az végrehajtható, a harmadik pedig azé a valamié, amin elvégezhető az előbbi eszközzel a parancs. Ez utóbbinak nem kell feltétlenül szerepelnie az állomány elején lévő tárgylistában. Végül a negyedik név egy hely neve. A szabály törzse kiíratást, feltétel vizsgálatát és logikai változó beállítását tartalmazhatja.

A parancsok törzsének szintaktikája és szemantikája

Négyféle utasítás létezik: kiíratás, logikai változó beállítása, feltételes utasítás és összetett utasítás. A szintaxisuk a következő:

* kiír <sztring>;
* igaz <azonosító>;
* ha <azonosító> akkor <összetett_utasítás> egyébként <összetett_utasítás>
* { [<utasítás>]... }

A parancsok törzse egy összetett utasítás. Az összetett utasítások végrehajtása a bennük szereplő alutasítások végrehajtását jelenti. A <kiír> utasítás a paraméterét és egy újsor karaktert a szabványos kimenetre írja. Az <igaz> utasítás egy logikai változót állít be. Arra jó, hogy meg tudjuk jegyezni, hogy a játékos már végrehajtta ezt a lépést. Végül a feltételes utasítás egy feltétel függvényében vagy az akkor vagy az egyébként utáni összetett utasítást hajtja végre. A feltétel egy tárgy vagy egy logikai változó lehet. Ha tárgy, akkor azt ellenőrzi, hogy a tárgy a játékosnál van-e, ha pedig nem, akkor azt, hogy lett-e már (igaz-zal beállítva ilyen nevű változó.

A szintaktika pontos definíciója: html, txt

Az <azonosító> Java azonosítót, a <sztring> pedig Java String literált jelöl.

A beküldött megoldásnak kompatibilisnek kell lennie ezzel a definícióval, de bővebb lehet. (Aki akarja, bonyolíthatja a játékot, csak az eddigiek működjenek.)
Példa

A feladathoz készített példában egy varázsló elhagyatott házában kalandozhatunk. Miután a nappaliban található láncot ráhegesztettük a vödörre, azt le tudjuk engedni a kútba, hogy megtöltsük vízzel. A vödör vízzel kell lelocsolnunk a varázslót ahhoz, hogy felébredjen mély álmából. (A békát nem szabad felvenni, különben megharagszik.)

* a játék definíciója html, kal
* a játék egy végigjátszása html, txt

Az, hogy a kimenetet milyen formában adjátok meg, teljesen rátok van bízva. A magyar nyelv toldalékai miatt elég nehéz helyes magyar mondatokban válaszolni a játékosnak, emiatt van a példaprogamban is pl. a "fel felé". Ez természetesen nem értékelési szempont.
Egyéb követelmények

* Java2 (1.2.0-tól 1.4.2-ig bármi)
* A program %bdquo;ne szálljon el” semmitől! Ha kivétel lép fel, azt kezeljétek le! A definíciós állományban lévő szintaktikai hiba esetén írjátok ki a megfelelő hibaüzenetet! Ha hibás parancsot ad meg a játékos, értelmes (magyar) hibaüzenet íródjék ki, ami pontosan megmondja a hiba okát, és lehessen folytatni a játékot!
* Írjatok ki promptot! (Nálam lemaradt. :-( )
* A saját kódotokat is lássátok el dokumentációs megjegyzésekkel! (Minden osztályt, adattagot és metódust!) Mintát találtok a KalandReader.java példában, bár egyszerű a szintakszis. Generáljatok API doksit a programotokból (javadoc), amin minden dokumentációs megjegyzés látszik (a privát tagok is)!

Megvalósítás

Mivel a feladat elég összetett, megpróbálok némi segítséget adni. Aki akarja, használhatja, természetesen nem kötelező. Aki segítség nélkül szeretné megoldani a feladatot, az ne olvasson tovább! ;-)

A definíciós állomány feldolgozása

Ez talán a feladat legnehezebb része. Az állomány tartalma alapján fel kell építeni egy összetett adatszerkezetet (Java Collection API), az állományra ezután már nincs szükség. Az állományt feldolgozó kód (parser) megírásához kiindulhattok a KalandReader osztályból. Az osztálydefiníció nem teljes, de tartalmazza az lexikális elemek (kapcsos zárójelek, vesszők, azonosítók és sztringek) beolvasásához szükséges kódot (lexer). A kódot dokumentációs megjegyzésekkel láttam el, és mindössze a konstruktor és négy metódus törzsét hagytam meg benne, amik a lexikai elemzést végzik.

A kalandot tároló objektum felépítése

A szintaktikai elemzéssel párhuzamosan kell felépíteni a kalandot tároló objektumot. Előtte, persze, definiálni kell a szükséges osztályokat. A fájl felépítéséből adódik, hogy milyen osztályokra van szükség, illetve a megadott kódban is szerepel néhánynak a neve. Figyeljetek rá, hogy az osztályok megvalósításához mindig olyan adatszerkezetet válasszatok, ami a későbbi munkát (a parancsok végrehajtását) a legkönnyebbé teszi! Ezzel rengeteg kódolást megspórolhattok.

A játék végrehajtása

A játék végrehajtásához gyakorlatilag egy nagyon egyszerű parancsértelmezőt (interpretert) kell megírnunk. Nem kell semmi ördöngősségre gondolni, hiszen a parancsok legfeljebb három szóból állhatnak, és nincs is túl sok fajta.

Irodalomjegyzék

1. Dömösi Pál - Fordítóprogramok, órai jegyzet, 2006

2. Varga László - Rendszerprogramozás

3. Simon Gyula - Számítástechnika

4. Juhász István - Programozás 1, Programozás 2, egyetemi jegyzet, 2004

5. Dr Dobb's magazin, 2006 szeptember – Multithreading in Java and OSGi

6. Dr Dobb's magazin, 2005 november – Functional programming in Java

7. http://www.prog.hu/cikkek/?aid=821

8. Dömösi Pál – Formális nyelvek és automaták, egyetemi jegyzet, 2003

9. Java Api