# Grafit
# gráf algoritmus interpreter
# nagyprogram dokumentáció

**Eötvös Loránd Tudományegyetem**
**Programtervező matematikus szak, nappali tagozat**
**Budapest, 2011**
**Konzulens: Kőhegyi János**

Készítette: Várkonyi Tibor Zoltán (VATNABI)

2011. január 19.

# Tartalomjegyzék

# 1. Bevezetés

Nagyprogramom célja, hogy egy hasznos és látványos szemléltető eszközt hozzak létre, amellyel tetszőleges – de főként gráfokon működő – algoritmus bemutatása egyszerű és könnyen megérthető legyen.
Ahhoz, hogy egy ilyen szemléltető eszköz működhessen, szükséges volt egy interpretálandó nyelvet választanom, amin a rendszer dolgozni tud. Megfelelő nyelvet sajnos nem találtam, ezért írtam egy saját nyelvet, amely minden elemet tartalmaz, amire az összetett feladatok megoldásakor támaszkodhatunk. Igyekeztem a lehető legegyszerűbb nyelvet létrehozni a célhoz mértan, azonban figyelembe kellett vennem, hogy a nyelv érthető maradjon általános programozási ismeretekkel rendelkezők számára.
Egy ilyen program elkészítésénél figyelembe kell venni a hordozhatóságot.
Ezen dokumentáció tartalmazza a nyelv leírását, valamint a beépített modulok teljes referenciáját.

# 2. Felhasználói dokumentáció

## 2.1. Bevezetés

A *Grafit* környezet gráf algoritmusok interpretálására és megjelenítésére szolgáló eszköz. A környezet három eszközt tartalmaz: egy soronkénti értelmezőt, egy interaktív konzolos felületet és egy interaktív grafikus felületet. A *Grafit* rendszer az interpreterből és a *Grafit* nyelvből áll.

## 2.2. Rendszerkövetelmények

Az alkalmazás futtatásához Java JRE szükséges. A futtatott algoritmusok bonyolultságától függően intenzív memória- és processzorhasználatot okozhat a program.

## 2.3. Telepítési útmutató

A programnak nincs szüksége telepítésre. A Java futtatókörnyezetet azonban telepíteni kell, amiről információk a Java honlapján olvashatók. Ha ez rendelkezésre áll, akkor csak el kell indítani az egyik .jar file-t.
Például:

```
> java -jar grafit.jar
```

## 2.4. Kezelés

### 2.4.1. Futtató felület

A futtató felület a legegyszerűbbrésze a *Grafit* környezetnek. Ez a környezet az előre megírt, *Grafit* nyelvű szintaktikusan helyes forráskódot futtatja le, majd ezután visszaadja a futtató operációs rendszernek a vezérlést.

```
> java -jar grafitscript.jar teszt.algo
```

| | |
|---|---|
| Processzor | 1.6 GHz Pentium IV |
| Memória | 256 MB DDR |

2.1. táblázat. Egy tesztelt konfiguráció

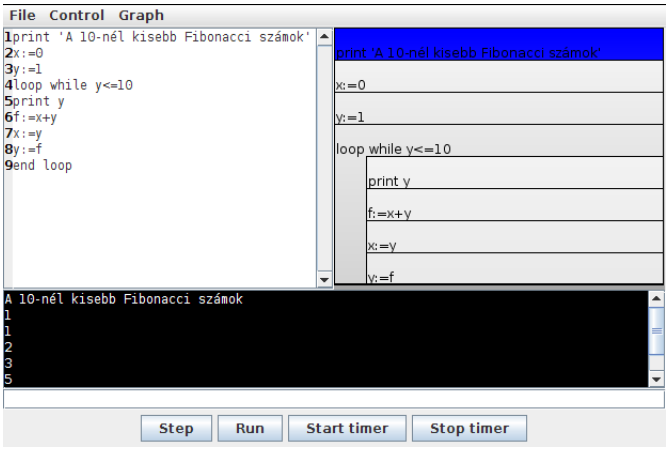
### 2.4.2. Konzolos felület

A konzolos interaktív felületen billentyűzetről adhatunk meg egyszerű utasításokat az értelmezőnek. Ilyenek a függvényhívások, az értékadások és az input/output műveletek.

### 2.4.3. Grafikus felület

A grafikus felület öt jól elkülöníthető részből áll.

1. Programkód
2. Struktogram
3. Konzol output
4. Konzol input
5. Vezérlő gombok




2.1. ábra. Felhasználói felület

A programkód egy sorszámozott szerkesztőmezőben írható és olvasható. A hibás programsorokat az értelmező pirossal jelöli meg a számmezőben. A struktogram a helyes programsorokból készít diagrammot, amin követhetjük az algoritmus futását. A konzol outputon a `print` utasítás hatására megjelenő információ olvasható a program futása során. A konzol input működése megegyezik a konzolos felület működésével. A vezérlő gombok segítségével lehetőségünk adódik az algoritmus léptetésére, lefuttatására és időzített működtetésére.

# 3. Fejlesztői dokumentáció

## 3.1. A választott implementáció

A program *Java* nyelven készült, a `java.awt.swing` komponenseket felhasználva. Alapvető fontosságú volt a hordozhatóság szempontjából, hogy ezt a környezetet válasszam, mivel ez a környezet rengeteg platformon implementált. A hatékonyság csak másodlagos szempont volt, mivel egy ilyen rendszernek nem célja, hogy az interpretált algoritmusokat hatékonyan hajtsa végre, sokkal inkább azok szerkezetének, működésének bemutatása a cél.
Awt formokkal bővíthető a rendszer.

## 3.2. A Grafit nyelv

A *Grafit* nyelvet a pszeudokód ihlette. Célja, hogy minél absztraktabb módon lehessen vele algoritmusokat leírni. Ugyanakkor jónéhány érdekes nyelvi konstrukciót tartalmaz, amely a nyelvleírás egyszerűsítésének következménye.
A nyelv kidolgozásakor figyelembe kellett vennem, hogy egy `LL(1)`[1] nyelvtanról van szó, ezért kerültek bele a `valueof` és `imageof` kulcsszavak – amelyek az implementációban új lehetőségeket nyitottak.

### 3.2.1. Újdonságok a nyelvben

`valueof, value`

Ilyen például az, hogy függvényhíváskor a függvény visszatérési értéke a `value` nevű változóba automatikusan bekerül. Ez egy interpreternél nagyon hasznos funkció lehet. Tegyük fel, hogy van egy bonyolult függvény, amelynek sok paramétere van. Ezt egyszer meghívjuk az értelmezőből, de a visszatérési értéket elfelejtettük lementeni. Ilyenkor egyszerűen csak a `value` változó értékét kell kiolvasni, és nincs szükség az újbóli futtatásra.

#### `->`, implikáció

A nyelvbe bevettem az implikációs műveletet, amely jobb asszociatív. (Valójában az összes művelet jobb asszociatív a nyelvben.)

#### Tömbkezelés

Igen sokoldalú szintaxist adtam a tömbök használatára (bár ezek nem túl hatékonyak). A tömböket a "[ ]" közé írt egész számokkal használhatjuk. Valójában a rendszer az egész szám értékét egy elválasztóval hozzákonkatenálja változó nevéhez. A valódi újdonság a multitömbök (mátrixok) kezelésekor tapasztalható.
A multitömb indexelés a *C* nyelvben megszokott.
A mátrix indexelés kicsit közelebb áll a matematikai megközelítéshez.
Ezen kívül létezik a vegyes indexelés, amely keveri a két szintaxist.

```
1 a[0]:=1 # egyszeru vektor
2 a[0][0]:=2 # multitomb
3 a[0,0]:=3 # matrix
4 a[0][0,0]:=4 # mix-tomb
5 print a[0]+a[0][0]+a[0,0]+a[0][0,0]
```

3.1. ábra. tomb.algo – Tömbök típusai

### 3.2.2. Működés

Az értelmező soronként értelmezi és hajtja végre a programokat. Ha ezek a sorok mind szintaktikusan helyesek, akkor az elemzés után a program összeszerkeszti a sorokat. Ilyenkor ugrópointereket állít be a szerkesztő a sorok között. Valójában minden sornak van visszatérési értéke. Ha ez az érték 0, akkor a hamis pointer aktivizálódik, különben az igaz pointer. Az aktív pointer lesz mindig a következő utasítás.
Minden sorban szerepelhet

- utasítás-sorozat
- vezérlő szerkezet
- I/O művelet

Sor végéig tartó megjegyzéseket akármelyik sor végére tehetünk.

### 3.2.3. Utasítások

#### Értékadás

A `:=` szimbólum bal oldalán szereplő balérték értékül kapja a szimbólum jobb oldalán szereplő értéket. Ilyen érték lehet egy művelet eredménye, vagy a `valueof` kulcsszó után egy függvény visszatérési értéke.

#### Függvényhívás

Eljárásként hívhatjuk a könyvtári függvényeket a `call` kulcsszóval. Ilyenkor (ha van), a függvény visszatérési értéke a `value` lokális változóba kerül. Értékadás jobb oldalán is szerepelhet függvényhívás, ilyenkor a `valueof` kulcsszót kell használni.

#### Visszatérés

A script függvény értékének visszaadására szolgáló utasítás. A szintaxisban szerepel, de a jelenlegi változat még nem tartalmaz script függvény kiválasztást - így ennek a szintaktikus elemnek csak a konzolos eszközben van jelentősége.

### 3.2.4. Utasítás-sorozatok

Az utasításokat a ; szimbólummal *választjuk el* egymástól. A szekvencia szimbólum csökkenti az interpreter szerkesztési idejét, de nincs hatással az elemzési sebességre.

### 3.2.5. Vezérlő szerkezetek

A vezérlő szerkezetek alkalmazásával strukturált végrehajtást használhatunk a nem interaktív felületeken. (A grafikus eszköz pufferében és a script futtató eszközben.)
Minden vezérlőnek van megfelelő lezáró párja. Így nem probléma a nyelvben a csellengő else – minden `if` egy `end if` vezérlővel végződik. Valójában az `end` érzéketlen arra, hogy mi áll utána. A szerkesztő ezt az információt nem veszi figyelembe, de erről majd még később esik szó.

| Vezérlő | | Lezárás | Leírás |
|---|---|---|---|
| `if <feltétel> then` | `else` | `end if` | Végrehajtja az `if-else` közötti részt, ha *`feltétel`* igaz. Ellenkező esetben az `else-end if` közötti rész hajtódik végre. |
| `loop while <feltétel>` | | `end loop` | A `loop-end loop` közötti részt (ciklusmag) mindaddig lefuttatja, amíg *`feltétel`* teljesül. |
| `procedure` *`név`* | `(<paraméter-lista>)` | `end procedure` | Eljárás-blokkot hoz létre. Az eljárás blokk elmenti a lokális változókat, majd a blokkból való kilépéskor visszatölti azokat. Egyelőre nincs megfelelően implementálva. |

## 3.3. Modulok

### 3.3.1. Beépített modulok

#### Global

A globális modul globális változókat kezel. A globális modul mindenhonnan elérhető, és minden tekintetben úgy működik, ahogyan ezt egy globális változótól elvárható.

#### Local

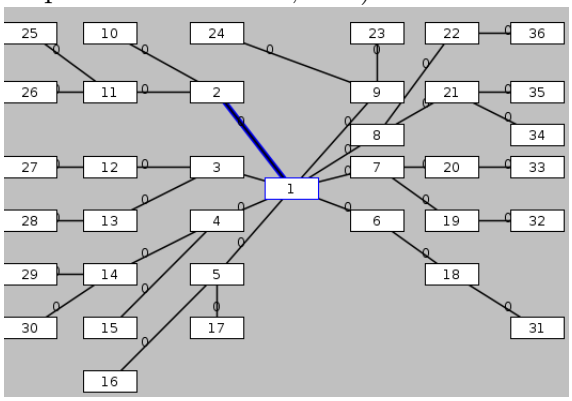
A lokális modul a globális modul kiterjesztése. Ez a modul az alapértelmezett, ha nem adunk modulválasztást, akkor a lokális modul szemantikai szabályai lesznek érvényesek. A lokális modul rendelkezik saját állapottérrel, amit függvényhíváskor a környezet elment, visszatéréskor pedig visszatölt (egy verem szerkezettel).

### 3.3.2. Mellékelt modulok

A *Grafit* környezet részeként megírtam néhány - a szemléltetéshez nélkülözhetetlen - modult.
Ezek a modulok természetesen nem a leghatékonyabb megoldásokat tartalmazzák – ezért nem kötelező használni ezeket. Bármely modul lecserélhető, készíthető mindegyiknél szebb és jobb implementáció.

#### Graph

Ez a modul a *Grafit* rendszer leglényegesebb része. Ebben a modulban gráfalgoritmusok futtatására alkalmas függvények, valamint egy gráf megjelenítő komponens található. A megjelenítő egy egyszerű `awt` form, amelyen egyszerű megjelentítést alkalmaztam. A formnak számos beállítása van, azonban ezeket nem vezettem ki a nyelv felé. Ezeket csak a forráskódban lehet megváltoztatni. (Ilyen például a gráfcsúcsok formája, alapértelmezett színe, stb.)



| Függvény neve | Bemenő paraméterek | Visszatérési érték | Leírás |
|---|---|---|---|
| init | | | Inicializálja a gráfot. Minden csúcsot és élt töröl |
| selectNode | érték | | Kiválasztja a megadott indexű csúcsot |
| selectEdge | érték | | Kiválasztja a megadott indexű élt |
| getNode getNodeValue | érték | érték | Visszaadja a megadott csúcs értékét |
| getEdgeValue | érték | érték | Visszaadja a megadott él értékét |
| addNode | csúcs értéke, x=0, y=0, szín=`$MColor.white` | csúcs értéke | Beilleszt egy új csúcsot a megadott paraméterekkel |
| setNode setNodeValue | érték | | Beállítja a megadott csúcs értékét |
| getEdge setEdgeValue | érték | érték | Beállítja a megadott él értékét |
| addEdge<br><br><br>setX | csúcs1,csúcs2<br><br><br>érték | | A két megadott csúcs között élt hoz létre<br>A kiválasztott csúcs X koordinátáját állítja be |
| setY | érték | | A kiválasztott csúcs Y koordinátáját állítja be |
| setColor | érték | | A kiválasztott csúcs színét állítja be |
| setEdgeColor | érték | | A kiválasztott él színét állítja be |

| Függvény neve | Bemenő paraméterek | Visszatérési érték | Leírás |
|---|---|---|---|
| getX | érték | | A kiválasztott csúcs X koordinátáját adja vissza |
| getY | érték | | A kiválasztott csúcs Y koordinátáját adja vissza |
| getColor getNodeColor | érték | | A kiválasztott csúcs színét adja vissza |
| getEdgeColor | érték | | A kiválasztott él színét adja vissza |
| getEdgeCount | érték | | A kiválasztott csúcsból induló élek számát adja vissza |
| nodeCount getNodeCount | | | A csúcsok számát adja vissza |
| getStartNode | érték | | A megadott él kezdőpontját adja vissza |
| getEndNode | érték | | A megadott él végpontját adja vissza |
| getEdge | csúcs, élindex | | A megadott csúcsból induló i-edik élt adja vissza |

### String

A String modul a nyelv szemantikájának kiterjesztésére szolgál.

| Függvény neve | Bemenő paraméterek | Visszatérési érték | Leírás |
|---|---|---|---|
| concat | string1 [,string2,...] | A konkatenált szöveg | Összefűzi az összes paraméter string értékét |
| set | név, érték=” | string | A megadott nevű string értékét beállítja az értékként megadott stringre |
| get | név | string | Visszaadja a megadott nevű string értékét |
| substring | név, honnan=0, mennyit=string hossza | string érték | A megadott nevű string részét adja vissza |
| length | név | érték | A megadott nevű string hosszát adja vissza |

### Random

| Függvény neve | Bemenő paraméterek | Visszatérési érték | Leírás |
|---|---|---|---|
| get | értékhatár1=0, értékhatár2=0 | érték | Az értékhatárok között megad egy véletlen számot. |

### Stack

| Függvény neve | Bemenő paraméterek | Visszatérési érték | Leírás |
|---|---|---|---|
| push | név,érték=0 | érték | A megadott nevű verembe beteszi az értéket |
| pop | név | érték | Visszaadja a megadott nevű veremből a legfelső értéket |

### Queue

| Függvény neve | Bemenő paraméterek | Visszatérési érték | Leírás |
|---|---|---|---|
| push | név,érték=0 | érték | A megadott nevű sorba beteszi az értéket |
| pop | név | érték | Visszaadja a megadott nevű sorból a legelőször betett értéket |

### PQueue

| Függvény neve | Bemenő paraméterek | Visszatérési érték | Leírás |
|---|---|---|---|
| push | név, prioritás=0, érték=0 | érték | A megadott nevű prioritásos sorba beteszi az értéket a megfelelő prioritással |
| pop | név | érték | Visszaadja a megadott nevű sorból a legelőször betett legnagyobb prioritású értéket |

**Color**

### 3.3.3. Modulok készítése

Modulokat Java nyelven kell a nyelvhez készíteni. Minden modul megvalósítja az `ITreeNode` interfészt. Legegyszerűbb a `mocman.interpreter.syntax.Modul` modult kiterjeszteni, mivel ez az osztály már rendelkezik a legtöbb implementációval, amire egy modulnak szüksége lehet. A helyes szemantikai kiértékelés érdekében a `clone()` és a `value()` függvényeket mindenképpen terheljük túl!

Három függvényt ajánlott túlterhelni az új modulokban. Az egyik függvény a `function()`, ami a szintaxis szerinti függvényhívást értelmezi. A másik két függvény a szintaxis szerinti értékadást és értékolvasást valósítja meg. Az értékadás a `set()`, az értékolvasás pedig a `get()` függvény végrehajtása során értékelődik ki. Valójában ezek a függvények adják az új szemantikai elemeket a *Grafit* nyelvhez.

```
1  package mocman.interpreter.modules;
2  import mocman.interpreter.syntax.*;
3
4  public class MyModule extends Module
5  {
6      public MyModule clone()
7      {
8          MyModule g=new MyModule();
9          g.setValue(getVal());
10         return g;
11     }
12
13     public ITreeNode value(String s)
14     {
15         MyModule l=new MyModule();
16         l.setValue(s);
17         return l;
18     }
19     [...]
20 }
```

3.2. ábra. Sample module

### 3.3.4. Scriptek

## 3.4. Tesztelés

### 3.4.1. Script eszköz

A script eszközt a `fibonacci.algo` algoritmussal és a `rekmely.algo` algoritmusokkal teszteltem.

#### Nem-Fibonacci számok

A következő algoritmus egy megadott intervallumban kiírja az összes egész számot, amelyek nem Fibonacci számok. Ebben a programban értékadások, szekvenciák és egymásba ágyazott ciklusok és feltételek vannak. A program egy futása:

```
1  x := 0; y := 1; c := 0
2  hatar := input 'Ertekhatar?␣'
3  loop while c <= hatar
4      loop while c < y and  c <= hatar
5          print c
6          c := c + 1
7      end loop
8      if c<=y then
9          c := c + 1
10     end if
11     f := x + y
12     x := y
13     y := f
14 end loop
```

3.3. ábra. fibonacci.algo – Nem Fibonacci számok

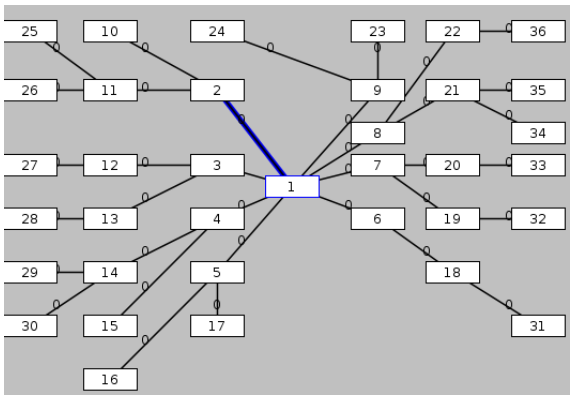
```
Értékhatár? 10
0
4
6
7
9
10
```

Teszteltem 100 és 1000 értékhatárra is.

#### Rekurzív mélységi bejárás

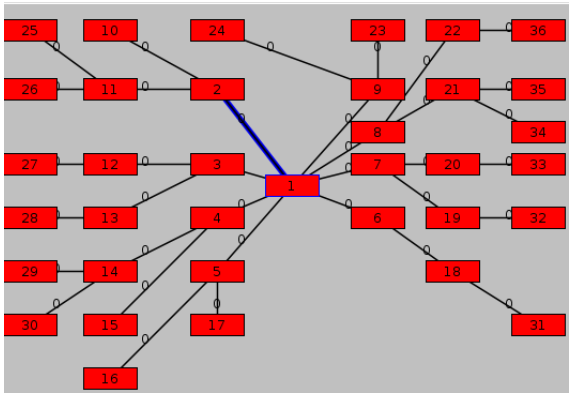
A második teszt egy mélységi bejárás rekurzív hívással. [2]
Először előállítom a gráfot:

```
call $Snagygraf.run()
```



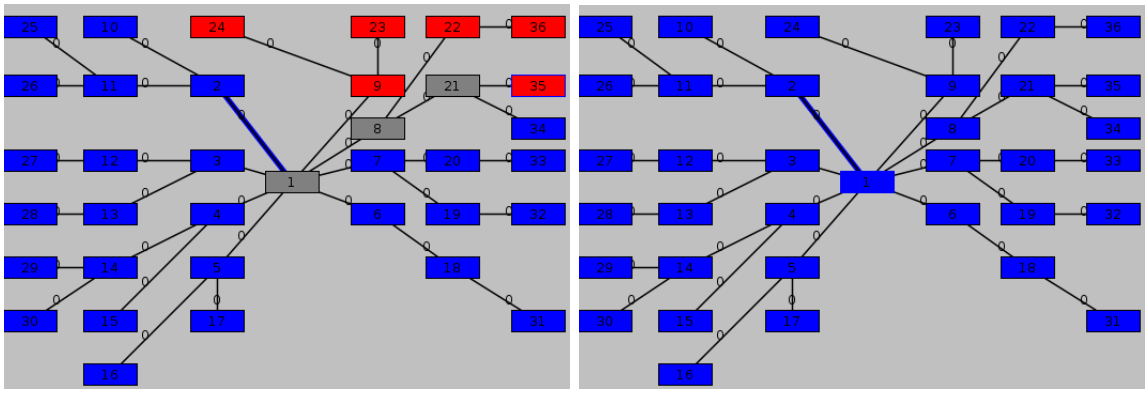

Beszínezek minden csúcsot pirosra:

```
call $Spiros.run()
```




Lefuttatom a mélységi bejárást:

```
call $Smelysegi.run()
```



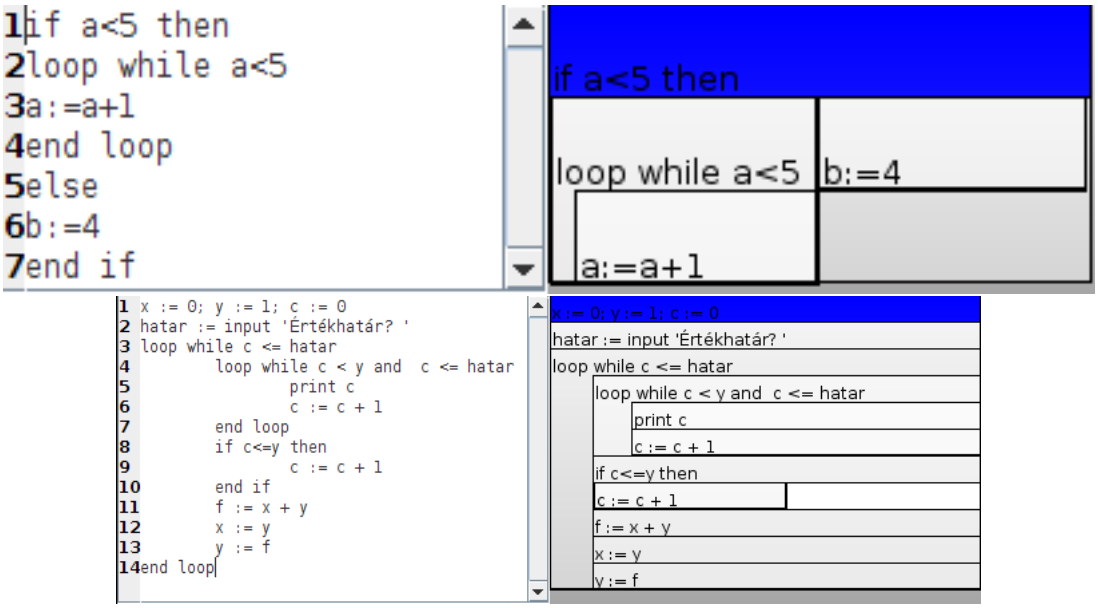

Futás közben jól látszik, hogy az algoritmus az elvárt viselkedést produkálja.

### 3.4.2. Grafikus felület

A grafikus felület ugyanazt az osztályt használja, mint a script, csak a futáshoz grafikus komponenseket biztosít. A legfőbb tesztelendő komponens a struktogram megje-

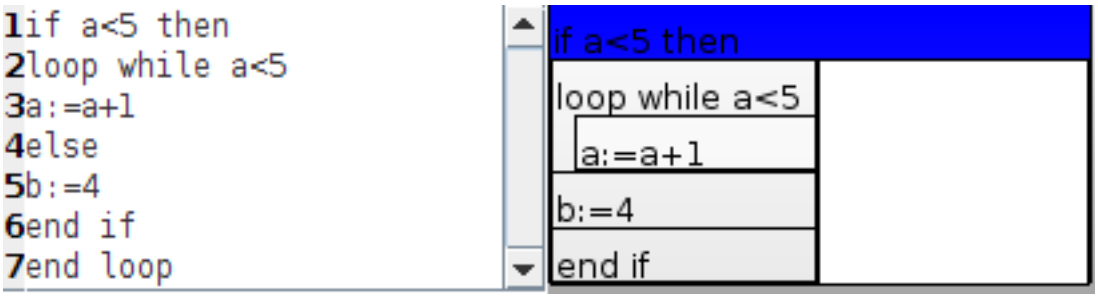
lenítő. Minden szerkezetet ismernie kell, tetszőlegesen egymásba ágyazva - helyes program esetén. Helytelen program esetén ugyan nem definiált a kimenet, viszont nem okozhat problémát a hibás szintaxis! Ezért első menetben szintaktikusan helyes programokat tesztelek.

Például:




Második menetben szintaktikusan hibás programokat tesztelek. A szintaktikailag hibás sorokat figyelmen kívül hagyja a megjelenítő. A rosszul összekötött programszerkezetek okozhatnak hibát. A rosszul felépített program (például if...loop while...end if...end loop nem okoz problémát, viszont lefuttatni érelmesen nem lehet.

Például:



# A. BNF leírás

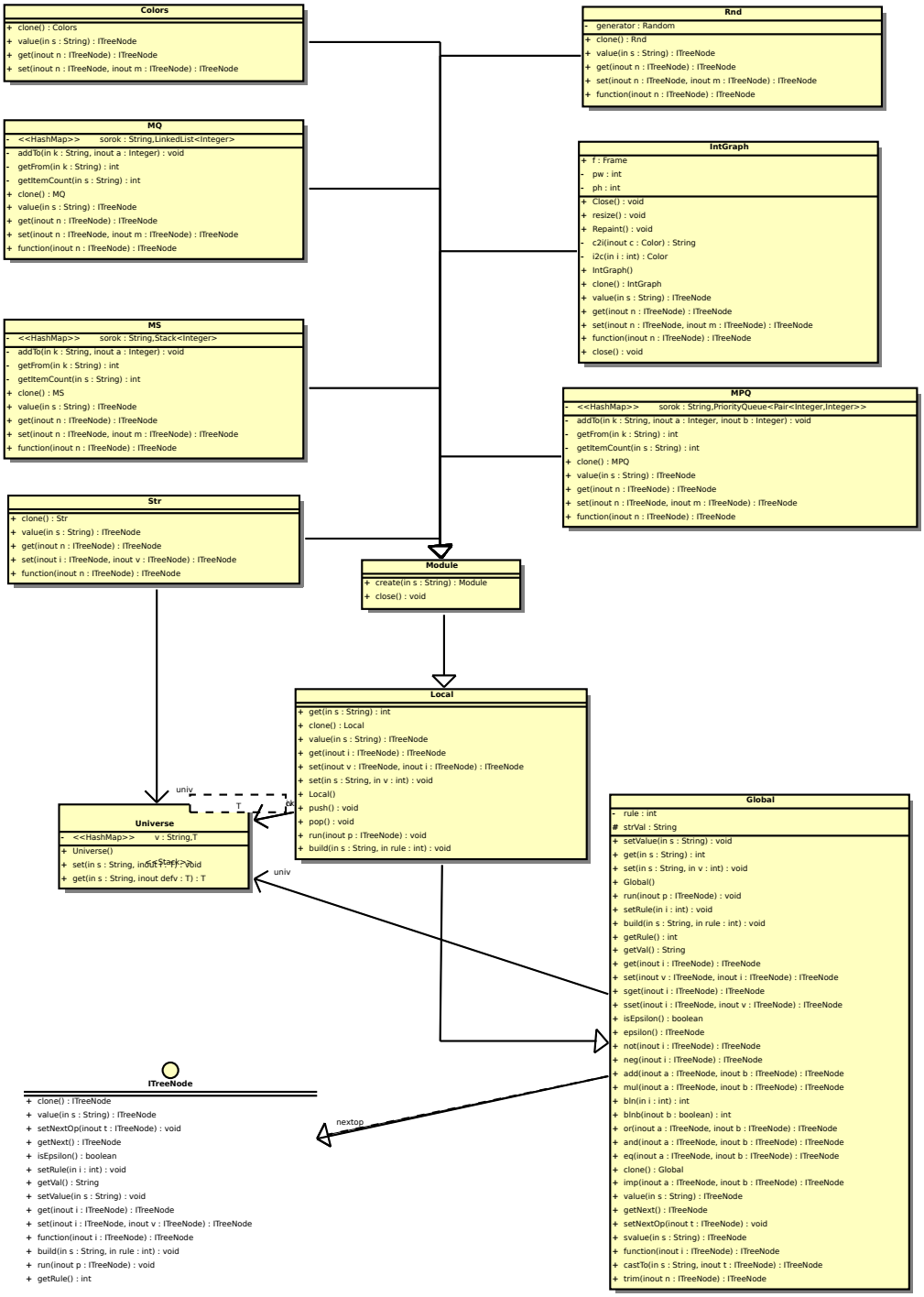
```
<INTEGER> ::= <INTIMP>
<INTIMP> ::= <INTOR> <INTIMP1>
<INTIMP1> ::= "->" <INTOR> <INTIMP1>
<INTIMP1> ::= <epsilon>
<INTOR> ::= <INTAND> <INTOR1>
<INTOR1> ::= "or" <INTADD> <INTOR1>
<INTOR1> ::= <epsilon>
<INTAND> ::= <INTEQ> <INTAND1>
<INTAND1> ::= "and" <INTEQ> <INTAND1>
<INTAND1> ::= <epsilon>
<INTEQ> ::= <INTADD> <INTEQ1>
<INTEQ1> ::= <EQOP> <INTADD> <INTEQ1>
<INTEQ1> ::= <epsilon>
<INTADD> ::= <INTMUL> <INTADD1>
<INTADD1> ::= <ADDOP> <INTMUL> <INTADD1>
<INTADD1> ::= <epsilon>
<INTMUL> ::= <INTN> <INTMUL1>
<INTMUL1> ::= <MULOP> <INTN> <INTMUL1>
<INTMUL1> ::= <epsilon>
<INTN> ::= "not" <INTV>
<INTN> ::= <INTV>
<INTV> ::= "\\(" <INTEGER> "\\)"
<INTV> ::= <INT>
<INTV> ::= "-" <INT>
<MULOP> ::= "\\*"
<MULOP> ::= "/"
<MULOP> ::= "%"
<ADDOP> ::= "\\+"
<ADDOP> ::= "-"
<EQOP> ::= "="
<EQOP> ::= "not" "="
<EQOP> ::= "<="
<EQOP> ::= ">="
<EQOP> ::= "<"
<EQOP> ::= ">"
<INT> ::= "0"
<INT> ::= "[1-9][0-9]*"
```

```
<INT> ::= "true"
<INT> ::= "false
<INT> ::= <LEFTVALUE>
<INT> ::= "valueof" <LEFTVALUE> <ACTPARAMLIST>
<INT> ::= "input" <INPUTPARAM>
<INPUTPARAM> ::= <epsilon>
<INPUTPARAM> ::= <STRING>
<STR> ::= "\'[^\']*\'"
<LEFTVALUE> ::= <MODULE> <ARRAY>
<MODULE> ::= "\\$" <IDENTIFIER> "\\." <IDENTIFIER>
<MODULE> ::= <IDENTIFIER>
<ARRAY> ::= "\\[" <ARRAYB> "\\]" <ARRAY>
<ARRAY> ::= <epsilon>
<ARRAYB> ::= <INTEGER> <ARRAYB'>
<ARRAYB'> ::= <epsilon>
<ARRAYB'> ::= "," <INTEGER>
<IDENTIFIER> ::= "[a-zA-Z][a-zA-Z0-9_]*"
<IF> ::= "if" <INTEGER> "then"
<ELSE> ::= "else"
<ENDSOMETHING> ::= "if"
<WHILE> ::= "loop" "while" <INTEGER>
<ENDSOMETHING> ::= "loop"
<EVALCALL> ::= ":=" <INTEGER>
<EVALCALL> ::= <ACTPARAMLIST>
<EVAL> ::= <LEFTVALUE> <EVALCALL>
<EVAL> ::= "call" <LEFTVALUE> <ACTPARAMLIST>
<ACTPARAMLIST> ::= "\\(" <ACTPARAM'> "\\)"
<ACTPARAM'> ::= <ACTPARAMS>
<ACTPARAM'> ::= <epsilon>
<ACTPARAMS> ::= <INTEGER> <ACTPARAMLISTPLUS>
<ACTPARAMS> ::= <STRING> <ACTPARAMLISTPLUS>
<ACTPARAMLISTPLUS> ::= "," <ACTPARAMS>
<ACTPARAMLISTPLUS> ::= <epsilon>
<PROGRAM> ::= "procedure" <IDENTIFIER> <FORMPARAMLIST>
<FORMPARAMLIST>::= "\\(" <FORMPARAM'> "\\)"
<FORMPARAM'> ::= <FORMPARAMS>
<FORMPARAM'> ::= <epsilon>
<FORMPARAMS> ::= <FORMPARAMID> <FORMPARAMPLUS>
<FORMPARAMID> ::= <IDENTIFIER>
<FORMPARAMID> ::= "!" <EVAL>
<FORMPARAMPLUS> ::= "," <FORMPARAMS>
<FORMPARAMPLUS> ::= <epsilon>
<RETURN> ::= "return" <RETURNSTATE>
<RETURNSTATE> ::= <epsilon>
```

```
<RETURNSTATE> ::= <INTEGER>
<SEQUENTIAL> ::= <SEQFIRST> <SEQNEXT>
<SEQNEXT> ::= ";" <SEQUENTIAL>
<SEQNEXT> ::= <epsilon>
<SEQFIRST> ::= <RETURN>
<SEQFIRST> ::= <EVAL>
<STATEMENT> ::= <IF>
<STATEMENT> ::= <ELSE>
<STATEMENT> ::= <WHILE>
<STATEMENT> ::= <PROGRAM>
<STATEMENT> ::= <SEQUENTIAL>
<STATEMENT> ::= "end" <ENDSOMETHING>
<STATEMENT> ::= "print" <PRN>
<STATEMENT> ::= "message" <PRN>
<STATEMENT> ::= "pause"
<ENDSOMETHING> ::= "procedure"
<PRN> ::= <INTEGER>
<PRN> ::= <STRING>
<PRN> ::= <epsilon>
<STRING> ::= <STR>
<STRING> ::= "imageof" <IMG>
<IMG> ::= <LEFTVALUE> <ACTPARAMLIST>
<epsilon> ::=
```

# B. UML diagramm

## B.1. Modulok szerkezete



# Irodalomjegyzék

[1] Csörnyei Zoltán, *Fordítóprogramok*. Typotex 2006.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Új algoritmusok*. Scolar Informatika 2003.

[3] M.G.J. van den Brand , C. Groza, *The Algebraic Specification of Annotated Abstract Syntax Trees*. `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.3593` 1994.