EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Algoritmusok és alkalmazásaik Tanszék

Programozási nyelv és környezet készítése számítási modellek integrált implementálására

Tichler Krisztián Steiner Ákos Pál

egyetemi adjunktus programtervező informatikus

Budapest  
2022

Tartalom

[Bevezetés - 4 -](#_Toc103533909)

[Motiváció - 4 -](#_Toc103533910)

[Számítási modellek - 4 -](#_Toc103533911)

[Determinisztikus automaták - 4 -](#_Toc103533912)

[Modellek leírása - 5 -](#_Toc103533913)

[Felhasználói dokumentáció - 7 -](#_Toc103533914)

[A szoftver célja - 7 -](#_Toc103533915)

[Rendszerkövetelmények - 7 -](#_Toc103533916)

[Telepítés - 7 -](#_Toc103533917)

[A szoftver használata - 8 -](#_Toc103533918)

[Felhasználói felület - 8 -](#_Toc103533919)

[Hibaüzenetek - 14 -](#_Toc103533920)

[A nyelv leírása - 16 -](#_Toc103533921)

[Lexikális elemek - 16 -](#_Toc103533922)

[Szintaxis - 18 -](#_Toc103533923)

[Modellek - 18 -](#_Toc103533924)

[Turing gépek - 19 -](#_Toc103533925)

[Hangyamodellek - 19 -](#_Toc103533926)

[Egydimenziós sejtautomaták - 20 -](#_Toc103533927)

[Kétdimenziós sejtautomaták - 20 -](#_Toc103533928)

[Fejlesztői dokumentáció - 21 -](#_Toc103533929)

[Megoldási terv - 21 -](#_Toc103533930)

[Programszerkezet - 21 -](#_Toc103533931)

[Megvalósítás - 24 -](#_Toc103533932)

[Fordító - 24 -](#_Toc103533933)

[Szoftver - 26 -](#_Toc103533934)

[Modellfuttatás - 28 -](#_Toc103533935)

[Tesztelés - 29 -](#_Toc103533936)

[Fordító tesztelése - 29 -](#_Toc103533937)

[Helyes tesztek - 29 -](#_Toc103533938)

[Helytelen tesztek - 30 -](#_Toc103533939)

[Felhasználói felület tesztelés - 33 -](#_Toc103533940)

[Menü tesztelése - 33 -](#_Toc103533941)

[Kódszerkesztő tesztelése - 33 -](#_Toc103533942)

[Parancssor tesztelése - 33 -](#_Toc103533943)

[Sejtautomata felület tesztelése - 33 -](#_Toc103533944)

[Turing-gép felület tesztelése - 33 -](#_Toc103533945)

[Továbbfejlesztés - 34 -](#_Toc103533946)

[Implementálandó modellek - 34 -](#_Toc103533947)

[Neurális sejtautomaták - 34 -](#_Toc103533948)

[Blokk automaták - 34 -](#_Toc103533949)

[Membránrendszerek - 35 -](#_Toc103533950)

[Program funkciók - 35 -](#_Toc103533951)

[Összefoglalás - 36 -](#_Toc103533952)

[Irodalomjegyzék - 36 -](#_Toc103533953)

# Bevezetés

## Motiváció

Számításelméletet tanulva az ember gyakran találkozik különböző számítási modellek alkalmazásával, amire jó példa egyetemen belül is a *Számítási modellek* tárgyak, melynek során megtanulunk többdimenziós sejtautomatákat, Turing-gépeket és egyéb, komplex szabályrendszerű modelleket írni, ám ezekhez - tapasztalat alapján - nem alkalmaznak egységes keretrendszert, szoftvert. Házidolgozatok leadásakor a hallgatók különböző online felületeken szerkesztett modelleket, kódokat, esetenként képernyőképet küldenek az oktatónak, így nehezítve meg mind a dolgozat leadásának folyamatát, mind az oktató dolgát a dolgozat javításakor. A ***CELL*** programozási nyelv és a hozzá tartozó ***CellDeveloper*** kódszerkesztő és fordító szoftver ezen nehézségek kiküszöbölésére szolgál, de természetesen bárkinek, aki számítási modellekkel szeretne foglalkozni, kényelmes munkakörnyezetet és gyors, egyszerű kódalkotást biztosít.

## Számítási modellek

Gyakran találkozhatunk olyan számítási nehézségekkel, melyek modellalkotás nélkül nehezen megoldhatók, értelmezésükhöz, áttekinthetőségük javításához célszerű absztrakciót létrehozni. Számításelméleti és programozási problémák vizsgálatához erre a célra létrehozott, a számítás alapelveiből alkotott absztrakciókat számítási modelleknek. Az egyik leggyakrabban használt ilyen modellek, melyekkel az ember egyetemi tanulmányai során is többször találkozik, a determinisztikus automaták.

### Determinisztikus automaták

A [számítástudományban](https://hu.wikipedia.org/wiki/Sz%C3%A1m%C3%ADt%C3%A1studom%C3%A1ny) a determinisztikus véges állapotú gép vagy determinisztikus véges állapotú automata egy [véges állapotú gép](https://hu.wikipedia.org/w/index.php?title=V%C3%A9ges_%C3%A1llapot%C3%BA_g%C3%A9p&action=edit&redlink=1), ahol minden állapot–bejövő szimbólum párhoz egy és csakis egy másik állapotba való átmenet tartozik. A **CellDeveloper** szoftver lehetőséget kínál ilyen determinisztikus automaták meghatározott típusaihoz könnyen programozható példányokat alkotni. Ezen modell típusok az alábbiak:

1. Turing-gépek
2. Hangya modellek
3. Egydimenziós sejtautomaták
4. Kétdimenziós sejtautomaták

### Modellek leírása

#### Turing-gépek

Alapelve, hogy véges szalagon különböző karakterek találhatók, valamint minden szalaghoz tartozik egy **író-olvasófej**. A szalagokon található **karakterek** mellett futás közben egy **állapot változó** is jelen van. Egy adott iteráció jelenlegi állapota attól függ, hogy az olvasófejek épp milyen karaktereket olvasnak a szalagokról, valamint az állapot változó jelenlegi értékétől. Az állapottól függően az író-olvasófejek módosítják az épp beolvasott karaktereket a szalagon, **majd vagy jobbra, vagy balra** mozdulnak el egy karakterrel, vagy helyben maradnak, az állapot változó pedig a szabályban hozzárendelt értékre módosul.

A szoftverben csak *lexikálisan* elfogadott karakterek kerülhetnek a szalagokra, melyeknek száma maximum 4 lehet. A szalagok hossza mindkét irányba véges.

#### Hangya modellek

A hangya modellek speciális Turing-gépek, ahol nem egy szalagon, hanem egy **kétdimenziós rácson** mozog az író-olvasófej és iterációnként **csak jobbra, vagy balra** fordulva tud előre mozdulni a következő mezőre.

A programon belül meghatározatlan számú állapot definiálható, melyeknek értékei **egészszámok**. Minden példány futásakor az olvasófej a rács **középpontjából indul** **el**, és ha eléri a rács szélét, megáll.

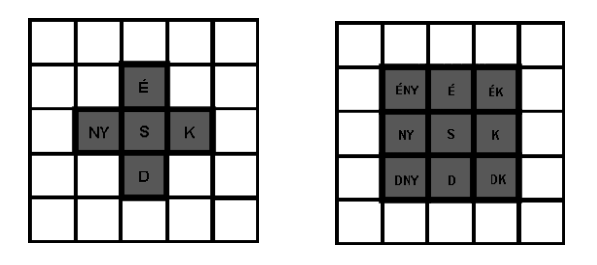
#### Egydimenziós sejtautomaták

Az egydimenziós sejtautomaták szintén a Turing-gépek speciális fajtája. Egy szalagon minden mező értéke attól függ, hogy az előző iterációban a **két szomszédos mezőnek** és **önmagának** mi volt az értéke – ezt egy számhármas reprezentálja. Mivel bináris automata, minden érték **vagy 0, vagy 1** lehet, így a lehetséges szabályrendszerek száma is véges.

A programban az egydimenziós automaták szintén rácsban vannak ábrázolva, ahol minden sor egy iterációt jelent. Kezdő állapotában csak a szalag középső értéke 1, a többi 0.

#### Kétdimenziós sejtautomaták

Az egydimenziós automatákhoz hasonlóan a kétdimenziós automaták mezői is **önmaguk és szomszédjaik** korábbi állapotától függően veszik fel új értéküket, mely a szoftveren belül szintén **vagy 0, vagy 1** lehet. Szomszédság alatt **Moore-szomszédságot** értünk, mely szerint a Neumann szomszédságtól eltérően minden mezőnek nyolc szomszédja van, néggyel csak közös sarkon, néggyel közös élen is osztozik.



. ábra: Balra: Neumann-szomszédság, jobbra: Moore-szomszédság

A programban példányfuttatáskor bal egérgombbal lehet egy mező értékét 1-re váltani, míg jobb egérgombbal 0-ra.

# Felhasználói dokumentáció

## A szoftver célja

A szoftver célja számítási modellek gyors, átlátható és egyszerű implementálása CELL nyelv segítségével, melynek megtervezése során a rövid, könnyen értelmezhető szintaxis és intuitív, dinamikus programozás lehetősége volt előtérben. Erre azért volt szükség, mert számítási modellekkel nem csak programozók dolgoznak, hanem matematikusok, logikusok is, akikkel szemben nem elvárhatóak komplexebb programozói ismeretek. A kódszerkesztő felületen megírt, dinamikusan frissülő lexikális szövegkiemeléssel ellátott kód könnyen, pár kulcsszóból álló parancsokkal hibaellenőrizhető és futtatható. Cél továbbá, hogy a modellek, amik vizuális szemléltetést igényelnek, látványos és könnyen kezelhető felületen jelenjenek meg futtatáskor.

## Rendszerkövetelmények

A program ugyan csak *Windows 10* és *11* operációs rendszerekben került precíz tesztelésre, *Windowson és Linuxon* is egyaránt futtatható. Nem igényel sem nagy memóriaterületet, sem nagy számítások végzésére alkalmas számítógépet.

## Telepítés

A program használatba helyezéséhez a **CELL.zip** fájl letöltése szükséges, melynek tartalma az alábbi:

A képen szöveg látható

Automatikusan generált leírás

2. ábra: A program mappaszerkezete

* ***appdata***: A programban megjelenített médiatartalmakat és a szoftverbeállításokat tartalmazza
* ***celldeveloper.exe*** – a futtatható szoftver (parancsikon létrehozásával a parancsikon áthelyezhető a mappából, az *exe* fájl nem!)
* ***README.txt*** – a szoftver bemutatásául szolgáló szöveges fájl

A program egyéb telepítési, letöltési folyamatokat nem igényel, az *exe* fájlt futtatva a szoftver használatkész.

A könnyebb használat érdekében célszerű a *cll* kiterjesztésű fájlokat rendszerszinten társítani a celldeveloper.exe programhoz.

## A szoftver használata

A szoftver kinézetét, ergonómiáját tekintve nagyban megegyezik a napjainkban használt általános szerkesztő programokkal. Lehetőség nyílik a segítségével átlátható, kiemelővel ellátott kódok alkotására és azok futtatására, hibaellenőrzésére.

### Felhasználói felület

A későbbiekben képekkel illusztrált szoftverelemek képernyőképei **sötét** **módban** készültek. A szoftvert elindítva az alábbi felület jelenik meg:

A képen szöveg, képernyőkép, monitor látható

Automatikusan generált leírás

3. ábra: A program felhasználói felülete

A felület négy fő részre osztható.

1. A **menüre**, ahol a megszokott szerkesztésekre, fájlkezelésre van lehetőség, melyek az alábbiak:

* Fájlkezelés
  + Új fájl létrehozása
  + Fájl betöltése
  + Fájl mentése
  + Fájl mentése másként
  + Kilépés
* Szerkesztés
  + Visszavonás
  + Ismét
  + Kivágás
  + Másolás
  + Beillesztés
* Nézet
  + Betűméret növelése
  + Betűméret csökkentése
  + Váltás világos és sötét téma közt
* Súgó
  + Első használat
  + Dokumentáció
  + Leírás

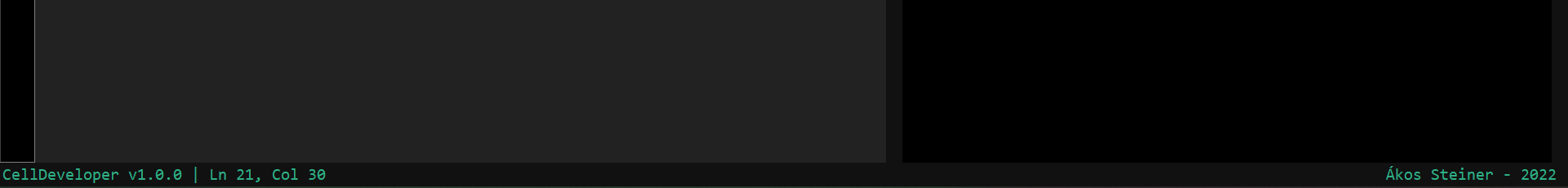
Az itt található funkciók legtöbbje billentyűkombinációkhoz kötött.

A képen szöveg látható

Automatikusan generált leírás

4. ábra: Menük felülete

1. A **kódszerkesztőre**, mely a későbbiekben kerül részletezésre (Felhasználói felület/Kódszerkesztő)
2. A **parancssorra**, mely a későbbiekben kerül kifejtésre (Felhasználói felület/Parancssor)
3. Egy **információs sávra**, ami tájékoztat a legutóbbi felhasználói akcióról (pl. mentés) és a kurzor jelenlegi pozíciójáról. Itt található továbbá a verziószám, a készítő neve és a program fejlesztésének éve.



5. ábra: Az információs sáv

#### Kódszerkesztő

A szoftver egyik fő alkotóeleme a **kódszerkesztő**. Itt nyílik lehetőségünk ***CELL*** programok készítésére. A folyamatot *lexikális elemző* segíti, mely a különböző kategóriájú *tokeneket* (lásd Nyelv leírása/Lexikális elemek) eltérő színnel és stílussal jeleníti meg. A kapott hibaüzenetek gyors detektálását a bal oldalon található **soszámozás** segíti. A felhasználót továbbá a fent található **fejléc** informálja, hogy melyik fájl van épp a szerkesztőben betöltve, illetve nem mentett módosítások esetén a program neve mellett csillag tájékoztat erről.

A képen szöveg látható

Automatikusan generált leírás

6. ábra: A kódszerkesztő

#### Parancssor

A kódszerkesztőben elkészített és elmentett programot egy-, vagy párszavas **parancsokkal** lehet futtatni, illetve hiba ellenőrizni. A program válasza az adott parancsokra különböző kategóriákra bontható, melyeket a parancssor eltérő színekkel ábrázol:

* **Zöld** színnel a felhasználói bemenetet írja a képernyőre (#2EB086)
* **Pirossal** a fordítás, vagy helytelen használat során felmerült hibaüzeneteket írja ki (#BB2F2F)
* **Lilával** a figyelmeztetéseket jeleníti meg, mint például nem mentett módosítások, vagy üres parancs futtatása (#7F4C7F)
* **Világosszürkével** az információkat írja ki, mint például a használható parancsok, vagy létező modellek a fájlban (világos téma esetén **sötétszürke**)

A parancssor felett két gomb és egy szövegbeviteli terület helyezkedik el, a futtatni kívánt parancsok az utóbbiba írandók. Evidens módon a **RUN COMMAND** felíratú gombbal futtatni tudjuk a parancsot, a **CLEAR CONSOLE** felíratúval pedig törölni a parancssor eddig megjelenített tartalmát. A parancs futtatásához az *Enter* gomb megnyomása is elegendő, ha pedig a legutóbb futtatot parancsot szeretnék ismét futtatni, a *felfelé* gombbal a szövegdobozban megjelenik a legutolsó bemenet.

A képen szöveg látható

Automatikusan generált leírás

7. ábra: A parancssor

#### Modellek futtatása

Két különböző felület áll rendelkezésre a modellek szemléltetésére. Az egyik az n szalagos *Turing-gépek futtatására* szolgál, míg a másik az összes többi – hasonló szerkezetű – modellek megjelenítéséért felelős. Ezen felületek **fix ablakmérettel** rendelkeznek, átméretezésre, teljes képernyős módra nem alkalmasak, de a sejtautomaták esetén **a cellák mérete skálázható**.

A **Turing-gépek** használatának felülete az alábbi:

A képen szöveg látható

Automatikusan generált leírás

8. ábra: Turing-gépek futási felülete

A szoftver jelenlegi állapotában csak **maximum négyszalagos** (mely általános problémák megoldására az esetek nagyrészében elegendő) gépek programozására alkalmas, így a megjelenítés is eszerint készült. Ugyan mind a négy szalag, és a hozzá tartozó beviteli mező (előző jobb, utóbbi bal oldalon jelenik meg) megtalálható a felületen, de a futtatott géphez alkalmazkodva dinamikusan csak annyi beviteli mezőbe írhatunk, amennyi szalaggal rendelkezik a gép, valamint a nem használt szalagokon nem jelennek meg **olvasófejek** (melyeket zöld **V** karakter szimbolizál). A felület bal felső részén továbbá a **futtatásra,** lépésenkénti **iterációra**, és a szalagok **kezdőállapotba helyezésére** van lehetőség, a beviteli mezők alatt pedig a megadott karakterek szalagokra való feltöltése és a mezők kiürítése lehetséges a megfelelő gombok megnyomásával.

A **sejtautomaták** megjelenítésére szolgáló ablak az alábbi struktúrával rendelkezik:

A képen szöveg, óra, eredményjelző tábla, megjelenítés látható

Automatikusan generált leírás

9. ábra: Sejtautomaták futási felülete

Minden, a fent látható ablakban megjelenített modell futtatására igaz, hogy a bal oldalon található **vezérlőpanellal** irányíthatjuk az iterációk menetét. Külön gomb áll rendelkezésre futtatás **elindítására**, **megállítására**, a cellák **kezdőállapotba helyezésére**, iterációnkénti **léptetésre**, valamint a futási **sebesség változtatására**. Utóbbi a többitől eltérően kevésbé intuitív megvalósítást kapott, ugyanis minden kattintással nő a sebesség, illetve mikor a maximális sebességet elérte, a minimálisra vált.

A legtöbb automata **beégetett kezdőállapottal** rendelkezik. Ez alól kivétel a kétdimenziós sejtautomata, ahol ugyanis a cellákra jobb, illetve bal **egérkattintással** rendre inaktívra, illetve aktívra állíthatjuk az adott cellák értékét. Ebben az esetben az aktív cellák kékek, az inaktívak sötétszürkék, azonosan az egydimenziós automatákhoz, ugyanis a szoftver jelenlegi formájában ezen modellek csak két állapotú implementációjára alkalmas. Hangya modellek esetében viszont tetszőleges mennyiségű állapotot vezethetünk be, melyekhez előre beégetett halmazból oszt ki a program színeket ciklikusan, így bizonyos mennyiségű állapotot követően egyes állapotok színe **megegyezik**. Erre példa az alábbi:

A képen szöveg, kültéri, óra, aláírás látható

Automatikusan generált leírás

10. ábra: Eltérő állapotok eltérő színnel való ábrázolása

### Hibaüzenetek

A programon belül fellépő hibaüzenetek három csoportra oszthatóak.

A kód megírását követően modellek futtatásakor, illetve hibaellenőrzéskor a parancssoron jelennek meg az esetleges **fordítás idejű hibák**. Ide tartoznak a nem megfelelő lexikális elemek, vagy szintaktikailag helytelenül felépített modellek, de a többszöri azonosító használat és a nem megfelelő típusú attribútumok is. Ezeket a parancssor a következő formában jelzi:

A képen szöveg látható

Automatikusan generált leírás

. ábra: Fordítás közben keletkezett hibák

Az első esetben lexikális hibát jelez a fordító, míg a másodikban nem megfelelő paramétert.

Abban az esetben is a parancssorra kerülnek ki a hibaüzenetek, ha helytelen parancsot próbálunk futtatni. Erre példa a helytelen parancs paraméterezés, vagy ismeretlen parancs bevitele:

A képen szöveg látható

Automatikusan generált leírás

. ábra: Helytelen parancsfuttatás által keletkezett hibák

Modellek **futtatása közben** kizárólag Turing-gépek esetében léphet fel hiba. Mivel a szalagok hossza véges a szoftver jelenlegi verziójában, így amennyiben bevitelkor, vagy bevitel után futtatás közben valamelyik olvasófej **elérné a hozzá tartozó szalag végét**, azt az állapotot jelző sávban írja ki a program. Amennyiben pedig olyan állást vesz fel a gép, amire **nincs megfelelő következő lépés**, és a program nem került elfogadó állapotba, szintén hibát delegál – azonos módon az előbb említetthez.

A képen szöveg, képernyőkép, elektronika látható

Automatikusan generált leírás

. ábra: Turing-gépek futtatása közben keletkezett hiba

A többi modell esetében nem lép fel futás közbeni hiba, amennyiben a folyamat a mátrixon túli indexekre próbál hivatkozni, a futás **automatikusan leáll**.

## A nyelv leírása

A forráskódban szereplő szövegeket saját nyelvfelismerő program értelmezi, melynek végeredménye egy kupac (heap), megy az interpretált modelleket tartalmazza. A folyamat lépésekre lebontva az alábbi:

1. Először egy **lexikális elemző** *tokenizálja* a forráskódot, aminek a végeredménye egy token lista. Ebben a lépésben kerülnek felismerésre a kulcsszavak, azonosítók, illetve számok, logikai értékek is.
2. A következő lépésben a **szintaktikai elemző** *szintaktikai fát* épít a *tokenekből*, így születnek meg az egyelőre értelmezetlen, de helyes adatszerkezetbe rendezett modellek.
3. Az értelmezetlen adatstruktúrákból a **szemantikus elemző** készít végül futtatható modelleket típusokkal, attribútumokkal és azonosítókkal kapcsolatos hibák kiszűrésével együtt.

Attól függően, hogy a fordító melyik szakaszában keletkezik probléma, azt a program **lexikális**, **szintaktikai**, illetve **szemantikai** (parse) hibaként delegálja.

A forráskód szövegét egy, a szoftverhez készített értelmező program alakítja futtatható modell példányokká. Az elfogadott nyelv szabályrendszere alább található.

### Lexikális elemek

A nyelv legalsó szintjén a **szavak és írásjelek** találhatók, melyek gyűjtőfogalma a *token*. Egy modell felismerésekor a program három különböző típusú *tokent* ismer fel.

A **konstansok** csoportjába tartozik minden olyan szöveg, ami csak **pontos** **egyezés** esetén feleltethető meg a hozzá tartozó *tokennel*. Ilyenek a modelltípus nevek, valamint a beégetett értékek iránytípusokhoz.

* Modelltípusok
  + Turing
  + Turmite
  + Automata1D
  + Automata2D
* Irányértékek
  + R (right)
  + L (left)
  + S (stay)
* Egyéb
  + \_ (none)

A következő csoportba **reguláris kifejezések** alkotta szabályok alapján kerülnek *tokenek*. Ilyenek az egészszámok, valamint az azonosítók.

* Egészszámok
  + nullával nem kezdődő bármely egészszám és a nulla
* Azonosítók
  + kis- vagy nagybetűvel kezdődő, számokat, kis- és nagybetűket opcionálisan tartalmazó szövegek

*A reguláris kifejezések kifejtése és részletes tárgyalása a fejlesztői dokumentációban található.*

A konstansokat, azonosító- és számértékeket **szimbólumok** egészítik modellekké. Ezek legtöbbször egy vagy két **speciális karakterből** álló szövegek, amelyek meghatározzák egyrészt, hogy a modellalkotás melyik fázisában járunk, valamint a tömbök, karakterek képzésében is segítik a felhasználót.

|  |  |
| --- | --- |
| :: | modelldefiníció szimbóluma |
| | | attribútumdefiníció szimbóluma |
| : | attribútumdeklaráció szimbóluma |
| ; | szintaktikai cukorka lista attribútum deklarálásához |
| [ ] | lista szimbólumai |
| ’ | karakter közrefogó szimbóluma |
| => | szabály szimbóluma |
| , | listaelemeket elválasztó szimbólum |

A tömör megfogalmazás jegyében a szimbólumok pár szavas leírása nem minden esetben teljesen egyértelmű, a nyelv racionális, célszerű használatához javasolt a **mellékelt tesztprogramok** átvizsgálása.

### Szintaxis

A különböző *tokenekből* „mondatok” állnak össze, ezeket fordító programok szintjén ***szintaxis fának*** nevezzük. A különböző adatszerkezetek az alábbi módon épülnek fel:

#### modell

azonosító :: Modeltípus

| attríbútum : érték

| lista attribútum

; érték

; érték

…

…

#### szabály

kezdő\_állapot => vég\_állapot

#### listaérték

[érték, érték, érték, …]

## Modellek

A szoftver jelenlegi verziója **négy beépített modellt** tartalmaz, melyek az alábbiak:

* Turing-gépek
* Hangya modellek
* Egydimenziós sejtautomaták
* Kétdimenziós sejtautomaták

A nyelv megírásának folyamán lényeges szempont volt, hogy könnyen bővíthető és módosítható legyen, így a későbbiekben egyéb automaták integrálása is megvalósítható (lásd Továbbfejlesztés/ Integrálandó modellek).

A program használata feltételezi az integrált modellek alapvető ismeretét felhasználói részről (valamint a bevezetőben ezeknek a bemutatására más sor került), így a későbbiekben a fent említett automaták csak programspecifikusan kerülnek tárgyalásra.

### A képen szöveg látható Automatikusan generált leírásTuring gépek

14. ábra: Turing gép vizualizációja

A **CellDeveloper** szoftverén belül determinisztikus, legfeljebb négyszalagos Turing-gépek készítése valósítható meg. Nem megfelelő állapotba jutás esetén a program jelez, ahogy akkor is, ha elfogadó állapotba ért, így a Turing-gépek helyességi vizsgálata teljes mértékben kivitelezhető a programon belül. Az olvasó fejek a kiindulástól jobbra, és balra is mozognak, a bevitt karakterek azonban minden inicializálásnál az olvasófejektől jobbra kerülnek a szalagra. A bevihető karakterek, valamint a szalagok hossza véges, de hétköznapi használathoz elegendő.

### Hangyamodellek

A képen szöveg látható

Automatikusan generált leírásA hangyamodellek integrálása során fontos szempont volt, hogy a cellák által felvehető állapotok száma ne legyen limitálva, így a jelenlegi verzió szerint tetszőleges számú állapot felvehető, annak jelzése ciklikusan ismétlődő színekkel kerül megvalósításra. Az implementált automata mindig a középső cellából indul ki, és amikor eléri a cellarács szélét, leáll.

15. ábra: Hangyamodell vizualizációja

### A képen zöld, világos látható Automatikusan generált leírásEgydimenziós sejtautomaták

16. ábra: Egydimenziós sejtautomaták vizualizációja

Működésüket tekintve az egydimenziós automaták nagyban hasonlítanak a hangyamodellekhez. Programfuttatáskor fix (felső, középső) cellából indulnak ki, és mikor a rács alját elérik, leállnak. Minden iteráció a hozzá tartozó sorszámú sorban található (első az elsőben, második a másodikban stb.). Az implementáció során elegendő megadni csak azokat az állapotokat, amik inaktív cellát eredményeznek, vagy csak azokat, amelyek aktív cellát. Ennek eldöntése a szabályok arányával mérlegelve a felhasználótól függ.

### Kétdimenziós sejtautomaták

A képen szöveg, zöld, óra, világos látható

Automatikusan generált leírásA kétdimenziós automaták az egydimenziósakhoz hasonlóan egyelőre csak aktív, illetve inaktív állapotot tudnak megkülönböztetni, így egyed modellek (pl. Brian’s brain) nem megvalósíthatóak a nyelvben. Az vizsgált mező szomszédjainak értékei súlyozhatóak, ilyen módon az implementáció több lehetőséget nyújt komplexebb modellek megalkotására is (ilyen módon például egydimenziós automaták is készíthetők). Futtatáskor kezdetben minden cella inaktív, bal egérgombbal aktívvá, jobbal ismét inaktívvá változtathatjuk a kívánt mezőket.

17. ábra: Kétdimenziós sejtautomaták vizualizációja

# Fejlesztői dokumentáció

## Megoldási terv

A szoftver célszerű működtetésének érdekében a fordító program egy ***python Tkinter projektben*** kerül meghívásra, hogy a szerkesztőprogram közvetlenül elérje annak osztályait és függvényeit. Az elkészített kódok szövegét rendre dolgozza fel először a lexikális, majd a szintaktikus elemző, végül a parser, majd az így kapott, helyesen interpretált modellek a **Tkinter**, esetenként **Pygame** programok felületén futtathatók. A fordító végül **python** programot generál, így gépi kódalkotást nem alkalmaz fordítás során.

### Programszerkezet

A projekt programrésze az alábbi mappaszerkezetet követi:

A képen asztal látható

Automatikusan generált leírás

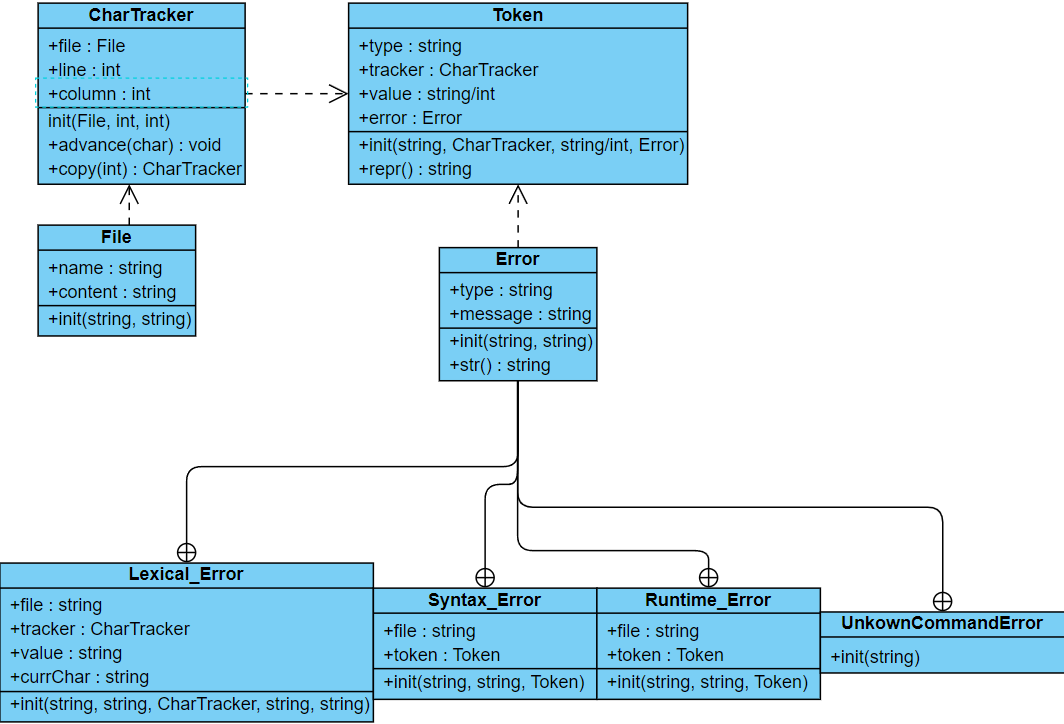
. ábra: A program tervezett mappaszerkezete

Az ***appdata*** mappában találhatóak a megjelenített médiatartalmak, mint például a program ikonja, az üdvözlő ablak képe, valamint gombok és mezők vizualizációja. Ezen kívül az ablak beállításai is itt kerülnek eltárolásra a ***data.json*** fájlban.

A program lényegi részét a ***source*** mappa tartalmazza. A fordításért felelős programok a következők:

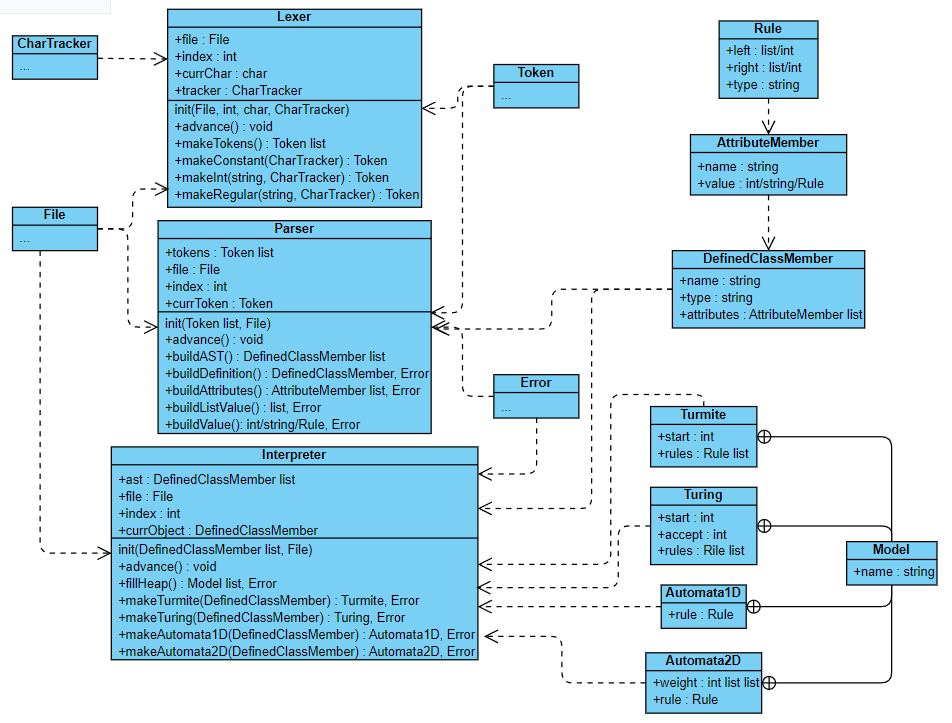
* lexer.py
* cellparser.py
* interpreter.py
* celltoken.py
* tracker.py
* error.py
* cellclass.py

A fordítás első állomásaként a szövegből *tokenek* készülnek, melyeknek **UML diagramja** alább látható:



. ábra: A token osztály tervezett UML-diagramja

A fordítás lényegi részét a ***Lexer***, a ***Parser*** és az ***Interpreter*** végzi, ezen fájlok szerkezet ábrája az alábbi:



. ábra: A fordító tervezett UML-diagramja

Mivel a **python** programozási nyelv konvencióit tekintve *több programozási paradigma* elvét is ötvözi, ezért szoftver felhasználói felületéért felelős kód *nem objektum orientált* szemléletben készült, így **UML diagrammal** nem reprezentálható, de függvényeiről és működési elvéről részletes leírás olvasható a Megvalósításban. Osztályok csak az ablak témájára, illetve betűszín témákra készült, melyek az alábbiak:

A képen szöveg látható

Automatikusan generált leírás

. ábra: Betűtéma osztály szerkezete

A képen szöveg látható

Automatikusan generált leírás

. ábra: Témaosztály szerkezete

## Megvalósítás

A megoldási tervtől a megvalósítás során releváns mértékben nem volt szükséges, fájlok, osztályok neve az evidencia jegyében helyenként módosult, így a ***lexer, parser, interpreter*** hármasból végül ***lexical, syntax, semantic hármas*** lett.

### Fordító

Az elemzők szabványszinten hasonlóan működnek, mint a fordítóprogramok többsége, így csak a kérdéses implementációk kerülnek tárgyalásra a későbbiekben.

#### Lexikális elemző

A **lexikális elemző** a fájl szövegét karakterről karakterre olvasva *tokenizálja* azt. A megengedett karaktereknek van egy halmaza, amik egy adott szó végét jelzik. Ilyenek a szóközök, tabulátor, sortörések, valamint a nyelv által használt szimbólumok.

continousCharacters = ",;:;='|[]> \n\t"

Reguláris kifejezéseket az egészszámok, azonosítók és karakterek leírására volt szükséges bevezetni, ezek a következők:

|  |  |
| --- | --- |
| Típus | Reguláris kifejezés |
| Egészszámok | 0|[1-9][0-9]\* |
| Azonosítók | [a-zA-Z\_][a-zA-Z0-9\_]\* |
| Karakterek | [a-zA-Z0-9\_] |

A lexikális elemző először a szóvégeket jelző karaktereknél fogva elválasztott szavakat **szimbólumokként** próbálja értelmezni. Amennyiben a nyelvben nem elfogadott karaktert tartalmaz, egyből hibát delegál a fordító. Ha nem sikerül szimbólumnak, a **konstansok** valamelyikének próbálja megfeleltetni. Amennyiben konstansnak sem feleltethető meg az aktuális szó, következőnek **egészszámként** vizsgálja, végül **azonosítóként** értelmezi.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| szimbólum |  | konstans |  | egészszám |  | azonosító |

#### Szintaktikai elemző

A **szintaktikai elemző** a felismert *tokeneket* sorra próbálja beilleszteni a *szintaxisfába*, ami alapján a modellek képződnek. Mivel csak egy modell szerkezet elfogadott, az elemzőnek kevés elágazás lehetséges végződéseit kell megvizsgálnia. A nyelvi elemek egy modellben rendre a következők:

1. azonosító
2. definíció szimbólum
3. típus
4. véges mennyiségű attribútum deklaráció
   1. attribútum szimbólum
   2. attribútum azonosító
   3. érték deklaráció
      1. egyszerű érték
         1. deklaráció szimbólum
         2. érték
            1. számérték
            2. karakterérték

karakter szimbólum

karakter

karakter szimbólum

* + - * 1. listaérték

listanyitó szimbólum

érték, vessző véges mennyiségben

érték

listazáró szimbólum

* + - * 1. szabály

baloldal

érték

…

szabály szimbólum

jobboldal

érték

…

* + 1. listaérték
       1. véges mennyiségű lista deklaráció
          1. listadeklaráció szimbólum
          2. listaérték

…

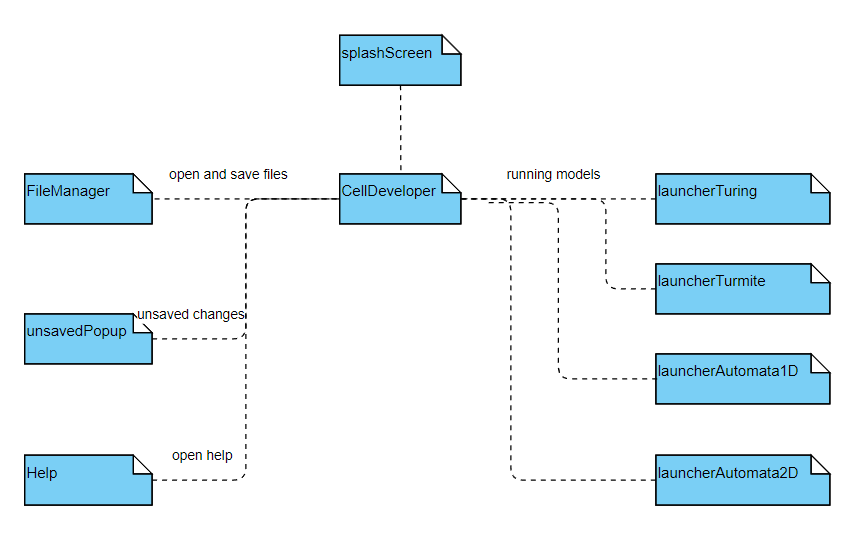
#### Szemantikus elemző

A már megalkotott példányokon a **szemantikai elemző** értelmezési vizsgálatokat végez.

1. Először megvizsgálja, hogy az adott példány azonosítója már szerepel-e a kupacban. Ha igen, hibát delegál
2. Ezt követően megvizsgálja, hogy a példány típusa megfelelő-e
3. Miután a típusvizsgálat megtörtént, az adott típus attribútumvizsgálatai következnek
   1. Ellenőrzi, hogy minden szükséges, és csak a szükséges attribútumok pontosan egyszer kerülnek-e deklarálásra. Amennyiben ez a feltétel nem teljesül, azt szintén hibával jelzi
   2. Végül a megfelelő attribútumok értékeit vizsgálja
      1. Ellenőrzi az értékek típushelyességét, valamint listák esetén az értékek mennyiségét is

### Szoftver

A felhasználói felület ***python Tkinter*** és ***python Pygame*** grafikai modulok segítségével került megvalósításra, a nyelv paradigmai keverékéből adódóan az objektumorientáltságot nélkülözve. A megjelenített ablakok szerkezete az alábbi:



23. ábra: A felhasználói felület ablakszerkezete

A szerkesztőben elkészített modelleket minden változtatáskor **lexikális elemző** vizsgálja át, így a **lexikális szövegkiemelés** teljes mértékben dinamikusan történik. Ennek kódja az alábbi:

lexer = lexical.Lexer(ctracker.File(fileName, editorText.get("1.0", "end"))) # analizing lexically

        tokens, error = lexer.makeTokens()

        if tokens: # non error occured

            for token in tokens:

                line, column = token.tracker.line, token.tracker.column

                if token.value: length = len(str(token.value))

                else: length = len(token.type)

                if token.type in classes:

                    editorText.tag\_add("class", f"{line}.{column}", f"{line}.{column+length}")

                elif token.type in defines:

. . .

A kód a ***lexer*** osztály felhasználását és az elemző segítségével generált *tokenek* megkülönböztetésének folyamatát is jól szemlélteti.

### Modellfuttatás

A modellek szabályrendszereinek és *iterácionkénti* állapotváltozásának logikája a hozzájuk tartozó futtató fájlokban található. Erre a sejtautomatákban rendszerint az ***iter()*** és a ***neighbours()*** függvények szolgálnak. Működésüket tekintve az alábbiak:

Az ***iter()*** függvény a példány szabályai közül megkeresik, melyiknek a bal oldalára illik rá a jelenlegi állapot, majd attól függően változtatják a következő ciklus állapotát, ezt grafikai változtatásokkal is jelezve.

A ***neighbours()*** függvény az ***iter()*** függvényben meghívva kikalkulálja, hogy az éppen vizsgált mező szomszédjai milyen értékeket vesznek fel, és ezeket summázza.

A sejtautomatáktól eltérően a Turing-gépek az erre a célra szolgáló ***stepCommand()*** függvényben végzik el a szükséges számításokat a következő állapot felállításához.

# Tesztelés

A programot tesztelés szempontjából két elkülöníthető részre bonthatjuk. Az első a **fordító** helyes működésének teszteléséért felelős, a másik pedig a szoftver **felhasználói felületének** funkcióit ellenőrzi.

## Fordító tesztelése

A fordító helyes működésének ellenőrzését **egységteszteléssel** végezzük. A felmerülő problémákat ezen belül. Lexikális, szintaktikai és szemantikai problémák merülhetnek fel egy forráskód fordítása során, valamint ezeken felül érdemes stressz tesztelni a fordítót, hogy kiderüljön, nagy mennyiségű szöveget is rövid idő alatt képes-e feldolgozni.

A modell példányok implementálásának megfelelő feldolgozása mellett ellenőrizhető a lexikális szövegkiemelő helyes működése is, mely minden tesztfájl megnyitásakor tanúbizonyságot nyer.

### Helyes tesztek

A fordító helyes működésének vizsgálatához minden modellhez készült egy teszt fájl, amely legalább 4 darab modell példányt tartalmaz. Ezek közül valamelyik egyszerűbb, valamelyik komplexebb. Ezen kívül a nagy terjedelmű fájlok kezelésére és a szintaktikai cukorka tesztelésére is készültek teszt fájlok. A helyes tesztek és azok tesztelési célja alább található táblázatos formában.

|  |  |
| --- | --- |
| Tesztfájl | Tesztelés célja |
| models.cll | több, különböző modell példányainak implementálása megfelelően működik |
| turing.cll | több, eltérő szalagszámú példány készítése és futtatása megfelelően működik |
| turmite.cll | több, eltérő mennyiségű példány készítése és futtatása megfelelően működik |
| automata1D.cll | több különböző példány készítése és működése megfelelően működik |
| automata2D.cll | több különböző példány készítése és működése megfelelően működik |
| longFile.cll | nagy mennyiségű szöveges adat gyors, problémamentes feldolgozása és a lexikális szövegkiemelő helyes működése |
| syntacticSugar.cll | lista attribútumok eltérő deklarálása azonos módon kerül értelmezésre |

### Helytelen tesztek

Helytelen tesztesetek vizsgálatával kiszűrhetőek a nem kívánt programműködési folyamatok, ezek feltérképezésére mind lexikális, szintaktikai, mind szemantikus hibákat tartalmazó forráskódokat táplálunk be a szoftvernek. Megfelelő működés esetén előre tudható, hogy milyen hibaüzenettel fogja jelezni a program a kód helytelenségét. Ezek az alábbi táblázatokban találhatók.

#### Lexikális hibák

Lexikális hibának minősülnek a nem tokenizálható szavak, karakterek. Ezek ellenőrzésére négy tesztfájl készült.

|  |  |  |
| --- | --- | --- |
| Tesztfájl | Hibaüzenet | Tesztelés célja |
| illegalNumber.cll | [!] Lexical\_Error : Illegal number in file illegalNumber.cll, line 2 : '012' | Nullával kezdődő egészszámok kiszűrése és nem elfogadott számábrázolás jelzése |
| illegalWord.cll | [!] Lexical\_Error : Illegal word in file illegalWord.cll, line 3 : '1one' | Számmal nem kezdődhetnek azonosítók, így az ilyen tokenek esetén a program illegal word hibát delegál |
| invalidCharacter.cll | [!] Lexical\_Error : invalid character in file invalidCharacter.cll, line 1 : '?' | Nem megfelelő karakterek használata a programban (pl. „?”) |
| invalidSymbol.cll | [!] Lexical\_Error : Invalid symbol in file invalidSymbol.cll, line 2 : '=' | Olyan szimbólum használata, mely megengedett speciális karaktereket tartalmaz, de aktuális formájában nem megengedett kompozíció, vagy önmagában nem megengedett karakter. Ilyen a „=>” helyett alkalmazott „=” |

#### Szintaktikai hibák

Az alábbi hibás fájlok a programnyelvtől eltérő szerkezetben írt kódokat, és azok hibaüzeneteit vizsgálják.

|  |  |  |
| --- | --- | --- |
| Tesztfájl | Hibaüzenet | Tesztelés célja |
| id.cll | [!] Syntax\_Error : missing ID for definition in file id.cll, line 1 | Hiányzó példány azonosító kezelése |
| definitionSymbol.cll | [!] Syntax\_Error : missing definition symbol [::] in file definitionSymbol.cll, line 1 | Hiányzó definíció szimbólum („::”) kezelése |
| class.cll | [!] Syntax\_Error : Unknown or missing classname in definition in file class.cll, line 2 | Hiányzó, vagy nem megfelelő modell típus használata |
| attributeSymbol.cll | [!] Syntax\_Error : missing symbol [|] for attribute declaration in file attributeSymbol.cll, line 2 | Hiányzó attribútum szimbólum („|”) kezelése |
| attribute.cll | [!] Syntax\_Error : missing attribute name in file attribute.cll, line 2 | Hiányzó attribútumnév kezelése |
| declarationSymbol.cll | [!] Syntax\_Error : missing symbol [:] or [;] for attribute declaration in file declarationSymbol.cll, line 2 | Hiányzó attribútum deklarációs szimbólum („:” vagy „;”) kezelése |
| value.cll | [!] Syntax\_Error : missing value in file value.cll, line 3 | Hiányzó attribútum érték kezelése |
| list.cll | [!] Syntax\_Error : incorrect list declaration in file list.cll, line 4 | Lista érték hibás deklarálásának kezelése |
| rule.cll | [!] Syntax\_Error : incorrect rule declaration in file rule.cll, line 4 | Szabály érték hibás deklarálásának kezelése |
| char.cll | [!] Syntax\_Error : missing symbol ['] for character declaration in file char.cll, line 4 | Karakter érték végéről hiányzó szimbólum („’”) kezelése |

#### Szemantikai hibák

Amennyiben egy modell egyes attribútumai hibás értékekkel kerülnek deklarálásra, vagy nem megfelelő, esetleg hiányzó attribútumokkal vannak ellátva, arról a szemantikai elemző tájékoztak. Helyes működését az alábbi tesztek hivatottak ellenőrizni:

|  |  |  |
| --- | --- | --- |
| Tesztfájl | Hibaüzenet | Tesztelés célja |
| missing.cll | [!] Parse error : missing attribute name rules in model 'test' | Hiányzó attribútum kezelése |
| unknown.cll | [!] Parse error : unkown attribute name something in model 'test' | Nem megfelelő attribútum kezelése |
| duplicated.cll | [!] Parse error : attribute is already declared 'start' in model 'test' | Duplikált attribútum deklaráció kezelése |
| length.cll | [!] Parse error : Not consistent length of lists in model 'test' | Nem konzisztens listahossz kezelése |
| value1.cll | [!] Parse error : int required in attribute 'start' in model 'test' | Nem megfelelő típusú érték kezelése |
| value2.cll | [!] Parse error : character or NONE value required in first list of rules in model 'test' | Nem megfelelő típusú érték kezelése |

## Felhasználói felület tesztelés

A felhasználói felület helyes működését **end to end** tesztelési terv készítésével lehet a leghatékonyabban tesztelni. A legtöbb felhasználói interakció által kiváltott válasz a program részéről determinálható.

### Menü tesztelése

#### Fájlkezelés tesztelése

|  |  |
| --- | --- |
| Új fájlra kattintás / Ctrl+N | a szerkesztő tartalma törlődik, a fájl neve törlődik, amennyiben a fájl mentve van, különben nem mentett dialógusablak ugrik fel |
| Fájl megnyitásra kattintás / Ctrl+P | Fájlkezelő dialógusablak ugrik fel, amennyiben a fájl mentve van, különben nem mentett dialógusablak ugrik fel |
| Fájl mentésre kattintás / Ctrl+S | Korábban létező fájl esetén a fájl tartalma felülíródik, nem létező esetén Fájl mentése másként dialógusablak ugrik fel |
| Fájl mentése máskéntre kattintás | Fájlkezelő dialógusablak ugrik fel |
| Kilépésre kattintás / Alt+F4 / Program bezárása | A program bezáródik, amennyiben a fájl mentve van, különben nem mentett dialógusablak ugrik fel |

#### Szerkesztés tesztelése

|  |  |
| --- | --- |
| Visszavonás gombra kattintás / Ctrl+Z | Az előző szerkesztő állás jelenik meg |
| Mégis gombra kattintás / Ctrl+Y | A visszavont változtatás újra megtörténik |
| Kivágásra kattintás / Ctrl+X | Kijelölt szöveg kivágása |
| Másolásra kattintás / Ctrl+C | Kijelölt szöveg másolása |
| Beillesztésre kattintás / Ctrl+V | Korábban kijelölt szöveg beillesztése a kurzor pozíciójától kezdődően |

#### Nézet tesztelése

|  |  |
| --- | --- |
| Nagyítás | A szerkesztő és a sorszámjelző betűmérete nő, amennyiben nem érte el a maximumot |
| Kicsinyítés | A szerkesztő és a sorszámjelző betűmérete csökken, amennyiben nem érte el a minimumot |
| Témaváltás | A felület témája megváltozik a kiválasztottra – amennyiben nem abban volt eddig is |

#### Súgó tesztelése

### Kódszerkesztő tesztelése

### Parancssor tesztelése

### Sejtautomata felület tesztelése

### Turing-gép felület tesztelése

# Továbbfejlesztés

A szoftver és a fordító megírásakor nagy hangsúly került a bővíthetőségre, így újabb tokenek, szintaktikai ágak és modellek is könnyen integrálhatóak, valamint a felhasználói felület is bővíthető néhány, a felhasználói élményt javító funkciókkal.

## Implementálandó modellek

A már integrált sejtautomaták körét bővítendő, hasonló vizualizációs felülettel egyszerűen integrációra kerülhetnek rokon modellek, ezek a következők:

### Neurális sejtautomaták

Szerkezetük hasonló a kétdimenziós sejtautomatákéhoz, a szomszédos cellák hasonló módon súlyozva vannak, de ez a súly megegyezik a cella értékével, így nem csak egy bites értékadás lehetséges. Ezen felül a következő iterációhoz kiszámolt cellaérték a szomszédok összegétől függően egy – akár komplex függvény segítségével kiszámolt – érték is lehet, valamint lehet tört és negatív szám is.

A modell beépítéséhez elég volna néhány új token megalkotása, ezek a következők:

* konstansok
  + neurális sejtautomata típus
* szimbólumok
  + függvény számításhoz használatos operátorok
  + függvénydeklaráció szimbóluma
* reguláris kifejezések
  + tört számok
  + negatív számok

### Blokk automaták

A neurális automatákhoz hasonlóan a blokk automaták is a kétdimenziós modellekre hasonlítanak. Minden iterációban egy 2x2-es terület értékkompozíciója alapján az adott terület egy új érték kompozíciót kap. Így például egy [0,1,2,3] indexű mező négyes lehetséges értékkompozícióján és a rákövetkező iterációban felvett értékkompozíciója alapján egy szabály a következő:

[0,0,1,1] => [1,0,0,0]

Az értékváltoztatás után a rács, ami alapján a 2x2-es blokkokat vizsgáljuk eggyel eltolódik vertikálisan és horizontálisan, így a következő iterációban másik 2x2 mező fog képezni egy kompozíciót.

A blokk automaták integrálásához fordítói szinten nincs szükség új nyelvi elemek bevezetésére a típuson kívül, a szemléltető felület pedig minimális szinten módosulna a kétdimenziós sejtautomatákéhoz képest.

### Membránrendszerek

A membránrendszerek egy, a sejtautomatákhoz hasonló számítási modell csoportot alkotnak, melyben különböző membránok, és azokon belül elhelyezkedő értékek sokasága alkot állapotot, valamint az átmenet-függvény az egységek membránokon belüli elhelyezkedésétől és azok közti áthaladásról, értékváltozástól függ.

A modell implementálásához egy új vizualizációs felület implementálására van szükség, valamint néhány új nyelvi elem bevezetésére is, mivel nagyban eltér a korábban tárgyalt modellek szerkezetétől.

## Program funkciók

A felhasználói felületet tekintve néhány apróbb fejlesztés javítaná a szoftver felhasználói élményét. Ezekre pár példa:

* Szöveg kijelölés, Ctrl+LMB és egyéb billentyű kombinációk támogatása
* Lexikális szövegkiemelő helyett szintaktikai szövegkiemelő használata
* Példányfuttatás sebességének könnyebb vezérlése

# Összefoglalás

# Irodalomjegyzék